FrontPage:

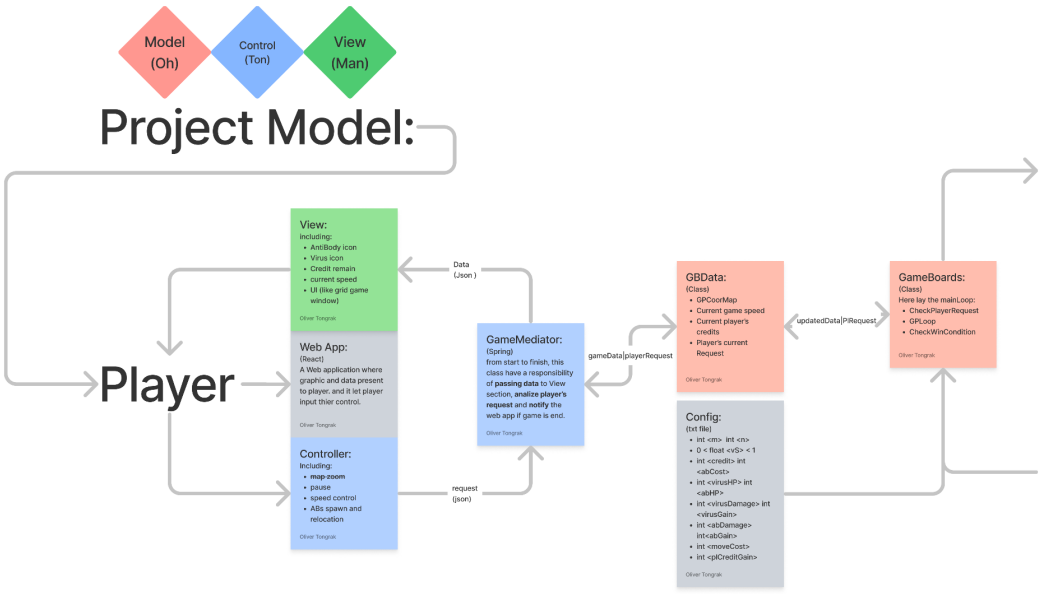
CPE200: Proj: CARIN:
Group: YukonGold:
Progress Report:

Progress Report:

Architecture:

Yet presented project's component:

Model Section:



(Figure1: Project Model Diagram:)

Note: Model section is limited to the back-end.

GBData:

a class representing a data need for front and processed request from front-end to *GameBoards*(class), such as all *game-piece*(virus or antibody) and its current *coor*(a pair of integers representing coordinates in the *game-board*), current game pause status, current game speed, current player's credit, etc. This class implementation ensures that *GameBoard* can focus with its mainloop and also ensures that the front-end can request any game data at any time during running.

Config File Integration:

As project spec required, an initial essential game variables configuration shall be implemented. This means, an initial config to *GameBoard*'s variable was added and set across classes in the Back-End.

GameBoard change in loop implementation:

GameBoard, a class response for the game loop, had been changing its looping method from *simple looping*(which is just a method that will loop for the entire duration of the game) to looping as a thread.

Architecture: Yet presented project's component:

Controller Section:

Back-End:

GameMediator: Class used in Mapping paths used to communicate with the Front-end.

GameData: A class used to store data for sending to the front-end.

ModelRequest: Format of receiving information from the front-end.

CarinApplication: Added the addCorsMappings function that makes it possible to The back-end to communicate with the front-end (allowedOrigins).

```
    @Bean
    public WebMvcConfigurer corsConfigurer() {
        return new WebMvcConfigurer() {
            @Override
            public void addCorsMappings(CorsRegistry registry) {
                registry.addMapping("/*").allowedOrigins("http://localhost:3000");
            }
        };
    }
}
```

Function addCorsMappings

Front-End:

Start button: get API from Back-end and back-end orders the game to begin.

Pause button: get API from Back-end and back-end paused the game.

Speed button: get API from Bacl-end and The back-end adjusts the speed.

```
const start = () => {
  axios.get("http://localhost:8080/start")
    .then(res => {
      console.log(res.data)
    })
}
```

Function(Start button) get API from bak-end

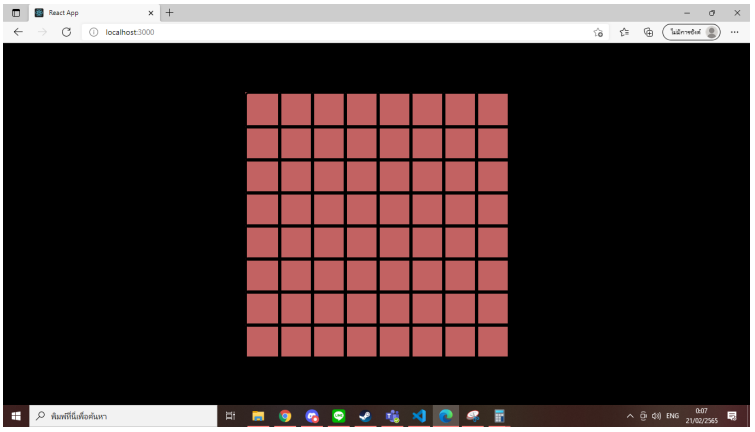
Architecture: Yet presented project's component:

View Section:

Web app's components: Import React to our workspace in FrontEnd. Adjust App.js to be suitable for our project. Create a necessary file that we need.

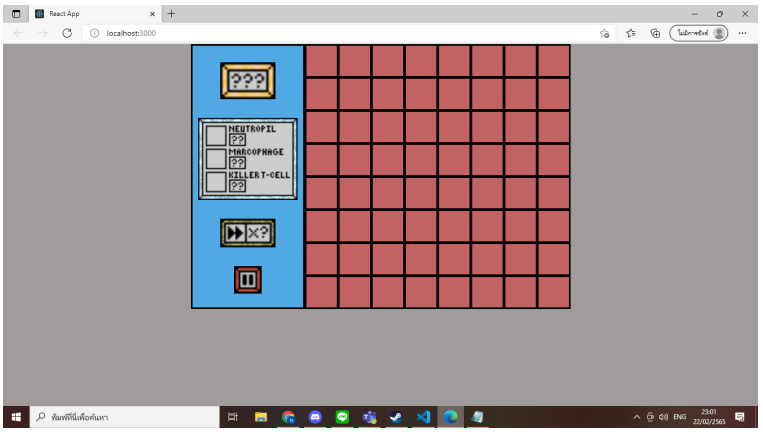
Such as:

- Board.js & .css (Generate a non interactive board gameplay to the website)



First version of website

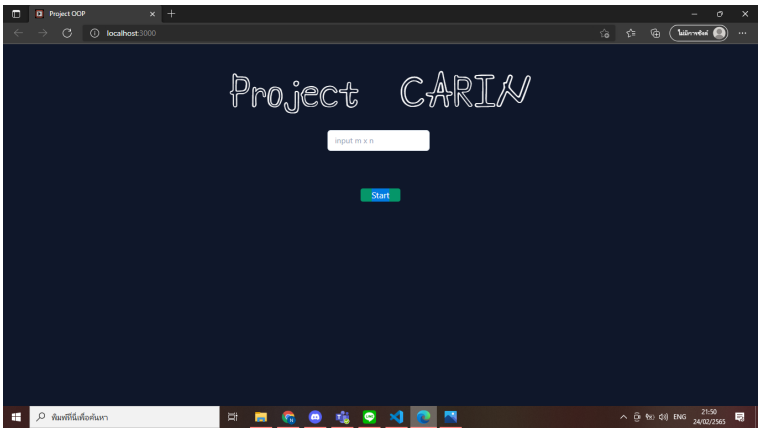
- Menu.js & .css (Generate a non interactive side menu to the website)
- Inside Menu also include:
    - Credit.js & .css
    - Shop.js & .css
    - Speed.js & .css
    - Pause.js & .css



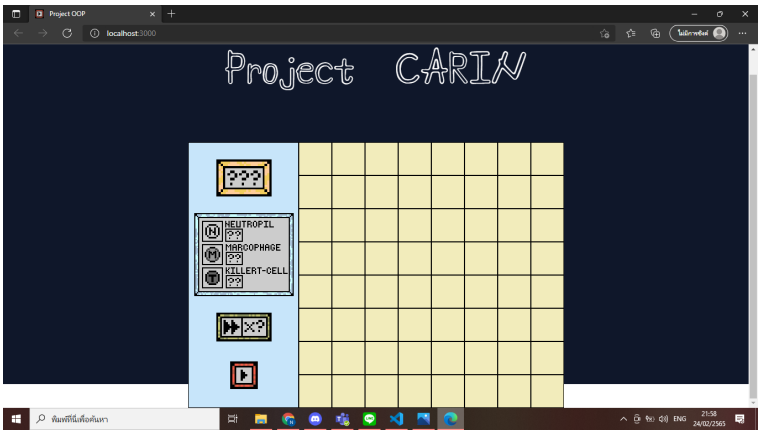
Second version of website

Next, we adjust and improve the look of our website.

This is the current look of our website.



Main interface (when not input anything)



When input something

Tile component:

We generate tiles by creating a function in Board.js and set its size and border in Board.css.

Finally, we adjust our project so the size of tiles is not fixed but depends on the input from the user.

Architecture:

What we had learn:

- Recap knowledge about React, Tailwind.css and other website design method
- Learn the steps used to communicate between Front-end now Back-end.
- a good and solid design could very ease code implementation.

Progress Report:

Testing:

Description and Test cases:

Model Section:

All out classes test: Each class had been tested during and after implementation with both black-box and glass-box ideas in mind. Some classes(many reported in last reported) had to be retested due to updates and changes.

To veridate said statement, the following section will include a brief testing example which will include some test cases. Tokenizer, DecodedGene, Statement and Expression class had to be retested. For example:

Tokenizer: check if geneCode(txt file) is tokenized properly plus check if the tokenizer splits special characters out of false connected tokens.

Statement, and Expression parser: check if Parser works properly or not. But this test was limited to one statement for the test method.(usually, one statement comes from one line of gene).

DecodedGene: was checked like the Statement parser but for a whole geneCode. Meaning the black-box test case focuses on the final statement list. For testing in the glass-box idea, I use debugging tools and follow the code while it works and check if the code works as intended. Also, check if the returned GP's action "makes sense" or not. for example: if the test case starts off with a Virus assigned closely to an Antibody the GPAction returned from DecodedGene should be shot in the opponent direction.

GPsStorage: focus on testing its game's critical methods: virus, antibody, nearby direction. and new GamePiece addition into the storage both valid and invalid(attempt to adding GamePiece in coor which already existed gamepiece). Also check adding or relocating GamePiece into invalid coor(say m x n = 8 x 8, some of invalid coor are -1,0 and 8,9) [side note: even though m and n were defined the max coor will be m-1,n-1. due to lowest coor is 0,0]

GPsManager: focus on testing its ability to enforce action requested from a GamePiece. Which includes virus's hp gain from attacking antibody , Then check if Game data got updated or not (if it needs). Say, if an antibody successfully kills a virus Player should gain a certain amount of credit.

GameBoard testing: currently, GameBoard tests consist of limited to heavy staged scenarios to check if each GamePiece works properly which include its movement, and its attack. Speed and pause features are tested to some extent.

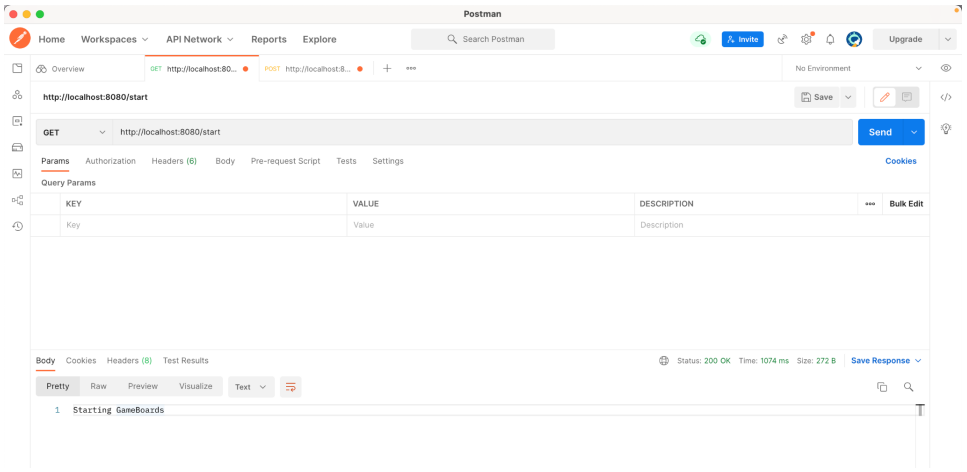
Testing:Description and Test cases::

Controller Section:

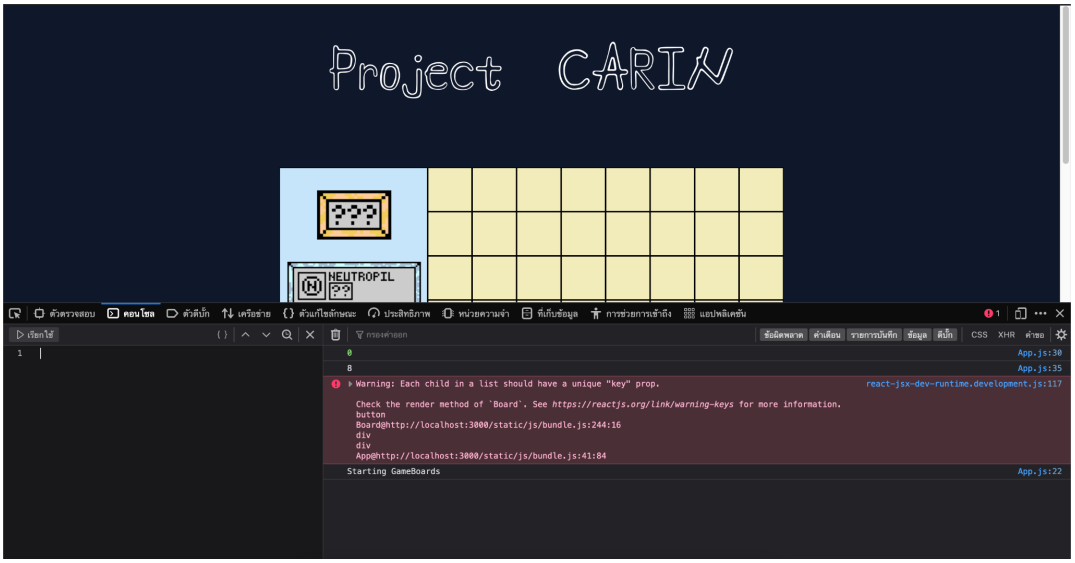
used postman to check the API for both GET and POST.

```java
@GetMapping("/start")
public String startGame(){
    GameBoard gb = GameBoard.getInstance();
    Thread t0 = new Thread(gb);
    t0.start();
    return "Starting GameBoards";
}
```

GET(“/start”)



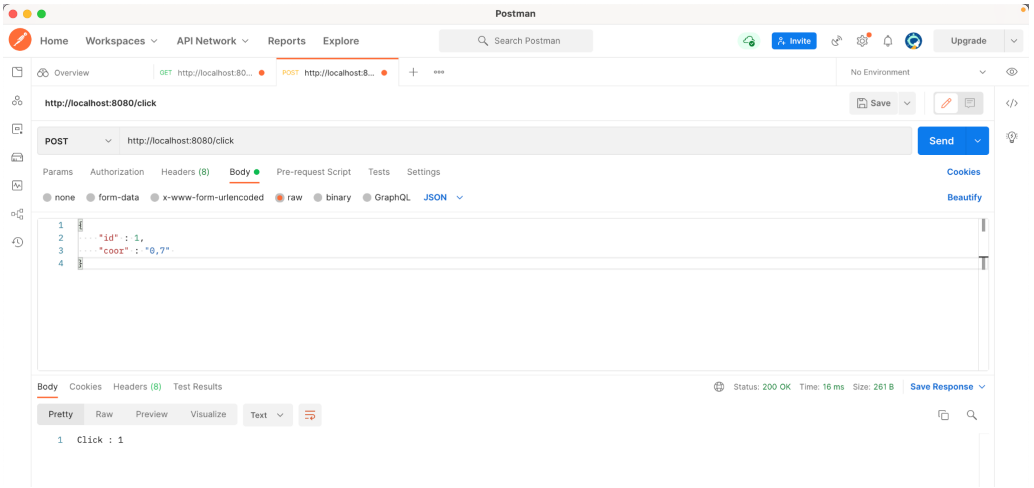
Postman



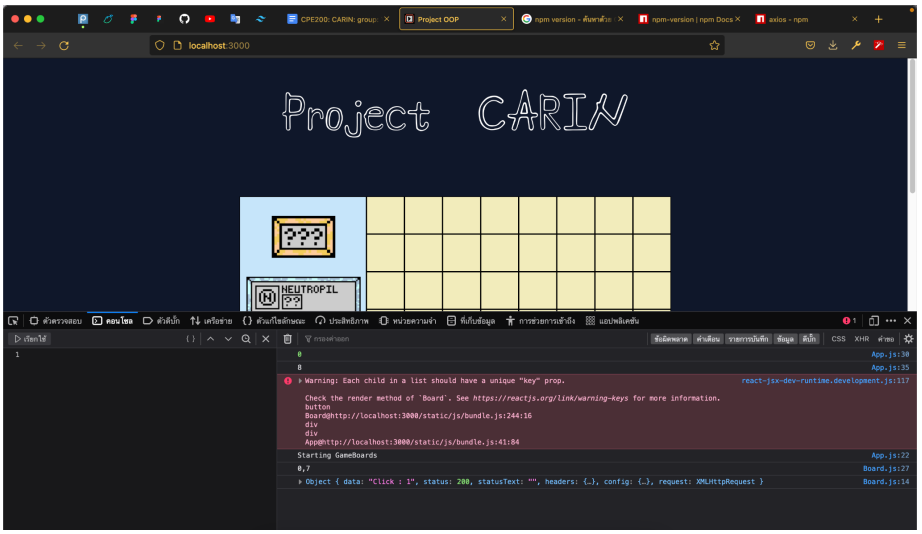
Web page(console)

```java
@PostMapping("/click")
public String click(@RequestBody ModelRequest request){
    int i = 1;
    boolean isFirst = true;
    if(click_count == 1){
        isFirst = true;
        clickCoor1 = request.getCoor().split(",");
        coor1 = new Coor(Integer.parseInt(clickCoor1[0]),
        Integer.parseInt(clickCoor1[1]));
        click_count++;
    }else if(click_count == 2){
        isFirst = false;
        clickCoor2 = request.getCoor().split(",");
        coor2 = new Coor(Integer.parseInt(clickCoor2[0]),
        Integer.parseInt(clickCoor2[1]));
        click_count = 1;
        // gbdata.checkInput2Coor(coor1, coor2);
    }else{
        click_count = 1;
    }
    if(!isFirst) i = 2;
    System.out.println(request.getCoor());
    return "Click : " + i;
}
```

GET(“/click”)



Postman



Web page(console)

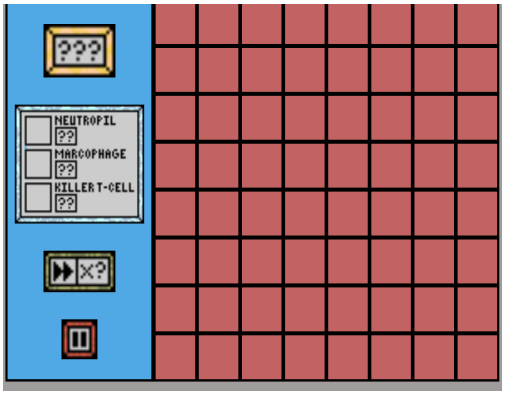
Testing:Description and Test cases::

View Section:

At first, when we successfully created a board with tiles and a side menu.
We found:

- When zooming in and out of our website, component placement looks very weird because in the first version we place our components in a grid placement and fixed position.

- Imported picture components have a lower resolution because the original picture sizes are too small for our project. So, when we change their sizes it makes them look blurred.



Credit, Shop, Speed and Pause buttons look very blurred

- Some background areas don't seem to have any colour. So, it a white blank area.

Testing:

What we had learn:

- Learn how to use a postman.
- Learned that non-public fetch APIs require permission.
- Learned how to fix various bugs that occurred throughout the work.
- Use resolution of pictures that match our work.
- Some of the bugs encountered can be detected before hitting the “run” button.
- Some code is hard to test by itself. Rather than trying to redesign to make testing easier, testers should create an ‘adapter’ to produce an output that testers can test with ease.

Progress Report:

Work plan:

Members Role:

Members' roles hadn’t changed since the 1st first report (or what was illustrated in the 2nd report).

Changes:

No major members' responsibility field had been changed. The only thing which is considered as a change is the lower frequency of was and up-coming code commission from Model section.

What we had learn:

- Successfully following a pattern can be quite difficult.
- Avoidance of turning oneself into a mule is more time efficient (in terms of project procession) and it is better for said ‘mule’ mental health.

Progress Report:

Problems:

General: At the current stage, our group is **likely** to finish the game on time. Because there are quite a lot of classes, cases and games to be tested. In addition, there are features yet implemented like: requesting spawning from player through front-end and zoom in/out.

model section: as you might notice, classes (specially for GameBoard) should be put through more tests.

Control section: Most of the problems lie in the transmission of data between the front-end and the back-end

View section: Blurred picture, component position in website is not right and unwanted blank area.