TIW8 - TP2 Application de Présentation multi-surface en React

Encadrants

- Aurélien Tabard (responsable)
- Louis Le Brun

Présentation du TP

L'objectif du TP est de mettre en place une Single Page Application (SPA) permettant de créer et contrôler des présentations. Elle sera développée principalement côté client avec React, avec un serveur Node/Express léger. Le serveur sera codé en JavaScript, le client en TypeScript.

Les points suivants seront abordés

- Composants React
- Gestion des états et flux de données
- Gestion de routes React
- Redux pour la gestion avancée des états
- Middleware pour gérer des effets de bord
- Websockets et communication temps réelle entre dispositifs
- Design responsif et adaptatif
- Reconnaissance de gestes

Ce TP s'étalera sur 4 séances et fera l'objet d'un rendu en binome et d'une note. Voir les critères d'évaluation en bas de la page.

Pensez à remplir le formulaire de rendu sur Tomuss (n'importe quand avant la date de rendu).

Quelques pointeurs vers la doc React

- Introduction à la structuration d'application React
- Components and Props
- Hooks at a Glance
- En quoi les fonctions composants sont-elles différentes des classes ?

TP2.1 Introduction à React

Nous allons repartir du TP1 pour ce projet, et le pousser dans un nouveau repo dédié au TP2 (pour les 4 séances du TP).

Structurer une application React en composants

Lire l'introduction à la structuration d'application React.

Nous allons commencer par créer un squelette d'application statique, nous rajouterons les parties dynamiques par la suite.

L'application est composée de transparents, d'outils d'édition, d'outils de navigation, et d'outils de présentations (notes, timer, ...)

Les transparents auront (à minima) deux modèles:

- titre (le transparent ne contient qu'un grand titre centré), et
- contenu qui peut avoir les éléments suivants :
    - Titre,

    - Image (url vers imgur ou autre fournisseur d'image),
    - Texte (texte libre ou liste).
    - Notes pour l'orateur
    - Visibilité (afficher/cacher le transparent)
- On peut penser à d'autre modèles: image seule, iframe intégrant une page html, slide avec flux de la webcam intégré...

Imaginez que le serveur envoie ce type de données (qui peuvent être améliorées/modifiées selon vos besoins) :

```
[
  { type: "title", title: "TIW 8", text: "", visible: true, notes: "" },
  {
    type: "content",
    title: "TP 1",
    text: "Le TP porte sur des rappels de developpement Web",
    visible: false,
    notes: "ce transparent est caché",
  },
  {
    type: "content",
    title: "TP 2",
    text: "Le TP porte sur la creation d'un outil de presentation HTML",
    visible: true,
    notes: "",
  },
  { type: "content", title: "TP 3", text: "Le TP 3", visible: true, notes: "" },
  { type: "content", title: "TP 4", text: "Le TP 4", visible: true, notes: "" },
  { type: "title", title: "Question ?", text: "", visible: true, notes: "" },
];
```

Créer des composants passifs

Nous allons créer la structure des composants correspondant à cette application, en suivant le guide de Thinking in React.

Nous allons commencer par créer des composants fonctionnels passifs, on rajoutera de l'interaction par la suite.

Créez un composant `App` principal qui contiendra un composant `Slideshow` et un composant `AppToolbar`. Nous définirons l'état de l'application dans `App` et passerons cet état comme une prop à `Slideshow` et `AppToolbar`.

```
import * as React from 'react'
import SlideShow from './components/SlideShow'
import AppToolbar from './components/AppToolbar'

const App: React.FC = () => {
    const [slides] = React.useState(VOIR PLUS HAUT)

    return (
        <div className="app">
            <AppToolbar slides={slides}  />
            <SlideShow slides={slides} />
        </div>
    )
}

export default App
```

Linter et Typage

Vous allez vous rendre compte rapidement que votre linter râle.

Plusieurs choses à faire :

1. Créez un fichier `type.d.ts` dans `client` qui définisse l'Interface `Slide`. C'est à dire le type d'un Slide (voir le tableau ci-dessus). Vous pouvez vous référer à cet exemple de Todo App (section 'Creating a Type Declaration File').

2. Si vous utilisez Prettier, lier proprement Prettier et Eslint qui peuvent parfois avoir des opinions différentes. Vous trouverez ici mon fichier .eslint

React Router

Pour terminer ce TP nous allons rajouter la gestion de routes, pour qu'il soit possible d'avoir un lien dédié pour chaque transparent. Mais aussi pour que l'on puisse controller la présentation depuis la toolbar. Cette dernière doit contenir deux boutons avant/arrière pour naviguer entre les transparents, et un menu pour sélectionner le transparent à afficher. Faites en sorte que la route change en fonction.

Nous allons utiliser react-router. Pour en comprendre la logique (et les différences avec d'autres outils de routing), je vous invite à lire cette page.

React router requiert d'envelopper votre application dans un composant `Router`.

En l'occurrence `HashRouter` (vous pouvez utiliser `BrowserRouter` mais cela demande une configuration côté serveur). L'idée est que charger un url de type http://monsite.net/#/3 charge le 3e transparent. Installez et importez `react-router-dom` dans votre index. Installez aussi `@types/react-router-dom` pour Typescript.

Dans votre composant `App` de base rajoutez un Switch qui s'occupera de capturer les chemins et d'afficher les bons composants. Par exemple:

```
<Switch>
  <Route
    path="/:id"
    render={() => (
        <SlideShow slides={slides} />
        <AppToolbar slides={slides} />
    )}
  />
</Switch>
```

Dans vos composants (ici SlideShow et AppToolbar), vous pouvez récupérer la route en utilisant `useParams` de `react-router`. Importez le puis à l'intérieur déclarez:

```
    const params = useParams<RouteParams>()
```

`params.id` correspondra à votre path. C'est un `string`, vous pouvez utiliser `Number(params.id)` pour le caster en entier et faire des opérations dessus.

Déploiement

Si ce n'est pas encore fait, ou que vous n'avez pas automatisé, c'est le moment de tester le déploiement sur Heroku.

TP2.2 Easy Peasy

Nous allons maintenant gérer l'état de l'application avec Easy-Peasy. Pensez à relire le cours et les ressources associées pour être au clair sur ce que vous êtes en train de faire. Historiquement nous utilisions Redux, mais Easy-Peasy permet une implémentation vraiment simplifiée et très légère.

Afin de vous faciliter le debug du TP, vous pouvez activer la création d'un Source Map dans votre webpack.config.js : `devtool: 'eval-source-map'.`

Installez le package `easy-peasy`, vous pourrez trouver plus d'informations et la documentation à cette adresse: https://easy-peasy.dev/

Création d'un store

Nous allons commencer par créer le store qui va gérer les états.

Créez la suite de dossiers `application/store` dans lequel il y aura un fichier `index.ts`. Notre application étant assez simple, nous allons utiliser un seul store. Puisque nous sommes en TypeScript, nous devons définir le type du store et les types des actions.

Le contenu du fichier pour créer le store va ressembler à ceci:

```
import { Action, action, createStore } from "easy-peasy";

interface SlideStore {
  name: string
  currentSlide: number
  nextSlide: Action<SlideStore>
  previousSlide: Action<SlideStore>
  setSlide: Action<SlideStore, SetSlideAction>
  changeSlideVisibility: Action<SlideStore, SlideVisibilityAction>
}

interface SetSlideAction {
  slideNumber: number
}

interface SlideVisibilityAction {
  isVisible: boolean
}
```

```
const store = createStore<SlideStore>({
  name: 'slidesApp',
  currentSlide: 0,
  nextSlide: action((state, payload) => {
    // TODO
  }),
  previousSlide: action((state, payload) => {
    // TODO
  }),
  setSlide: action((state, payload) => {
    // TODO
  }),
  changeSlideVisibility: action((state, payload) => {
    // TODO
  })
});

export default store;
```

Vous remarquerez que le store spécifie à travers son interface les différentes "Actions" qu'il peut appeler afin de modifier son état.

Utiliser le store

Dans votre `index.tsx` principal exposez le store pour pouvoir l'afficher via la console du navigateur. Cela permettra d'effectuer les premiers tests de easy-peasy, sans l'avoir branché à votre application React.

```
import { store } from './application/store' // verifiez que le chemin est correct

declare global {
    interface Window {
        mystore: unknown
    }
}
window.mystore = store
```

Et enveloppez votre application dans une balise :

```
<StoreProvider store={store}>`
  ...
</StoreProvider>`
```

Lien React - Redux

Maintenant on va tester que le flux d'information ce passe bien. On va rajouter un bouton à Toolbar. Quand on clique dessus, il va modifier la propriété `visible` du slide courant. Si le slide est visible il deviendra invisible et inversement.

Pour vous faciliter la vie, on ne va pas le rendre vraiment invisible mais simplement changer son opacité de 100% à 10%.

Pour faire cela nous allons devoir modifier trois fichiers

1. Le composant toolbar
2. le composant d'affichage d'un transparent
3. la slice qui gère l'état de l'application

Ajoutez d'abord un bouton à la toolbar. On va importer les éléments suivants dans la Toolbar:

```
import { useDispatch } from "react-redux";
import { AppDispatch } from "../store";
import { changeVisibilitySlide } from "../slices/slideshowSlice";
```

Lorsque l'on clique sur le bouton on va dispatcher une action :

```
  // dans votre composant on branche le dispatch au store :
  const dispatch = useDispatch<AppDispatch>()
```

```
...
// lors du click sur le bouton
onClick={() => {
    dispatch(
        changeVisibilitySlide(
            currentSlide // vient de la route dans la barre de navigateur via react-router
        )
    )
}}
```

Dans le composant transparent (`SlideView` chez moi), récupérez l'état de visibilité du slide. S'il est visible l'opacité est normale sinon à 10%. Rajoutez un div enveloppant pour gérer ça.

```
  const opacity: string = isVisible ? 'opacity-10' : 'opacity-100'
  ...
  <div className={opacity}>
```

Enfin, si ce n'est pas encore fait dans le code de votre slice, définissez `changeVisibilitySlide` pour que l'état global de votre application soit bien mis à jour.

TP2.3 Distribution d'interface multi-dispositif

Nous allons maintenant travailler à la distribution de l'application sur plusieurs dispositifs et à leur synchronisation.

Définition de nouvelles routes et des vues associées

Nous allons définir une route par situations d'usage :

- `controler` : route pour dispositif mobile qui va contrôler la présentation et afficher les notes de présentation.
- `present` : route pour le mode présentation plein écran, seule une diapositive plein écran sera affichée (pas de toolbar).
- `edit`: mode actuel permettant l'édition des transparents

Il n'existe pas de bibliothèque à l'heure actuelle pour gérer de manière simple de la distribution d'interface, nous allons donc devoir le faire "à la main".

Rajouter des `Redirect` (doc) à la racine de votre application pour faire une redirection vers une route en fonction du dispositif utilisé et de son état.

Vous pouvez utiliser `react-device-detect` (doc) pour détecter le dispositif (mobile ou non). Et la `fullscreen API` (doc) pour contrôler le plein écran.

Déployez et tester.

Créez une vue controler

Cette vue pour mobile affiche les notes de présentation associées à un transparent ainsi que les boutons suivant précédent.

Nous allons travailler sur la synchronisation entre les dispositifs ci-dessous. Pour l'instant la vue doit simplement afficher les notes correspondant au transparent courant.

Gestion "à la main" des routes des transparents

Nous allons maintenant préparer la synchronisation des dispositifs. Pour cela nous allons devoir gérer le transparent courant dans notre état (`currentSlide` dans le store). `ReactRouter` n'est pas conçu pour bien gérer le lien entre route et état. Et les routeur alternatifs (type `connected-react-router`) ont aussi des limites. Nous allons donc gérer cette partie de la route à la main.

Changer l'état à partir de la route

En écoutant l'évènement `popstate` nous pouvons êtres informé d'un changement dans l'url du navigateur. Si ce changement correspond à un changement dans le numéro de transparent à afficher, nous allons déclencher l'action `setSlide` avec le numéro de transparent approprié.

Changer la route à partir de l'état

Dans votre composant principal (là ou vous utilisez `useAppSelector`), en cas de changement de currentSlide dans le store, on change l'url du navigateur

```
const hash = "#/" + slides.currentSlide;
if (location.hash !== hash) {
  window.location.hash = hash;
  // Force scroll to top this is what browsers normally do when
```

```
  // navigating by clicking a link.
  // Without this, scroll stays wherever it was which can be quite odd.
  document.body.scrollTop = 0;
}
```

Un premier Middleware de logging

Pour comprendre la logique du Middleware suivez la documentation Redux. Faites un essai qui reprend en suivante cette courte vidéo) (pensez juste à installer `@types/redux-logger` en plus).

```
export const store = configureStore({
  reducer: slideshowReducer,
  middleware: [logger],
});
```

Nous allons maintenant créer un logger similaire "à la main" (vous pouvez faire ça dans le fichier de base de votre store). Un middleware a une signature un peu particulière. Il s'agit en fait de 3 fonctions imbriquées:

```
const myLoggerMiddleware: Middleware<Dispatch> = (store: Store) => (next) => {
  return (action: AnyAction) => {
    console.log("State Before:", store.getState());
    return next(action);
  };
};
```

- La fonction externe est le middleware lui-même, appelée par `applyMiddleware` (voir ci-dessous), elle reçoit un objet de type `Store` qui contient les fonctions {dispatch, getState} du store.
- La fonction centrale reçoit une fonction `next` comme argument, qui appellera le prochain middleware du pipeline. S'il c'est le dernier (ou l'unique), alors la fonction `store.dispatch`
- La fonction interne reçoit l'action courante en argument et sera appelée à chaque fois qu'une action est dispatchée.

Vous pouvez importer tous les types nécessaire depuis `@reduxjs/toolkit`

Notre Middleware de diffusion des actions avec des websockets

Nous allons maintenant faire communiquer plusieurs navigateurs entre eux grâce à socket.io. Pour cela nous allons rajouter un middleware dédié. Sur un navigateur, quand la slide courante sera changée, un message sera envoyé aux autres navigateurs afin qu'ils changent eux aussi leur slide courante.

Websockets côté serveur

Côté serveur, importez `socket.io` (tuto officiel) et mettez en place le callback permettant de recevoir les messages d'action provenant d'un client et de les propager à tous les autres clients. Ce guide permet de créer et tester une micro-application express utilisant socket.io en local et sur Heroku.

Le serveur ne va quasi rien faire, quand il reçoit un message d'action, il le broadcast à tous les clients connectés:

```
socket.on("action", (msg) => {
  console.log("action received", msg);
  socket.broadcast.emit("action", msg);
});
```

Websockets côté client

Passons à la création de notre propre Middleware dans lequel on importera `socket.io-client` (installez le avec yarn). Le middleware devra, dès qu'il intercepte une action (`setSlide` ou autre) la propager au serveur via un websocket par un message adéquat, avant de faire appel à `next(action)`.

```
import io from "socket.io-client";
import { store } from "./index";
import { setSlide, changeVisibilitySlide } from "../slices/slideshowSlice";
import { Middleware, Dispatch, AnyAction } from "redux";

// on se connecte au serveur
const socket = io();
```

```
export const propagateSocketMiddleware: Middleware<Dispatch> =
  () => (next) => (action: AnyAction) => {
    // Explorez la structure de l'objet action :
    console.log("propagateSocketMiddleware", action);

    // TODO traiter et propager les actions au serveur.
    // Vous pourrez utiliser
    // socket.emit('type_du_message', 'contenu du message, peut être un objet JS');

    // Après diffusion au serveur on fait suivre l'action au prochain middleware
    next(action);
  };
```

Toujours dans le middleware, configurez la socket pour qu'à la réception des messages, les actions soient dispatchées au store.

```
socket.on("action", (msg) => {
  console.log("action", msg);
  switch (
    msg.type // ajuster le msg.type pour qu'il corresponde bien à celui dédifit pour l'action votre reducer
  ) {
    case "set_slide": // <- probablement autre chose à vous de trouver
      store.dispatch(setSlide(msg.value, false));
      break;
  }
});
```

Vous remarquerez sans doute qu'au point où nous en sommes nous allons provoquer une boucle infinie d'émissions de messages. Pour éviter cela, les actions peuvent embarquer un information supplémentaire grâce la propriété `meta`. Faites en sorte que seuls les dispatchs provenant d'un clic sur un bouton ou d'une modification de l'URL provoquent la propagation d'un message via Websocket.

Comme nous utilisons ReduxToolkit et TypeScript, il faut utiliser un `prepare` callback comme décrit ici

On déploie

Branchez tout et déployez. Corriger la connexion websocket au besoin.

TP2.4 Modalité d'entrées (gestes, stylet)

Gestion de modalités d'entrée

Nous allons maintenant ajouter un espace pour faire des gestes sur son téléphone pour déclencher des actions.

Création d'un canvas sur lequel dessiner

Pour cette partie, nous prendrons exemple sur ce tutoriel W. Malone.

Dans votre composant dédié au mobile, ajoutez un élément `canvas` déclarant une Référence React:

```
<canvas className="stroke" ref={refCanvas}></canvas>
```

Vous pouvez ajouter la propriété css `touch-action: none;` à votre canvas pour bloquer le zoom par défaut dans certains navigateurs.

On déclarera une fonction `useEffect` qui va gérer le dessin du geste au fur et à mesure que les événements de pointer (touch, souris, pen/stylus) arrivent. Associer les handlers d'événements `pointerdown, pointermove et pointerup au canvas dans useEffect. Déclenchez la fonction de` redraw`.

Afin de vous faciliter la tâche, voici le code *presque* complet pour faire marcher le dessin sur le canvas.

Assurez-vous de bien faire les imports nécessaires au bon fonctionnement du code ci-dessous. Faites en sortes que l'on ne dessine que si c'est la souris ou du touch qui est utilisé.

```
var clickX = new Array();
var clickY = new Array();
var clickDrag = new Array();
var paint = false;
```

```
// Cette ligne permet d'avoir accès à notre canvas après que le composant aie été rendu. Le canvas est alors
disponible via refCanvas.current
let refCanvas = useRef(null);

function addClick(x, y, dragging) {
  clickX.push(x);
  clickY.push(y);
  clickDrag.push(dragging);
}

function redraw() {
  let context = refCanvas.current.getContext("2d");
  let width = refCanvas.current.getBoundingClientRect().width;
  let height = refCanvas.current.getBoundingClientRect().height;

  //Ceci permet d'adapter la taille du contexte de votre canvas à sa taille sur la page
  refCanvas.current.setAttribute("width", width);
  refCanvas.current.setAttribute("height", height);
  context.clearRect(0, 0, context.width, context.height); // Clears the canvas
  context.strokeStyle = "#df4b26";
  context.lineJoin = "round";
  context.lineWidth = 2;

  for (var i = 0; i < clickX.length; i++) {
    context.beginPath();
    if (clickDrag[i] && i) {
      context.moveTo(clickX[i - 1] * width, clickY[i - 1] * height);
    } else {
      context.moveTo(clickX[i] * width - 1, clickY[i] * height);
    }
    context.lineTo(clickX[i] * width, clickY[i] * height);
    context.stroke();
  }
}

function pointerDownHandler(ev) {
```

```
    console.error(
      "HEY ! ICI ON PEUT DIFFERENCIER QUEL TYPE DE POINTEUR EST UTILISE !"
    );

    let width = refCanvas.current.getBoundingClientRect().width;
    let height = refCanvas.current.getBoundingClientRect().height;
    var mouseX = (ev.pageX - refCanvas.current.offsetLeft) / width;
    var mouseY = (ev.pageY - refCanvas.current.offsetTop) / height;

    paint = true;
    addClick(mouseX, mouseY, false);
    redraw();
  }

  function pointerMoveHandler(ev) {
    if (paint) {
      let width = refCanvas.current.getBoundingClientRect().width;
      let height = refCanvas.current.getBoundingClientRect().height;
      addClick(
        (ev.pageX - refCanvas.current.offsetLeft) / width,
        (ev.pageY - refCanvas.current.offsetTop) / height,
        true
      );
      redraw();
    }
  }

  function pointerUpEvent(ev) {
    paint = false;
  }
```

Reconnaissance de gestes

Pour terminer, nous allons effectuer de la reconnaissance de geste lors d'évènements touch.

Pour ce faire nous allons utiliser le $1 recognizer vu en cours. Nous allons utiliser une version modifiée de OneDollar.js pour fonctionner avec React. Il n'y a pas de module TypeScript (ou JS) récent pour cette bibliothèque. Nous devrions donc le créer, mais pour plus de simplicité nous allons placer directement la bibliothèque dans le dossier `client/` pour qu'elle soit facilement bundlée par Webpack.

Gérer le recognizer

Le recognizer est du bon vieux JS, on va échapper la vérification des types à ce stade (je suis preneur d'une version TS de $1 recognizer si l'envie vous prenait).

Au niveau de votre composant, importer et initialiser votre le One Dollar Recognizer.

```
// Voir ici pour le détails de options https://github.com/nok/onedollar-unistroke-coffee#options
const options = {
  score: 80, // The similarity threshold to apply the callback(s)
  parts: 64, // The number of resampling points
  step: 2, // The degree of one single rotation step
  angle: 45, // The last degree of rotation
  size: 250, // The width and height of the scaling bounding box
};
const recognizer = new OneDollar(options);

// Let's "teach" two gestures to the recognizer:
recognizer.add("triangle", [
  [627, 213],
  [626, 217],
  [617, 234],
  [611, 248],
  [603, 264],
  [590, 287],
  [552, 329],
  [524, 358],
  [489, 383],
  [461, 410],
  [426, 444],
  [416, 454],
  [407, 466],
```

```
    [405, 469],
    [411, 469],
    [428, 469],
    [453, 470],
    [513, 478],
    [555, 483],
    [606, 493],
    [658, 499],
    [727, 505],
    [762, 507],
    [785, 508],
    [795, 508],
    [796, 505],
    [796, 503],
    [796, 502],
    [796, 495],
    [790, 473],
    [785, 462],
    [776, 447],
    [767, 430],
    [742, 390],
    [724, 362],
    [708, 340],
    [695, 321],
    [673, 289],
    [664, 272],
    [660, 263],
    [659, 261],
    [658, 256],
    [658, 255],
    [658, 255],
  ]);
  recognizer.add("circle", [
    [621, 225],
    [616, 225],
    [608, 225],
```

```
    [601, 225],
    [594, 227],
    [572, 235],
    [562, 241],
    [548, 251],
    [532, 270],
    [504, 314],
    [495, 340],
    [492, 363],
    [492, 385],
    [494, 422],
    [505, 447],
    [524, 470],
    [550, 492],
    [607, 523],
    [649, 531],
    [689, 531],
    [751, 523],
    [782, 510],
    [807, 495],
    [826, 470],
    [851, 420],
    [859, 393],
    [860, 366],
    [858, 339],
    [852, 311],
    [833, 272],
    [815, 248],
    [793, 229],
    [768, 214],
    [729, 198],
    [704, 191],
    [678, 189],
    [655, 188],
    [623, 188],
    [614, 188],
```

```
    [611, 188],
    [611, 188],
  ]);
```

Traitement différencié selon le type du pointerEvent.

Etendre les fonctions `pointerDownHandler`, `pointerMoveHandler`, `pointerUpHandler` pour qu'elles traite différemment les sources `touch`, `pen` et `mouse`.

Nous allons associer les gestes au `touch`. Toutefois pour débugger plus facilement, vous pouvez commencer traiter les gestes sur le pointerEvent `mouse`, et basculer sur le touch une fois que cela marche bien.

Stocker les points composants le geste dans un Array `gesturePoints`.

Dessiner le geste

Dans la fonction de dessin `redraw` vous pouvez ajouter un cas à la fin qui dessine en cas de geste (les points composant le geste sont stockés dans `gesturePoints`). Vous devrez être **vigilant à convertir vos points pour être dans le référentiel du canvas**, comme dans le code fournit ci-dessus.

```
  function redraw(){

    ...

    if (gesture) {
      context.strokeStyle = "#666";
      context.lineJoin = "round";
      context.lineWidth = 5;

      context.beginPath();
      context.moveTo(gesturePoints[0][0]*width, gesturePoints[0][1]*height);
      for(var i=1; i < gesturePoints.length; i++) {
        context.lineTo(gesturePoints[i][0]*width-1, gesturePoints[i][1]*height);
      }
```

```
      context.stroke();
    }
  }
```

Reconnaitre un geste prédéfinit

Quand le geste se termine (`pointerUpHandler`), vous pouvez lancer la reconnaissance du geste.

```
let gesture = recognizer.check(gesturePoints) as Gesture
```

`as Gesture` permettant de caster le résultat dans un type que vous pouvez rajouter à votre fichier, ou à `type.d.ts` la fonction check peut aussi renvoyer un résultat booleen ou un -1. Vérifiez donc que le cast s'est bien passé, avant de traiter `gesture`.

```
type Gesture = {
    name: string
    score: number
    recognized: boolean
    path: {
        start: any[]
        end: any[]
        centroid: any
    }
    ranking: {
        name: string
        score: number
    }[]
}
```

Inspectez l'objet gesture dans la console, et vérifiez que vous arrivez bien à reconnaitre un cercle et un triangle.

Pensez à réinitialiser `gesturePoints` une fois le geste terminé.

Apprendre de nouveaux gestes

Toujours dans `pointerUpHandler`, vous pouvez imprimer les trajectoires correspondants à des gestes.

```
console.log("[[" + gesturePoints.join("],[") + "]]");
```

Utiliser cette sortie pour ajouter deux nouveaux gestes: '>' et '<' (partant du haut vers le bas) à votre recognizer.

Associer le geste à une action

Une fois le geste exécute, s'il correspond à un de ces deux nouveaux gestes (`recognized == true`), dispatcher les actions suivant ou précédent.

Vérifier que l'action est bien distribuée sur tous les dispositifs connectés.

FIN

Vous pouvez maintenant tester, nettoyer le code, et rendre.

Rendu

À rendre pour le dimanche 14/11 à 23h59.

1. Déployez votre code sur Heroku
2. Pousser votre code sur la forge
3. Déposer les liens sur Tomuss "UE-INF2427M Technologies Web Synchrones Et Multi-Dispositifs"

- Le lien vers Heroku pointe vers le 1e transparents
- Le lien vers la forge permet de faire un clone (format suivant: https://forge.univ-lyon1.fr/xxx/tiw8-tp2.git)

Critères d'évaluation

- Fichier `README.md` décrivant le process de build en dev, en prod, et de déploiement.
- Fichier `package.json` nettoyé ne contenant que les dépendances nécessaires.
- Linter bien défini qui ne renvoie pas d'erreur (et pas d'exception partout)
- Typage réalisé
- Déploiement sur Heroku
- Qualité globale du rendu (= application qui ressemble à quelque chose, un minimum de mise en page, orthographe propre, composants s'appuyant sur Windmill ou stylés à la main).

TP2.1

- Composants fonctionnels React pour le `Slideshow`, les `Slides`, la `Toolbar`.
- Styling des composants avec Tailwind,
- Gestions des routes pour les transparents

TP2.2

- Store qui contient l'état de l'application
- Le flux de données suit le flow React, des actions sont déclarées, et les changements d'états passent par des actions unitaires qui modifient le store.
- Les changement sont des fonctions qui renvoient un nouvel état (immutabilité) dans le reducer.
- Redux pour la gestion avancée des états
- Suivant/precedent change l'URI. Changer la route dans la barre d'URL du navigateur change l'état de l'application.

TP2.3

- Implémentation des Websockets côté client et serveur
- Synchronisation du transparent affiché entre les dispositifs s'appuyant sur un middleware
- Adaptation du contenu au dispositif (routage selon le dispositif) et affichage des bons composants.
- Gestion du plein écran.

TP2.4

- Gestion différenciée des pointer-events.
- Gestion des gestes pour des commandes suivant, précédent.
- Les commandes associées aux gestes sont bien propagées et permettent de contrôler un dispositif à distance.