Sistemes i Tecnologies Web

Gotta Catch 'Em All: Part 2 - Frontend

Contents

1 Objectives

The main objectives of this exercise are:

- Reinforce the Vue concepts seen in the lectures.
- Use the Vue library to develop the client side and communicate with the server side using HTTP requests.

2 Back-end server

You can choose to use the NodeJS back-end implemented by your group in the first part of the project as your server, or to use the one that we provide for you on `https://pokemon-backend-62520a453d3a.herokuapp.com`. Take into account that this server is shared between everyone that uses it, so `/initial_info` can return information saved by other groups, or the endpoints can return incoherent responses if multiple groups are calling them at the same time.

3 Files structure

The provided Vue project scaffold has been created using Vite; and then some skeleton files have been added. We list below the folders/files worth mentioning.

- **package.json**: This file contains the list of npm dependencies the front-end needs. Before starting up our front-end using command `npm run dev`, we need to install those dependencies with `npm install`.
- **/src folder**: This folder contains all the files we must interact with.
  - **App.vue**: The 'entrypoint' Vue component of the application. On subsequent sections we will see the functionalities it must perform and how to implement it.
  - **/components folder**: Inside this folder there will be all the Vue components that we must implement (along with App.vue).
  - **/base_components folder**: Inside this folder there will all the Vue components that we must **not** modify. They will be used by App.vue and by the components in the */components* folder; listening to their events and passing them props.
  - **globals.js**: This file contains some global variables that we will use when we implement some Vue components. Later in section 5 we will expand on each variable that this file contains.
  - **base_globals.js**: This file contains some global variables used by the base components. We will **not** interact with this file.

4 Functionality

In this exercise we will develop a front-end application that will use the Node back-end implemented in the first part of the project.

In the first part of the project we implemented the NodeJS back-end that this front-end will use. If we remember, the back-end only stored some initial information to prevent losing the 'progress made' in case we reloaded the page. This means the front-end is the one that must keep track in real time of the position of the player in the map, his facing direction, and the list of pokemons seen or/and owned (the Pokedex).

To better distribute the tasks the front-end must perform, we have divided the application into components. Apart from the *App* component; that will be the 'entrypoint' of the application; we have separated the components of this project into two groups:

- **Base components**: These are the components already implemented that perform low level manipulation of images, animations, and movement/actions buttons management. We must not modify these components.
- **Middleware components**: Each one of these components encapsulate one base component. They transform the 'raw' information (buttons pressed/released, variables coming from another components, etc.) coming from *App* to convert it into something useful for their base components to do its job. They also manage events emitted by the base components, and, in case it's needed, emit themselves events towards *App*. These are the components that we must implement, along with the root component, *App*.

To better understand the components that we must implement/interact with, we list below each one of them, along with a description of its purpose:

- **ControlsFrame**: We must only listen to it's events, it's already implemented. It will provide us with the information about the pressed/released buttons.
- **App**: It will be the one to display `ControlsFrame` and the middleware components on its template tag. Once mounted, it must perform a call to the `/initial_info` endpoint to retrieve the game's initial information. It must act as intermediary between the base-component `ControlsFrame` and the middleware components; and between the middleware components themselves. It must listen to its events, store any variable needed, and pass props between them.
- **MapMiddleware**: This component will be the middleware of the base-component *Map*, which is the one that displays and moves the map of the game. Apart from acting as a middleware, it will also perform the requests `/enter_grass` and `/leave_grass` to the back-end.
- **PlayerMiddleware**: This component will be the middleware of the base-component *Player*, which is the one responsible to display the player sprite on screen, and to change its facing direction ('north', 'south', 'west' or 'east') and movement type ('walking' or 'standing').
- **WildEncounterMiddleware**: This component will be the middleware of the base-component *WildEncounter*, which is the one that displays the wild encounter screen. Apart from acting as a middleware, it will also perform the request `/capture` to the back-end.
- **MenuMiddleware**: This component will be the middleware of the base-component *Menu*, which is the one that displays the game's menu when we press the 'MENU' button. Apart from acting as middleware, it will also perform the request `/save` to the back-end.
- **PokedexMiddleware**: This component will be the middleware of the base-component *Pokedex*, which is the one that displays the Pokedex screen, where we can see the list of seen/owned Pokemons.

5 globals.js file

Before starting to detail the list of components to implement, let's take a look at the variables we are going to interact with inside */src/globals.js* file.

- **blocks**: This 2D array contains the block type (positive integer) that must be placed on each coordinate of the map. In this case, the map that we are going to display has 23 columns and 38 rows, this means the **blocks** array has 38 elements, each element itself being an array of 23 elements.
- **isBlockWalkable**: We identify each block type with a positive integer, as we have seen on variable **blocks**. This variable is a dictionary with key equal to the block type, and value equal to a boolean indicating if the block type is walkable through (`true`) or not (`false`).
- **pokemonNames**: This dictionary contains a key equal to a Pokemon id, and a value equal to the name of the Pokemon.

6 Components - 8.5 points

We list below the components to be implemented (except `ControlsFrame`, which we must only listen to its events), listing for each one the functionalities to implement, and, if required, the **mandatory** *components* to use on its template tag, *mounted* [1] callbacks to overwrite, *props* [2] to pass to its base component, *events* [3] to listen to, and endpoint calls to perform (remember that in Javascript, the fetch [4] function allow us to perform HTTP requests). You are allowed to use the Vue *methods*, *computed*, *watch* attributes, or any other Vue functionality you see fit.

6.1 App

- Child components:
  - **ControlsFrame**: Component that will inform us about the buttons being pressed or released at any time. It must not be implemented, just used.
  - **MapMiddleware**: Component that must manage the base-component `Map`.
  - **PlayerMiddleware**: Component that must manage the base-component `Player`.
  - **WildEncounterMiddleware**: Component that must manage the base-component `WildEncounter`.
  - **MenuMiddleware**: Component that must manage the base-component `Menu`.
  - **PokedexMiddleware**: Component that must manage the base-component `Pokedex`.
- Mounted callback: Once mounted, we must perform a call to the `/initial_info` endpoint to retrieve the game's initial information (see *'Endpoint calls'* section below).
- Endpoint calls:
  - `/initial_info`: Once the component is mounted, we must perform a call to the `/initial_info` endpoint. The method type must be GET, and we must not send any information. The returned dictionary attributes will be used by some components:
    * ***'x'*** and ***'y'***: `MapMiddleware` must use them to set the initial values of the props `playerX` and `playerY` passed to `Map` base-component.
    * ***'direction'***: `PlayerMiddleware` must use it to set the initial value of the prop `facingDirection` passed to `Player` base-component.
    * ***'pokedex'***: This dictionary must be set as a variable in some place in order to be used as initial value by some components and later be updated once we sight/capture new Pokemons. For simplicity, from now on we will call this variable `pokemonRegister`. The components that will use this variable will be:
      · `WildEncounterMiddleware`: In order to determine the prop `isPokemonOwned` passed to `WildEncounter` base-component.
      · `PokedexMiddleware`: In order to determine the props `numberOfOwnedPokemons`, `numberOfSeenPokemons` and `pokemonsInfo` passed to `Pokedex` base-component.
      · `MenuMiddleware`: In order to send the Pokedex information to the `/save` endpoint.

6.2 ControlsFrame

This component is not to be implemented. We must only listen to its events.

- Emits:
  - **onMovementUp**, **onMovementDown**, **onMovementLeft** and **onMovementRight**: This four events are emitted every time the corresponding arrow buttons are pressed or released. The event has one argument, that it's a boolean, indicating if the button has been pressed (`true`) or has been released (`false`).
  - **onButtonA**, **onButtonB** and **onButtonMenu**: This three events are emitted every time the corresponding action buttons are pressed or released. The event has one argument, that it's a boolean, indicating if the button has been pressed (`true`) or has been released (`false`).

6.3 MapMiddleware

- Child components:
  - **Map.vue**: We must display the `Map` base-component in the template tag. We must provide him with the props `playerX` and `playerY`; and we must listen to the events presented below.
- Events to listen from:
  - **onStartedMoving**: This event will be triggered by `Map` base-component every time the map starts moving from one 'x' and 'y' position to another.
  - **onFinishedMoving**: This event will be triggered by `Map` base-component every time the map finished moving from one 'x' and 'y' position to another.

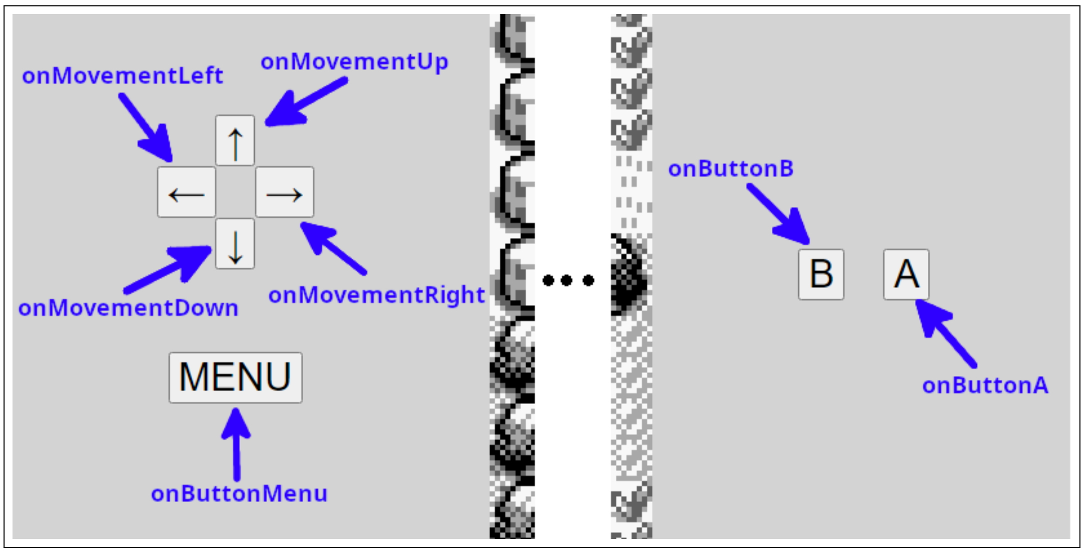

Figure 1: Event corresponding with each button.

- **onEnteredGrass**: This event will be triggered by **`Map`** base-component every time the player is entering a patch of grass. When it triggers, we must perform a call to the **`/enter_grass`** endpoint (see *'Endpoint calls'* section below).
- **onExitedGrass**: This event will be triggered by **`Map`** base-component every time the player is exiting a patch of grass. We must perform a call to the **`/leave_grass`** endpoint (see *'Endpoint calls'* section below).

- **Endpoint calls**:
  - **`/enter_grass`**: When the **`onEnteredGrass`** event triggers, we must perform a call to the **`/enter_grass`** endpoint do determine if a wild encounter must occur. The method type must be GET, and we must not send any information. In case the response status code is 400, we must enter a wild encounter. This means, keeping track though a variable or another Vue mechanism that we have entered a wild encounter. Keep in mind that various components need to know if we are inside a wild encounter, like **`WildEncounterMiddleware`** to show or hide its base-component **`WildEncounter`**; or this component to allow or disable the modification of its props **`playerX`** and **`playerY`** (we explain below the modification restrictions this two props have). In case the response status code is different than 400, we must do nothing.
  - **`/leave_grass`**: When the **`onExitedGrass`** event triggers, we must perform a call to the **`/leave_grass`** endpoint. The method type must be GET, and we must not send any information. We must ignore the response, the server will be the one to make sure the previously called **`/enter_grass`** endpoint returns 400 (in case it's been less than 4 seconds since it was called).

- **Props to pass to base-component `Map`**:
  - **playerX** and **playerY** (***number***): This two numerical props must indicate the 'x' and 'y' position that the player must move to. Its initial values must be the ones retrieved from the **`/initial_info`** call performed by the **`App`** component. To calculate this props, we must know the pressed/released movement buttons of base-component **`ControlsFrame`**. Given we can only click one movement button at a time, we just need to check if any movement button is being pressed, and determine the new **`playerX`** or **`playerY`** value. But there are some situations where we must not update the values of the props:
    - ∗ We must not change this props if the map is moving. We can know if the map is moving by listening to the **`Map`**'s base-component events **`onStartedMoving`** and **`onFinishedMoving`**. By default the map is not moving. Then, when **`onStartedMoving`** event is emitted, it means that the map has stared moving, and it will be moving until the **`onFinishedMoving`** event is emitted.
    - ∗ We must not change this props if we are inside a wild encounter. We enter a wild encounter once our call to **`/enter_grass`** endpoint presented above responds with an status code of 400. On the other hand, **`WildEncounterMiddleware`** will be the component to know if we exit a wild encounter, that will happen when either we capture the Pokemon or we run away from the wild encounter.
    - ∗ We must not change this props if the menu is open. **`MenuMiddleware`** will be the component to know if the menu is open.

    * We must not change this props if the Pokedex is open. Same as before, **MenuMiddleware** will be the component to know when to open the Pokedex given it will control the selector position inside the menu. On the other hand, **PokedexMiddleware** will be the component to know when to close the Pokedex given it knows if a button has been pressed/released through the **ControlsFrame** component. **PokedexMiddleware** uses this information to show/hide the base-component **Pokedex**.
    * Once we calculate the new 'x' or 'y' position, we must not change this props if the calculated new 'x' or 'y' position correspond with a non-walkable block. We can do that using the global variables **blocks** (to get the block type of the destination 'x' and 'y' position) and **isBlockWalkable** (to check if the block type is walkable or not). For example, on Figure 2, the player is located on `x=12,y=6` and we are pressing the movement down button. The block below him its `x=12,y=7`, and if we check its block type in the global variable **blocks**, we will see that `blocks[7][12]` is equal to `5`. Now, we must check if that block type is walkable using global variable **isBlockWalkable**. We will see that it's `false`, and thus we must not update **playerY** with the new calculated value.

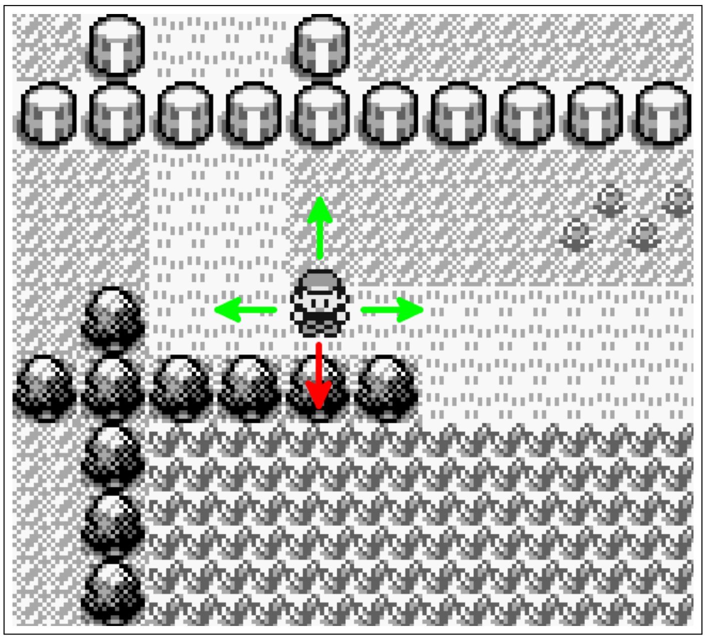

Figure 2: We cannot move down due to 'bush' block type being non walkable.

6.4 PlayerMiddleware

- **Child components**:
  - **Player.vue**: We must display the *Player* base-component in the template tag. We must provide him with the props `facingDirection` and `movementType`.
- **Props to pass to base-component `Player`**:
  - **facingDirection (*'north'* | *'south'* | *'west'* | *'east'*)**: This string prop must indicate the direction the player is moving towards. Its initial value must be the 'direction' attribute from the response dictionary of the `/initial_info` call performed by the **App** component. We must know the pressed/released movement buttons from **ControlsFrame** base-component in order to determine the prop value at any time. We must not update this prop though, in case:
    * The map is moving (remember we have seen that in **MapMiddleware**).
    * We are inside the game's menu.
    * We are inside a wild encounter.
    * We are inside the Pokedex.
  - **movementType (*'walking'* | *'standing'*)**: This string prop must indicate if the player is 'walking' or it is 'standing'. We must know the pressed/released movement buttons from **ControlsFrame** base-component to determine the prop value. If any of the movement buttons is being pressed, the prop value must be 'walking', and 'standing' otherwise. But there are some exceptions:

* If the map is moving, the prop value must always be 'walking', even if none of the movement buttons are being pressed.
* If we are inside the game's menu, the prop value must always be 'standing', even if the movement buttons are being pressed.
* When we exit a wild encounter, we must make sure the prop value is 'standing'. Given we could be 'walking' when we entered the wild encounter, could happen that we are still 'walking' when we exit, even if we are not pressing any movement button. Make sure the value is 'standing' once the base-component `WildEncounter` is hid, and we go back to seeing the map and the player.

6.5 WildEncounterMiddleware

- **Child components**:
  - **WildEncounter.vue**: We must display the `WildEncounter` base-component in the template tag. We must provide him with the props `state`, `selectorPos`, `pokemonId` and `isPokemonOwned`. We must only display the `WildEncounter` base-component in case we are inside a wild encounter. Check *'Endpoint calls'* in `MapMiddleware` section to see when we enter a wild encounter, and Figure 3 to check when we exit the wild encounter.
- **Props to pass to base-component `WildEncounter`**:
  - **state (*'START'* | *'MAIN'* | *'RUNNING_AWAY'* | *'CATCHING'* | *'CAUGHT'*)**: This string prop must indicate the current state of the wild encounter. Its starting value must be 'START', and then it must follow the state diagram presented in Figure 3.

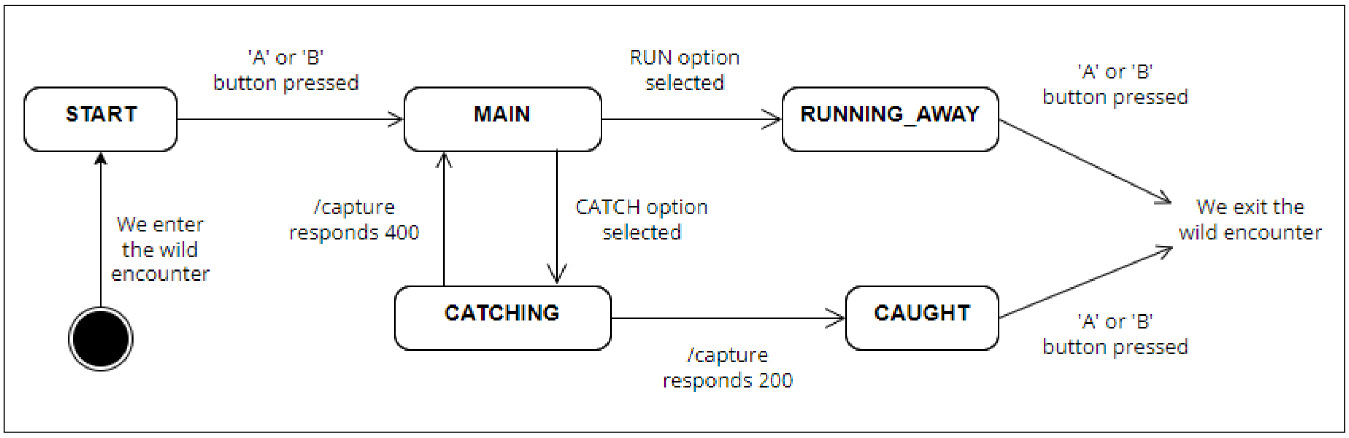
'A' or 'B' button pressed
RUN option selected
'A' or 'B' button pressed
START
MAIN
RUNNING_AWAY
We enter the wild encounter
/capture responds 400
CATCH option selected
We exit the wild encounter
CATCHING
CAUGHT
/capture responds 200
'A' or 'B' button pressed

Figure 3: Wild encounter state diagram.

  - **selectorPos (*number*)**: This positive integer prop must indicate the selector position in the wild encounter inner menu (Figure 4). Its starting value must be 0, and then it must increase or decrease depending on the pressed/released movement up and down buttons from `ControlsFrame`. In case the arrow up button is pressed while being 0; or the arrow down button is pressed while being in the last option of the list; it must circle around. We must **only** modify this prop in case we are inside a wild encounter.

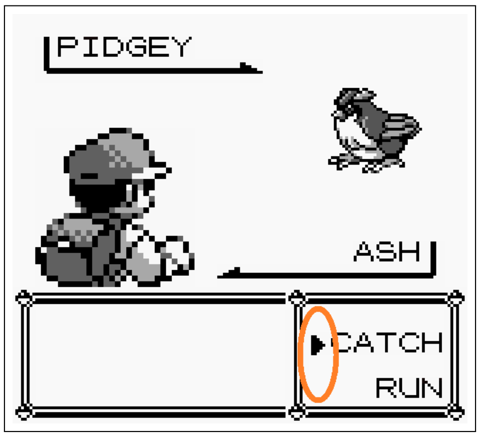

Figure 4: Selector in the wild encounter menu.

- **pokemonId (*number*)**: This position integer prop must indicate the Pokemon id of the Pokemon we must encounter. We must calculate this prop when the component is mounted (check 'Mounted callback' section below).
- **isPokemonOwned (*boolean*)**: This boolean prop must indicate if we own the Pokemon we encounter. Once we know the `pokemonId` value, we need to check if we already own it (remember the `pokemonRegister` variable presented before). On Figure 5 we can see the icon it should be displayed if we set `isPokemonOwned` prop to `true`.

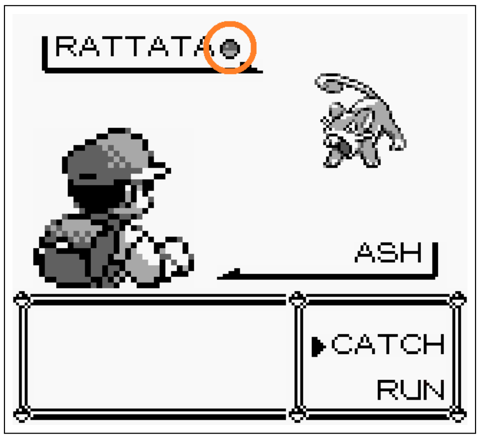

Figure 5: Indication that we already own the Pokemon.

- **Mounted callback**:
  - Once mounted, we must perform the following steps:
    - Compute a random value between 0 and 9.
    - If the value is below 5, `pokemonId` prop must be the number 16 (Pokemon Pidgey). Otherwise, it must be the number 19 (Pokemon Rattata).
    - In case it's the first time we have seen the Pokemon, we must update the `pokemonRegister` variable.
- **Endpoint calls**:
  - `/capture`: Once we select the capture option inside a wild encounter, we must perform a call to the `/capture` endpoint. The method type must be GET, and we must not send any information. In case the response status code is 200, we must update the `pokemonRegister` variable with our newly captured pokemon id; and we must change the `status` prop to 'CAUGHT' (as seen in Figure 3).

6.6 MenuMiddleware

- **Child components**:

- **Menu.vue**: We must display the *Menu* base-component in the template tag. We must provide him with the props **`state`** and **`selectorPos`**. We must only display the **`Menu`** base-component in case we are inside the menu (same as before, keeping track though a variable or another Vue mechanism that we have entered the menu). To determine when to open and close the menu, **`MenuMiddleware`** must know when the MENU button has been pressed/released from **`ControlsFrame`** base-component. Every time it's pressed, we must open or close the menu (depending if it was closed or opened respectively); but there are a few exceptions:
  - We must not open/close the menu if we are inside a wild encounter.
  - We must not open/close the menu if we are saving the game.
  - We must not open/close the menu if we are inside the Pokedex.

- **Props to pass to base-component `Menu`**:
  - **state (*'START'* | *'SAVING'* | *'SAVED'*)**: This string props indicates the current state of the menu. Its starting value must be 'START', and then it must follow the state diagram presented in Figure 6. In case we select to save the game, we must perform a call to the **`/save`** endpoint (see *'Endpoint calls'* below).

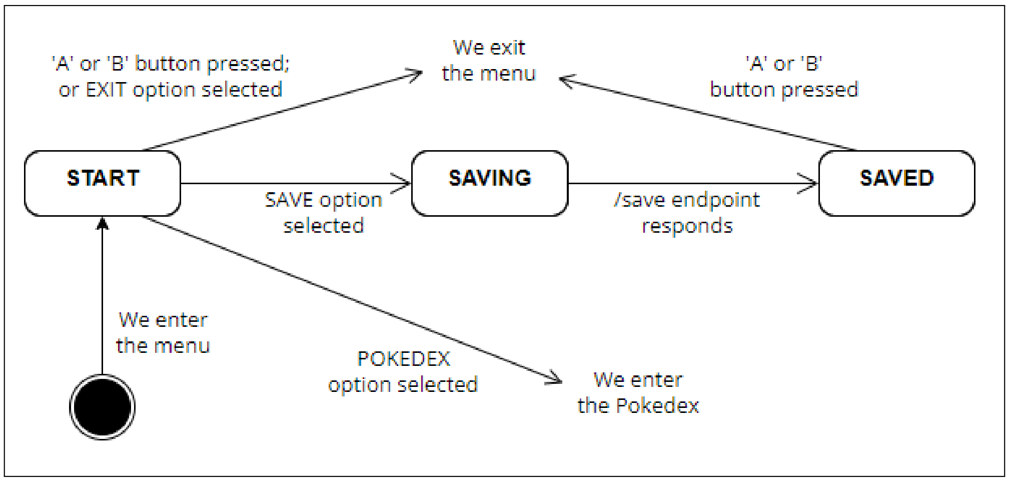

Figure 6: Menu state diagram.

  - **selectorPos (*number*)**: This positive integer prop indicates the selector position in the menu (Figure 7). Its starting value must be 0, and then it must increase or decrease depending on the pressed/released up and down movement buttons from **`ControlsFrame`**. In case the arrow up button is pressed while being 0; or the arrow down button is pressed while being in the last option of the list; it must circle around. We must only modify this prop in case:
    - The menu is open.
    - We are not saving the game.
    - We are not inside the Pokedex.

- **Endpoint calls**:
  - **`/save`**: When we select the SAVE option in the menu, we must perform a call to the **`/save`** endpoint. The method type must be POST, and we must send a dictionary in the request body with the following attributes:
    - ***'x'*** and ***'y'***: This values must be the same ones as the **`playerX`** and **`playerY`** props of **`Map`** base-component.
    - ***'direction'***: This value must be the same one as the **`facingDirection`** prop of **`Player`** base-component.
    - ***'pokedex'***: This value must be the same one as the **`pokemonRegister`** presented above.

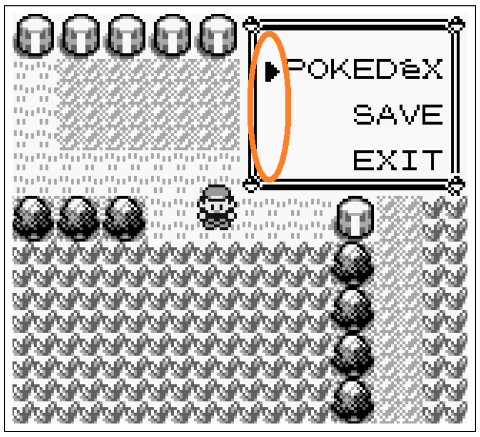

Figure 7: Selector in the game's menu.

6.7 PokedexMiddleware

- **Child components**:
  - **Pokedex.vue**: We must display the *Pokedex* base-component in the template tag. We must provide him with the props `numberOfOwnedPokemons`, `numberOfSeenPokemons` and `pokemonsInfo`. We must only display the `Pokedex` base-component in case we are inside the Pokedex (same as before, keeping track though a variable or another Vue mechanism that we have entered the Pokedex). `MenuMiddleware` will be the one to know if we must enter the Pokedex in case we have selected the POKEDEX option in the menu. On the other hand, this component must know the pressed/released action buttons from `ControlsFrame`; because, in case the B option button is pressed while the Pokedex is open, we must close it.
- **Props to pass to base-component `Pokedex`**:
  - **numberOfSeenPokemons (*number*)**: This positive integer prop must indicate the number of pokemons that we have seen (owned Pokemons are also Pokemons that we have seen). We can calculate it through the `pokemonRegister` variable presented before.
  - **numberOfOwnedPokemons (*number*)**: This positive integer prop must indicate the number of pokemons that we own. We can calculate it through the `pokemonRegister` variable presented before.
  - **pokemonsInfo (*array*)**: This prop must be an array of dictionaries. Each dictionary must represent a Pokemon present in the `pokemonRegister` variable (seen or owned). Each dictionary must contain the following keys:
    - **id**: The id of the Pokemon.
    - **owned**: A boolean indicating if we own the Pokemon or not.
    - **name**: An string indicating the name of the Pokemon. Given a Pokemon id, we can obtain its name using the global variable `pokemonNames`.

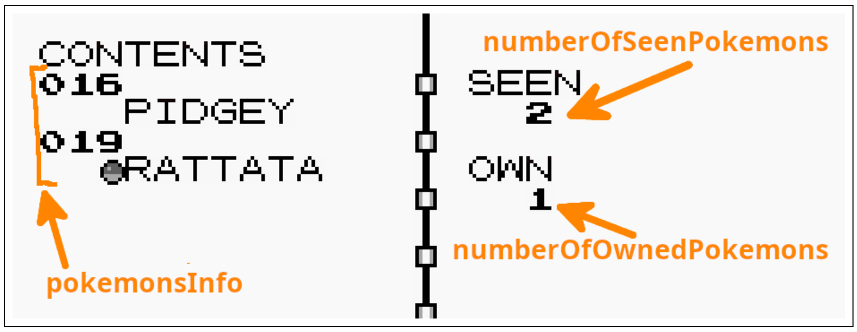

Figure 8: How each prop is displayed in the Pokedex.

6.8 Testing the movement

Below is a list of manual tests to ensure that you have correctly implemented the movement of the map and the management of the player's animations:

1. After loading the map and player correctly, press and quickly release (i.e., click) one of the movement buttons. The map should move one block, and during that time, the player should change to the corresponding 'walking' animation, reverting back once the map finishes moving.

2. Next, press and hold any movement button. The map should start moving continuously until you release the button. During that time, the player's sprite should display the corresponding 'walking' animation.

3. Click the 'menu button' to open the menu. Then, click any of the movement buttons. The player should remain stationary, and the map should not move.

4. Select the POKEDEX option on the menu. Once inside the Pokedex, click any of the movement buttons. Click the B button to return. The menu should remain open, with the cursor on the POKEDEX option, and the player should be in the same position as before.

5. Enter a patch of grass and wait for the wild encounter to trigger. After entering the encounter, click any movement button. Exit the wild encounter afterward. The player should be standing in the same position as before entering the wild encounter.

6. Finally, repeatedly click any movement button. The player should move through the map in the same manner as when continuously pressing a movement button without releasing it.

7 Capturing the Unown - 1 point

Implementing all the components presented in the previous section, will make you obtain up to 8.5 points in your grade. On this section we will explain how to obtain one more point; and, in section 8, the remaining 0.5 points.

In the backend part of the project, some groups discovered the mysterious endpoint, and found out that there was a third Pokemon available to capture. To achieve it you must perform the following steps:

- Capture all the Pokemons in the grass: **Rattata** and **Pidgey**.
- Click the 'A' action button looking towards the sign at the bottom of the map, as seen in Figure 9 (The riddle said *'Once you own all the Pokemons in the grass, reading is the only path to the third'*. This is the only block of the map where the player could 'read' something).

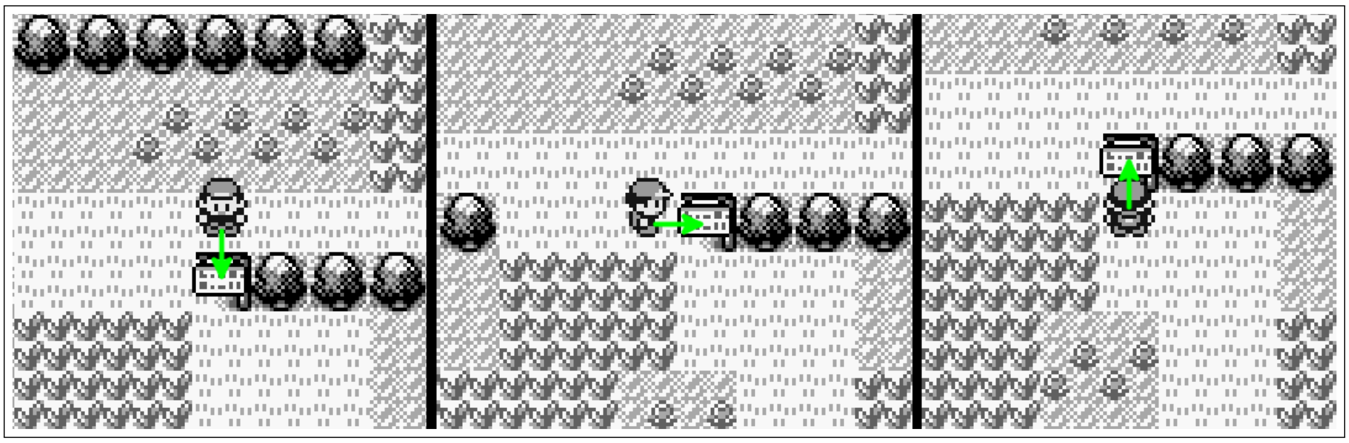
Figure 9: Each possible way the player can look towards the sign.

- This will provoke the frontend to perform a call to a GET endpoint called **`/my_status_code_is_unknown`**. You will be able to see it through your browser's *Console* or *Network* tabs.

- Now you know the mysterious endpoint's name and method type. To know what you must respond, let's take a look at the endpoint name. My status code is unknown. If you have played Pokemon before (specially the old generations), you could know that there is an 'unknown' Pokemon, called **Unown**. The Pokedex id of that Pokemon is `201`. Doesn't it seems pretty similar to a success status code?

- If we implement in our backend a GET endpoint called **`/my_status_code_is_unknown`** that returns an status code of `201`; once we click the sign in the frontend, a wild encounter will trigger, and the third Pokemon will appear, as we can see in Figure 10.

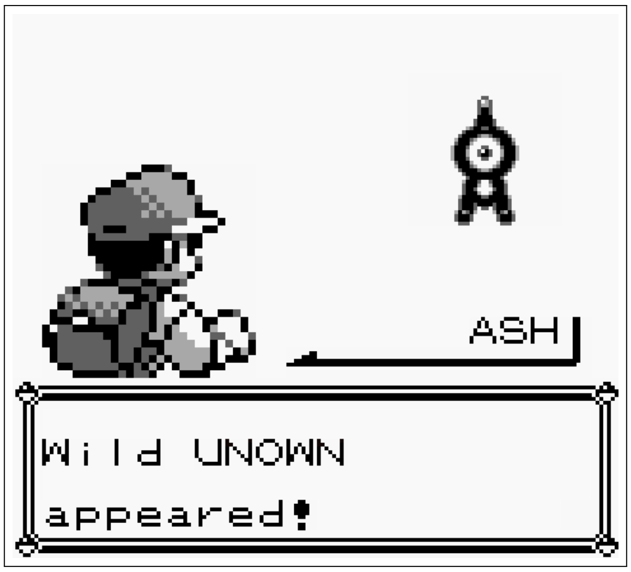

Figure 10: If we have correctly implemented the mysterious endpoint, an **Unown** will appear once clicking the sign.

.

You must implement this functionality. You can write code on any 'middleware' component and in the *App* component, but it is **prohibited** to modify any base-component. Here are a few advices before you start:

- At some point we will have to know if the block the player is looking towards (using global variable **`blocks`**) is the sign. The sign has a block type of `10`.

- At some point we will have to check if we have pressed the 'A' action button. **`ControlsFrame`** provides you with the information about buttons being pressed/released.

- Once **`/my_status_code_is_unknown`** answers and we check its status code is `201`, we must start a wild encounter where an **Unown** must appear. Until now, in the mounted callback of **`WildEncounterMiddleware`**, we 'randomly' computed the **`pokemonId`** prop to pass to **`WildEncounter`** base-component. We will have to modify that functionality to, in case the wild encounter comes from the sign instead of from the grass, display an **Unown** instead of a **Rattata** or a **Pidgey**.

8 Grass visual effect - 0.5 points

To obtain the remaining 0.5 points, we must implement the following functionality. You might have noticed that, in the game provided in the backend part of the project (from now on, 'the provided game'), once the player enters a patch of grass, his body seems to be halfway covered by the grass, having only a part of his head above it (Figure 11 - Left).
But in the project you are implementing, the player always appears in front of the grass (Figure 11 - Right). **You must implement this visual effect** in the same way the provided game does. In Figure 12 you can see how the grass starts to cover the player once he starts entering it. As we can see, the superior part of the player's sprite is always visible, and the inferior one is always underneath the grass sprite.
You are **allowed to write code on any component of the project**, including the base components. Here are a few advices before you start:

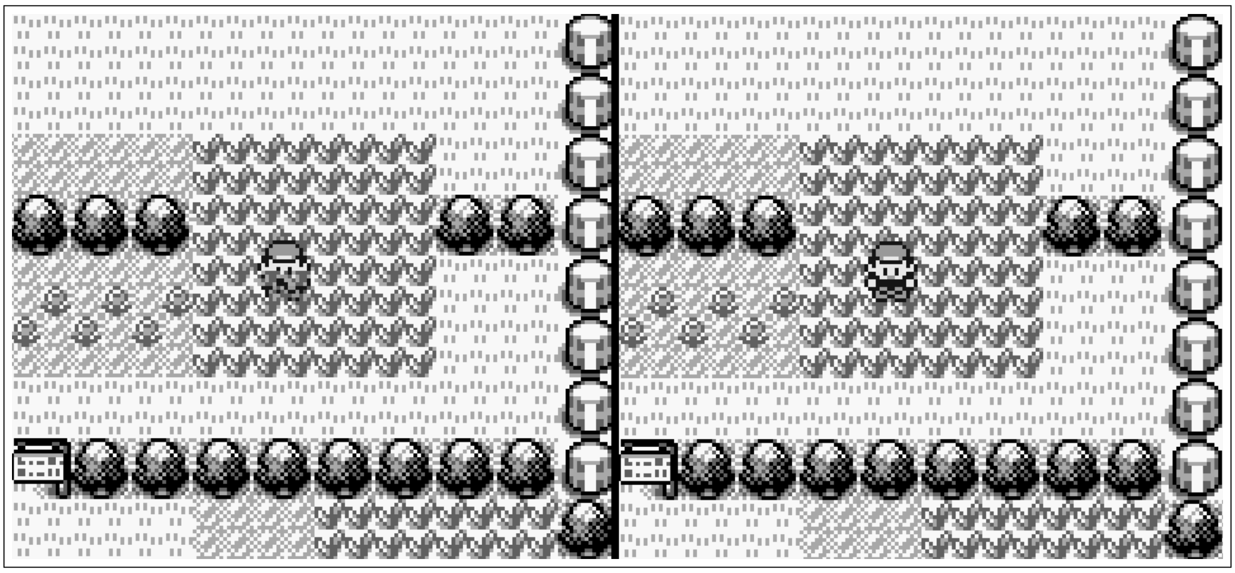
Figure 11: Player inside the grass in the provided game (left) vs in your project (right).

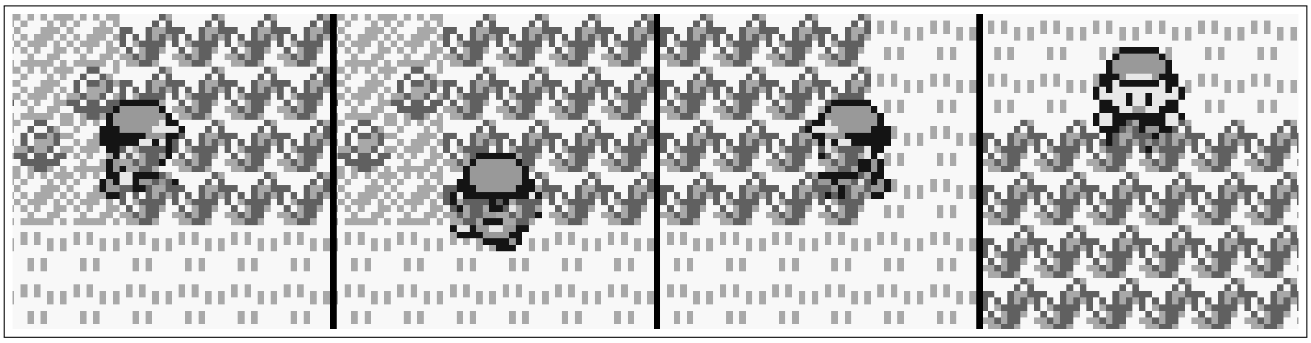
Figure 12: Entering grass from west, south, east and north respectively.

- We recommend you to check how **`Player`** base-component displays the player sprite on screen. The player sprite it's divided into two images (with id **`playerHead`** and **`playerBody`**), the superior part of the player, and the inferior one. This can help you achieve this functionality, but if you manage to do it without using this splitted images, it is also correct.
- Given the fact that the player's sprite inferior part is underneath, but visible through the grass, we might find useful an image in */assets/map* folder called *high_grass_transparent.png*. It's the grass sprite, but without background.
- At some point we will have to know if the block the player is about to enter/move through/leave is a grass block (using global variable **`blocks`**). The grass has a block type of `4`.

References

[1] **Vue mounted lifecycle callback**: `https://vuejs.org/api/options-lifecycle.htmlmounted`

[2] **Vue props**: `https://vuejs.org/guide/components/props.html`

[3] **Vue events**: `https://vuejs.org/guide/components/events.html`

[4] **Fetch API**: `https://developer.mozilla.org/en-US/docs/Web/API/Fetch`$_{A}PI$