SIMPLON.CO

COOKIE CLICKER REMAKE

PROJET EN ÉQUIPE

CSS + JavaScript

Interactions et animations

BOM - DOM – Événements

Logique algorithmique

0. Avant de commencer

Quelques conseils/contraintes pour votre bien !

Architecture du CSS

Commencez à organiser votre CSS avec plusieurs sous-dossier et plusieurs fichiers. Ça vous permettra de rétrécir le nombre de lignes de vos fichiers, et donc d'avoir un code plus lisible et facilement maintenable.

Votre fichier HTML ne se lie qu'à un seul fichier CSS qui sera le fichier principal qui va indexer et importer tous les autres.

La fonction magique c'est **@import** ! Et attention à l'ordre des imports, c'est très important !

Exemple d'architecture du CSS :

```
css/
  base/
    _fonts.css          // @font-face
    _reset.css          // CSS Reset
    _typography.css     // Règles génériques sur le texte
    ...
  components/
    _buttons.css        // Les différents boutons que vous utilisez partout sur votre site
    _carousel.css       // Votre carousel
    _inputs.css         // Les différents inputs que vous utilisez partout dans les formulaires
    ...
  layout/
    _grid.css           // Système de grille de votre layout
    _header.css         // Règles spécifiques du header
    _footer.css         // Règles spécifiques du footer
    _forms.css          // Règles spécifiques des formulaires
    ...
  pages/
    _contact.css        // Règles concernant le header
    _home.css           // Règles spécifiques de la home page
    ...
  main.css              // Fichier principal qui sera lié au(x) fichier(s) HTML et qui importe tous les fichiers CSS
```

C'est très inspiré des architectures SASS. Le _ est une convention SASS qui indique qu'il s'agit d'un fichier partiel qui sera importé par un autre fichier (en l'occurence le fichier principal).

De la veille utile sur les bonnes pratiques CSS et les architectures CSS :

- Atomic Design
- BEM
- ITCSS (Inverted Triangle CSS)
- Une petite histoire intéressantes sur les différentes architectures

Architecture du JavaScript

Dans ce projet vous allez avoir beaucoup plus de JavaScript que dans les "petits" challenge de Wes Bos. Vous allez avoir plusieurs fonctionnalités dans le même projet, ça vite devenir le chaos.

Comme pour le CSS, il serait bienvenu de commencer à organiser son code, là aussi pour qu'il soit plus lisible et facilement maintenable. De la même manière que pour le CSS, votre fichier HTML sera lié à un seul fichier JavaScript qui exécutera les fonctions principales et appellera les modules JavaScript dont il a besoin pour exécuter ses tâches.

De plus, pour ce projet vous allez aussi faire de la POO (Programmation Orientée Objet). Une bonne pratique en POO, je dirais même la base, c'est que chaque classe a son propre fichier.

Pour créer des modules, les rendre disponible à l'export, et les importer, vous allez devoir vous familiariser avec les fonctionnalités **import** et **export** de **ES6**. À lire : **le guide de MDN sur les modules JavaScript** et **un article explicatif** du blog MozFr.

Oui mais... les navigateurs ils n'aiment pas l'approche modulaire et les imports dynamiques...
Sauf les versions récentes de ces derniers comme on peut le constater grâce à **Can I use**.

Pour pouvoir exécuter du JavaScript "modulaire", il faut prévenir le navigateur comme ceci :

```
<script type="module" src="/js/main.mjs"></script>
```

La convention veut que tous les fichiers contenant un module JavaScript portent l'extension **.mjs** pour... Module JavaScript... (waouw quelle révolution!).
/! Cette règle n'est valable que pour des modules JavaScript NATIFS qui sont directement exécutés dans le navigateur.

Exemple d'architecture du JavaScript :

```
js/
  classes/
    garage.mjs          // La classe Garage
    car.mjs             // La classe Car
    ...
  dom/
    gallery.mjs         // Script qui affiche la galerie de voiture de mon garage au chargement de la homepage
    ...
  utils/
    string.mjs          // Des fonctions utilitaires que j'ai créées pour manipuler les chaînes de caractères
    ...
  main.mjs              // Script principal qui sera lié au fichier HTML et qui exécutera les actions JavaScript
```

1. Mise en page de départ

Un fichier HTML contenant le squelette de départ et les assets (images, sons, favicon, fonts) sont fournis **sur un dépôt GitHub**.

La première mission est de construire la mise en page de départ à l'aide de **CSS uniquement** **ET sans modifier** la structure HTML fournie.

/!\ Pas de mobile first, très peu de responsive, pas de media queries.
Le device ciblé est desktop/laptop uniquement. On doit pouvoir afficher le jeu correctement (toutes les sections sont visibles, sans défilement horizontal) sur une résolution minimale de 1024x768.

Spécifications globales

- Afficher la favicon qui est dans le dossier *favicons*
- Fond d'écran : bg-blue.jpg
- Polices de caractères : Merriweather, serif
- Taille de base de la police : 16px
- Couleur de base de la police : blanc (#ffffff)
- Les titres de niveau 1 et 2 ont une taille proportionnelle à 1.5x la taille de base, sont gras, alignés au centre et ont deux ombres :
    - une ombre de couleur noir (#000000) avec aucun offset et un blur-radius de 4px
    - une ombre de couleur noir (#000000) avec aucun offset-x, un offset-y de 1px et un blur-radius de 4px

- Aucun texte, aucune image, aucun bloc, aucun élément ne peut être sélectionner/mis en surbrillance par l'utilisateur

Spécifications de la grille

- 3 sections (*Bakery*, *Production lines* et *Store*) séparées chacune par un séparateur
- Les 3 sections et les séparateurs doivent occupés toute la hauteur
- Les 3 sections ont un padding-top de 15px
- Pour la section *Bakery* :
  - Largeur minimale est égale à 250px
  - Largeur maximale est égale à 400px
- Pour la section *Production lines* :
  - Largeur minimale est égale à 500px
  - Cette section occupera toujours au moins 2 fois plus d'espace que les deux autres
- Pour la section *Store* :
  - Largeur minimale égale à 150px
  - Largeur maximale égale à 300px
- Pour les séparateurs :
  - Image : panel-vertical.png
  - Largeur fixe de 16px

/! *Les spécifications des sections sont volontairement moins précises. C'est là qu'est le challenge ! Je vous laisse apprécier le positionnement, les mesures et le style des différents éléments, mais certaines choses sont tout de même spécifiées. Il faut être au plus proche de l'impression écran fournie ci-dessus.*

Spécifications de la section *Bakery*

- Fond :
  - Il doit être assombri avec un joli dégradé…
    mais il n'y a pas d'image de fond pour la section…
- Bloc avec les textes :
  - Il prend tout l'espace en largeur de la section
  - Le fond est légèrement sombre et transparent
  - Le texte présentant le nombre total de cookies doit être plus grand que le texte présentant le nombre de cookies produits par seconde
- Cookie :
  - Il mesure 225x225px et est centré dans la section
  - Il a une forme circulaire
  - Il a une ombre floue blanche (#ffffff)
  - Au survol, le curseur est une main avec un doigt
  - Au survol, il s'anime pour s'agrandir jusqu'à 1.1 fois sa taille d'origine (voir animation ci-dessous) et retrouve sa taille normale lorsque la souris le quitte avec la même animation

    - Au clic, il se rétrécit jusqu'à 0.95 fois sa taille d'origine en utilisant la même animation que celle décrite précédemment et lors du relâchement il retrouve sa taille normale ou celle au survol (en fonction d'où se trouve la souris)

Spécifications de la section *Production lines*

- La barre de scroll vertical doit toujours s'afficher
- Fond :
    - Il doit être assombri avec un joli dégradé léger dans le haut de la section… mais il n'y a pas d'image de fond pour la section là non plus…
- Séparateur horizontal (il sera utilisé à plusieurs reprises dans la section plus tard) :
    - Fond : panel-horizontal.png
    - Hauteur fixe de 16px
    - Aucune marge, sauf pour le premier séparateur de la section : margin-top de 15px
      Note: pour visualiser l'effet, vous ajoutez temporairement (juste le temps de trouver la bonne règle CSS) un ou plusieurs autres séparateurs ***<hr />*** *dans la section*

Spécifications de la section *Store*

- La barre de scroll vertical doit toujours s'afficher
- Fond :
    - Il doit être assombri avec à peu près le même dégradé que pour la section *Bakery*
- L'encart *Buildings* :
    - Fond : panel-horizontal.png
    - Marge du haut : 15px
    - Hauteur minimale : 16px
    - Le texte "Buildings" doit apparaître au-dessus du fond au survol de la souris
        - Pas d'ajout d'élément HTML dans le code ! C'est le moment d'apprendre à utiliser des techniques CSS un plus avancées -> le pseudo-élément *::before*
        - Taille du texte : 11px
        - Texte gras et en majuscules (via le CSS)
        - Deux ombres :
            - une ombre de couleur noir (#000000) avec aucun offset et un blur-radius de 4px
            - une ombre de couleur noir (#000000) avec aucun offset-x, un offset-y de 1px et un blur-radius de 4px
        - Une transition douce entre l'état normal et l'état au survol

2. Initialisation du jeu

La deuxième mission est d'une importance capitale et risque d'être un peu longue. Nous allons poser les bases pour le reste du projet. Nous allons donc la découper en plusieurs étapes. L'objectif est d'initialiser le jeu. Pour ce faire, nous avons besoin de configurer toutes les valeurs pour commencer le jeu, afficher celles qui sont présentes sur l'interface, initialiser et afficher les deux premières tuiles du *Store*.

Le code JavaScript va découper en deux parties :

- les classes et les objets qui contiennent la logique du jeu
- les scripts de manipulation du DOM

Á chaque nouvelle interaction, chaque nouvel événement, il faudra potentiellement ajouter des éléments HTML dynamiquement avec JavaScript et coder le CSS nécessaire pour afficher correctement ces nouveaux éléments HTML.

/! *Comme nous avons envie de se simplifier la vie, nous allons coder uniquement ce dont nous avons besoin, c'est-à-dire le strict minimum. Même si parfois le code nous paraît un peu bête, le mieux c'est de construire son application de manière incrémentale.*
On ne va pas se poser la question de savoir si nous avons besoin de telle ou telle fonctionnalité plus tard... C'est la pire chose à faire ! C'est difficile d'anticiper correctement exactement ce dont nous aurons besoin plus tard et c'est le meilleur moyen de complexifier son code et de s'y perdre.

Créer la logique de base pour initialiser le jeu

Le coeur du jeu se trouve dans les classes et les objets que nous allons créer et manipuler à partir de ces classes.

Pour commencer nous en aurons 2 :*Bakery* et *Building*. La boulangerie (*Bakery*) possède des bâtiments de production de cookies (*Building*).

La classe *Bakery*

C'est la classe qui contient toutes les informations concernant la boulangerie. On y retrouvera des informations telles que le stock de cookies, les bâtiments de production, les améliorations, etc.

Bakery	
- name	// le nom de la boulangerie
- cookies	// le stock actuel de cookies de la boulangerie
- buildings	// la liste des bâtiments de production que possède la boulangerie
- cookiesPerClick	// le nombre de cookies produits à chaque clic
- cookiesPerSecond	// le nombre de cookies produits automatiquement par seconde

Le constructeur initialise l'attribut *name* à une valeur par défaut de votre choix, l'attribut *cookies* à 0, l'attribut *cookiesPerClick* à 1 et l'attribut *cookiesPerSecond* à 0. À l'attribut *buildings* on assignera un tableau qui contiendra la liste des bâtiments de production (*Building*) disponibles dans le jeu.

/! Pour pouvoir ajouter des bâtiments de production à notre boulangerie, il nous faut des données. Les données "fixes" du jeu sont fournies dans un fichier nommé **data.mjs**. Ce module JavaScript exporte un tableau *buildings* contenant la liste de tous bâtiments de production disponibles dans le jeu, avec les caractéristiques détaillées pour chaque bâtiments.

Problème : je ne peux pas créer un bâtiment dans l'état actuel. Next step... la classe *Building*...

La classe *Building*

C'est la classe qui contient toutes les informations concernant les bâtiments de production. On y retrouvera toutes les caractéristiques des bâtiments.

Building	
- name	// le nom du bâtiment
- description	// la description du bâtiment
- number	// le nombre de bâtiments du même type achetés
- cookiesPerSecond	// le nombre de cookies produits par seconde
- cost	// le coût de la prochaine évolution

Le constructeur recevra un objet en paramètre qui contiendra toutes les propriétés du *Building* sauf le *number*. Le *number* sera initialisé par défaut à 0.

/! Pour le moment, nous n'ajoutons aucun getter, aucun setter, aucune méthode à nos classe. Nous ajouterons ce dont nous aurons réellement besoin dans l'étape suivante *(spoil : ce seront quelques getters)*.

Afficher les éléments de base de l'interface

Maintenant que nous avons la logique de base du jeu qui est construite, nous allons nous attarder sur l'initialisation de certains éléments dans l'interface.

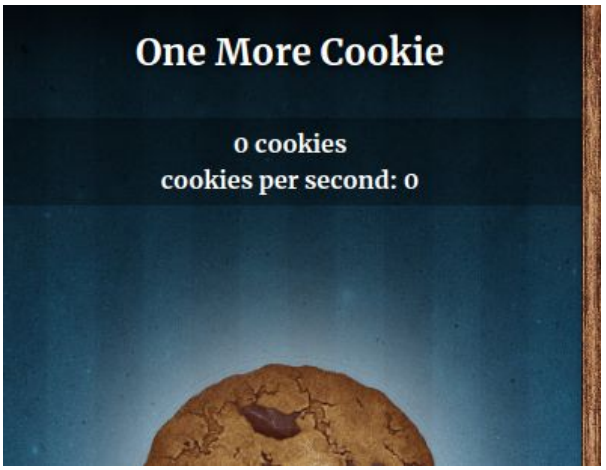
Tout d'abord, dans notre script principal il faudra créer un nouvel objet de type *Bakery*.

```
const myBakery = new Bakery();
```

Cet objet contient toutes les informations du jeu (de notre boulangerie en l'occurence) et c'est avec lui et les objets qu'il contient que nous allons interagir.

Les informations de la boulangerie

Au chargement de la page, il faut afficher dynamiquement le nom de la boulangerie, le stock initial de cookies et le nombre de cookies produits par seconde.



Les bâtiments du *Store*

Pareil, au chargement de la page, il faut afficher dynamiquement les deux premières tuiles du *Store* permettant d'acheter des bâtiments de production.

Cette fois-ci il faudra au préalable créer les éléments HTML avec les bons id et les bonnes classes.

Voici la structure HTML de la tuile pour le bâtiment de type *Cursor* :

```
<div id="building-cursor" class="locked disabled">
  <div class="icon"></div>
  <div class="name">Cursor</div>
  <div class="cost">15</div>
  <div class="number"></div>
</div>
```

L'id doit être généré dynamiquement à partir du nom du bâtiment. L'id doit toujours être préfixé par **building-** et le nom du bâtiment doit être en **minuscule**.

Une tuile peut être bloquée (**locked**) ou débloquée (**unlocked**), et activée (**enabled**) ou désactivée (**disabled**). Lorsqu'elle apparaît elle est toujours bloquée et désactivée par défaut.

/! Le mécanisme des tuiles est expliqué en détail dans l'annexe B à la fin de ce document.

Dans la div avec la classe **.name**, le nom du bâtiment de production est affiché dynamiquement.
Dans la div avec la classe **.cost**, le coût du bâtiment de production est affiché dynamiquement.

Ne reste plus que le CSS

Pour chaque tuile (chaque div enfant de la div #buildings) :

- Fond : store-tile.jpg
- Le curseur est une main avec un doigt
- Lorsqu'une tuile est bloquée (classe .locked) la silhouette de l'icône est affichée
- Lorsqu'une tuile est désactivée (classe .disabled) :
    - Le texte du coût du bâtiment est de couleur corail (#ff6666)
    - L'image de fond de la tuile est assombrie *

* Là aussi, pas question d'ajouter du HTML pour y parvenir. C'est le moment d'apprendre à utiliser des techniques CSS un peu plus avancées, le pseudo-élément *::after* est un ami.

Pour afficher l'icône ou sa silhouette dans la div avec la classe .icon, servez-vous de l'image *buildings.png*. Toutes les icônes des bâtiments de production sont sur la même image…

Dans la div avec la classe **.name** :

- La taille du texte est proportionnelle à 1.3 fois la taille de référence

Dans la div avec la classe **.cost** :

- Fond : money.png
- La taille du texte est proportionnelle à 0.7 fois la taille de référence
- Le texte est vert (#66ff66) par défaut
- Lorsque la tuile est désactivée (classe .disabled) le texte est corail (#ff6666)

Commun aux div avec la classe **.name** et aux div avec la classe **.cost** :

- Le texte est gras
- Deux ombres :
    - une ombre de couleur noir (#000000) avec aucun offset et un blur-radius de 4px
    - une ombre de couleur noir (#000000) avec aucun offset-x, un offset-y de 1px et un blur-radius de 4px

Au survol d'une tuile une ombre intérieure blanche (#ffffff) avec aucun offset, un blur-radius de 15px et un spread-radius de 3px.

Une transition douce entre les différents états (normal, survol, actif) est souhaitable.

3. Un clic, un cookie !

C'est là que les choses intéressantes commencent ! Pour cette troisième mission, on aimerait bien qu'il se passe quelque chose lorsqu'on clique sur le "big" cookie.

Vous procéderez toujours en deux étapes, et dans cet ordre :

- ajouter les fonctionnalités nécessaire dans le coeur du jeu (dans les classes)
- ajouter les fonctionnalités qui concernent l'affichage et les interactions dans le DOM

Ajouter un cookie à chaque clic

Il est temps d'augmenter le stock de cookies !

Ajouter une fonctionnalité dans la classe Bakery

Dans la classe *Bakery*, vous allez ajouter une méthode `bakeCookies(howMany)` qui permettra d'ajouter le nombre de cookies précisé à l'attribut `cookies` (qui représente le stock de cookies).

Mettre à jour le stock de la boulangerie

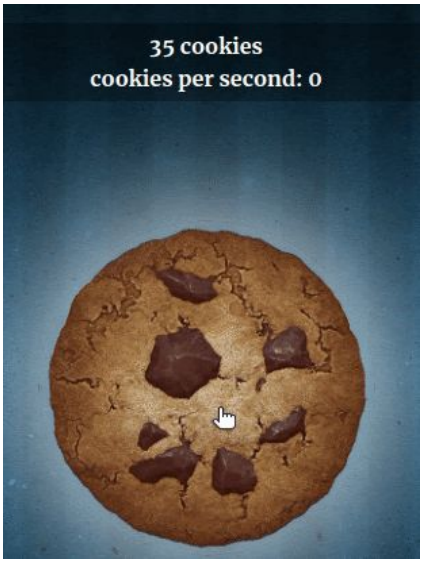
Il va se passer des choses lorsqu'on clique sur le "big" cookie...

1| On a une fonction à notre disposition pour ajouter des cookies au stock de notre boulangerie. Il ne reste plus qu'à s'en servir pour ajouter la valeur de `cookiesPerClick`.

2| Ensuite, on va mettre à jour l'affichage pour afficher le nouveau stock de cookies.

Une petite animation en plus

À chaque clic, le texte **+1** doit apparaître au-dessus du curseur et se déplacer vers le haut pour disparaître progressivement.



Le texte fait 1.3 fois la taille de référence et est gras. Il est contenu dans une div qui sera ajoutée au DOM au moment du clic et qui sera supprimée du DOM à la fin de l'animation.

Le déplacement vers le haut se fait sur 200px sur une durée d'environ 3 secondes (la durée de l'animation est à votre discrétion).

À 20% de l'animation, le texte aura une opacité de 50%. À 50% de l'animation il aura une opacité de 20%. À 100% il aura disparu.

Jouer un son aléatoire au clic

Lorsqu'on clique sur le "big" cookie, on veut qu'un son aléatoirement choisi parmi les sons `clickX.mp3` soit joué.

Vous pouvez le faire éventuellement en deux étapes. D'abord joué le son `click1.mp3`, et puis ajouté l'aléatoire ensuite.

Ça donnera plus d'humanité à votre jeu.

Activation des tuiles et affichage des suivantes

Dans le *Store*, la tuile d'un bâtiment de production se débloque (**unlocked**) et s'active (**enabled**) lorsque le stock de cookies est assez grand pour acheter le bâtiment.

Au même moment, la tuile du prochain bâtiment de la liste qui n'est pas affichée apparaît et cette tuile est bloquée et désactivée.

/! Le mécanisme des tuiles est expliqué en détail dans l'annexe B à la fin de ce document.

Un peu de CSS

Normalement, vous n'avez pas encore le CSS des classes **.unlocked** et **.enabled**.

Quelques ajustements sont à faire :

- Il faut afficher l'icône colorée lorsque la tuile se débloque. Souvenez-vous que les icônes sont sur une seule image (buildings.png).
- Au clic sur une tuile débloquée et activée, l'ombre intérieure devient noire.

4. Des cookies sans rien faire !

Flemme de cliquer pour produire des cookies ! Même si ça muscle le doigt ! Cette quatrième mission va consister à développer une fonctionnalité qui permet de générer des cookies automatiquement.

Acheter un bâtiment de type *Cursor*

Nous avons déjà développé le mécanisme pour ajouter des cookies, pas celui pour acheter un bâtiment de production.

Ajouter une fonctionnalité dans la classe *Building*

Vous allez ajouter une méthode `buy()` dans la classe *Building*. Cette méthode nous permettra d'acheter un bâtiment de production. Elle aura deux rôles :

- incrémenter le nombre de bâtiments (`number`) de 1
- augmenter le coût (`cost`) du prochain bâtiment de 15% (le coût doit toujours être un nombre entier, le nouveau coût sera arrondi au nombre entier supérieur)

Mettre à jour la tuile dans le *Store*

Au clic sur la tuile, deux actions vont se produire :

- La première, c'est d'acheter de manière effective un bâtiment (on vient de coder une fonction pour le faire).
- La seconde, c'est de mettre à jour la tuile dans le *Store*. Il faut afficher le nouveau côut dans la div **.cost** et le nouveau nombre de bâtiments possédés dans la div **.number**.

Jouer un son aléatoire au clic

Lorsqu'on clique sur la tuile, on veut qu'un son aléatoirement choisi parmi les sons `buyX.mp3` soit joué. Vous pouvez le faire éventuellement en deux étapes. D'abord joué le son `buy1.mp3`, et puis ajouté l'aléatoire ensuite.

Oui mais… c'est pas gratuit !

On a un truc qui fonctionne à peu près sauf que… il faut passer à la caisse ! Et oui ! C'est pas gratuit cette histoire, Madame !

Ajouter une fonctionnalité dans la classe *Bakery*

Vous allez ajouter une méthode `buyBuilding(which)` dans la classe Bakery. Cette méthode nous permettra d'acheter un bâtiment de production d'un type donné (`which`). Elle aura plusieurs rôles :

- faire l'opération d'achat du bâtiment souhaité (en appelant la fonction `buy()` du *Building*)
- diminuer le stock de cookies de la boulangerie en fonction du coût de l'achat
- recalculer la valeur de `cookiesPerSecond`

Mettre à jour l'affichage du stock de cookies

Au clic sur la tuile, plutôt que d'utiliser la méthode `buy()` du bâtiment concerné, on passera par la méthode `buyBuilding(which)` de la boulangerie en passant en paramètre le type de bâtiment que l'on souhaite acheté.

(i) *Si vous ne l'aviez pas compris, c'est par ce mécanisme que l'on peut accéder au stock de cookies de la boulangerie et avoir un impact dessus. Un objet de type* Building *n'a aucun accès aux données d'un objet de type* Bakery *car il n'a pas connaissance de ce dernier. Par contre contre un objet de type* Bakery *possède un tableau d'objets de type* Building*. Lui connaît les* Building *et peut donc les utiliser.*

Pour finir, il faudra mettre à jour l'affichage du stock de cookies pour qu'il corresponde à ce que la boulangerie possède réellement.

La banque ne fait pas crédit !

On ne devrait pas pouvoir acheter un bâtiment lorsqu'on n'a pas les fonds nécessaires. Il y a deux endroits où il faut gérer ça :

- au niveau de l'intelligence du jeu
- dans l'interface graphique

Je vous laisse un peu galéré… sinon c'est pas drôle…

Produire les cookies automatiquement

Selon les règles du jeu, chaque bâtiment de production de type *Cursor* produit 0.1 cookies par secondes, soit l'équivalent de 1 cookie toutes les 10 secondes. Je vous ai beaucoup guidé jusque là, je vous laisse réfléchir à une solution.

Un affichage propre

Les nombres

Même si le stock de cookies de la boulangerie est égal à 13,2 cookies, on souhaiterait avoir un affichage propre du nombre de cookies sans décimale. On arrondira à l'entier inférieur. On ne veut jamais afficher un nombre de cookies que l'on ne possède pas !

Concernant l'affichage du nombre de cookies par seconde, on souhaiterait avoir un affichage sans décimale lorsque le nombre est entier et avec 1 décimale maximum lorsque le nombre est flottant.

Activation et désactivation des tuiles

Il est probable qu'il reste une imperfection... il faut que les tuiles se désactivent (**.disabled**) et s'activent (**.enabled**) instantanément en fonction du stock de cookies disponible.

Il y a plusieurs moments où il faut vérifier l'état du stock par rapport au coût des différentes tuiles :

- Lorsque j'achète un bâtiment, si mon stock de cookies est devenu trop petit, la tuile se désactive.
- Lorsque je clique sur le "big" cookie, si mon stock de cookies devient assez conséquent, la tuile d'un bâtiment que je peux acheter s'active.
- Lorsque mon stock de cookies devient assez conséquent grâce à la production automatique, la tuile d'un bâtiment que je peux acheter s'active.

5. J'ai faim, mémé !

On va avoir tous les mêmes mécanismes pour *Grandma* qu'avec *Cursor*. Normalement, pas d'intelligence à développer, juste des interactions dans l'interface.

La nouveauté c'est que le premier achat d'une mémé affiche un nouvel élément dans la section *Production lines* et on affichera autant de mémés qu'on en possède dans cet élément.

Au premier achat (clic) d'une ligne de production (en l'occurence, Grandma dans notre cas), il faut ajouter la ligne de production avec 1 mémé, et un séparateur horizontal :

```
<div id="production-grandma">
  <div class="grandma"></div>
</div>
<hr>
```

Faisons ça intelligemment, comme pour tuiles du *Store*. L'id doit être généré dynamiquement à partir du nom du bâtiment. L'id doit toujours être préfixé par **production-** et le nom du bâtiment doit être en **minuscule**.

Chaque nouvel achat d'une mémé ajoutera une mémé ajoutera une div avec la classe **grandma** en **minuscule** (là aussi généré dynamiquement à partir du nom du bâtiment).

Chaque mémé est légèrement décalée par rapport à la précédente comme vous le voyez sur l'image ci-dessus. La deuxième apparaît au-dessus de la première et la troisième au-dessus de la deuxième. On fait des "colonnes" de 3 mémés :)

(i) *J'insiste sur la génération dynamique parce que c'est une contrainte vraiment légère et qui vous permet de produire un code réutilisable pour les prochaines briques du jeu. Ok, je dis et je répète qu'il faut faire le code minimal pour que ça fonctionne mais ne soyons pas trop idiot non plus.*

Spécifications pour la div #production-grandma :

- Fond : grandma-background.png
- Hauteur : 128px

Spécifications pour la div .grandma :

- Fond : grandma.png
- Taille : 64x64px

Je vous laisse trouvé des valeurs qui vous conviennent pour les décalages.

6. Et si on sauvegardait la partie ?

En l'état, si je ferme mon navigateur, la partie est perdue. Le challenge ici est "simple" : je veux pouvoir sauvegardé ma partie pour la récupérer lorsque je me reconnecte au jeu.

Vous allez faire ça avec le `localStorage` en stockant l'objet principal *Bakery* qui contient toutes les infos de votre partie.

Vous allez faire une sauvegarde automatique une fois toutes les minutes. Pas la peine d'enregistrer 50 fois l'objet, on ne conserve que la sauvegarde la plus récente.

Lorsqu'après avoir fermé votre navigateur, vous l'ouvrez à nouveau dans le jeu, il faudra récupérer la sauvegarde et réinitialiser l'interface avec les informations qu'elle contient.

7. Des bonus

SI ET SEULEMENT SI vous avez terminé le projet avant le délai de livraison, voici quelques bonus à faire, au choix :

Affichage propre des grands nombres

Que ce soit dans l'affichage du stock de cookies, du nombre de cookies automatiquement produits chaque seconde ou dans les tuiles du *Store*, on pourrait largement améliorer l'affichage des grands nombres pour plus de lisibilité (et pour pas casser l'interface).

On va ajouter les séparateurs de milliers, et transformer des suites de chiffres en mots sur les très grands nombres.

Exemple :

```
1000 -> 1 000
19342 -> 19 342
1000000 -> 1 millions
1234000 -> 1.234 millions
17958154 -> 17.958 millions
17958999 -> 17.958 millions
3000000000 -> 3 billions
36987654321 -> 36,987 billions
7000000000000 -> 7 trillions
etc.
```

/! Ça ne doit pas prendre le dessus sur les règles d'affichage déjà énoncées dans le brief. À savoir que l'affichage du stock de cookies est toujours un nombre entier arrondi à l'entier inférieur par exemple.

Ajustement aléatoire du positionnement de certains éléments

Pourquoi ne pas ajuster de quelques pixels le positionnement de certains éléments de manière aléatoire de façon à rendre l'interface plus agréable et humaine ?

Il y a deux choses sur lesquelles on peut le faire :

- Le label qui indique le nombre de cookies produit au clic
- Les mémés, les fermes, les mines, les usines dans les lignes de production

Golden Cookie

De temps à autres, de manière totalement aléatoire, apparaît un cookie en or quelque part sur l'écran. Il apparaît progressivement (en 5 secondes) par son opacité et sa taille et restera à l'écran 15 secondes. Durant ces 15 secondes l'image grossit et se rétrécit en continu. Si on ne clique pas dessus, il disparaîtra progressivement de la même manière qu'il est apparu.

L'image est `gold-cookie.png`.

Lorsqu'on clique dessus, il disparaît et :

- soit on gagne un bonus de cookies équivalent à 10% du stock actuel
- soit on gagne un bonus de 13 secondes durant lequel la valeur des clics est multipliée par 5

On a 50% de chances que ce soit le premier bonus et 50% de chances que ce soit le second.

Pour savoir si on fait apparaître un cookie en or, je vous propose une formule simplifiée : une fois toutes les 30 secondes, on a 5% de chance de faire apparaître un cookie en or.

Lorsque le bonus est activé pour 13 secondes, le fond d'écran passe de bleu à gold durant les 13 secondes pour redevenir bleu ensuite. Faites-ça avec un transition douce. Le fond : `bg-gold.jpg`

Cookie Shower 1

Lorsque des cookies sont automatiquement produits, on fera apparaître des mini-cookies à une position aléatoire sur la largeur de la section. Ils descendront en passant derrière les div avec le nom de la boulangerie, le stock de cookies, le nombre de cookies produits par seconde et le "big"cookie.

On commencera à générer des mini-cookies à partir de 1 *cookies per second*, et on en générera maximum 7 à la fois.

Vous trouverez des images de mini-cookies dans le fichier `icons.png` (les icônes font 48x48px sur l'image). Vous pouvez commencer par utiliser le premier mini-cookie (1ère colonne, 4ème rangée).

Pour plus de réalisme, ajoutez une rotation aléatoire pour que le mini-cookie généré ne tombe pas toujours dans le même sens.

Bonus : lorsqu'un mini-cookie est généré, l'image est choisi aléatoirement parmi les 10 premières icônes de la 4ème rangée.

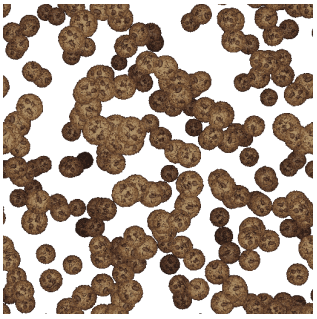
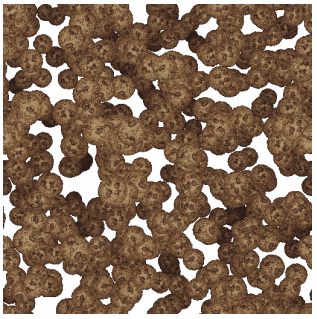
/! Pas de découpage dans les images. Vous gérer le positionnement sur icons.png avec le CSS.

Cookie Shower 2

Lorsque des cookies sont automatiquement produits, on fera apparaître un fond avec plein de cookies dans la section **#bakery**. Le fond se situe entre le dégradé, passe derrière les div avec le nom de la boulangerie, le stock de cookies, le nombre de cookies produits par seconde et le "big" cookie, et est positionné derrière les mini-cookies du bonus précédent.

En fonction du nombre de cookies produits par seconde, on affichera un fond différent :

- 50+ cookies : `cookie-shower1.png`
- 500+ cookies : `cookie-shower2.png`
- 5000+ cookies : `cookie-shower3.png`

Lignes de production de type *Farm*, *Mine* et *Factory*

En principe, vous avez déjà développé toutes les mécaniques nécessaires. La seule variante, c'est qu'il n'y a que pour les mémés qu'on affiche 3 mémés par "colonne". Pour les autres, c'est l'un à côté de l'autre. Inspirez-vous du jeu original.

/! Vous allez ajouter des EventListener sur des éléments qui ne seront pas affichés à l'initialisation d'une partie vierge...

Mémés aléatoires

Lorsqu'on achète une *Farm* pour la première fois, on convertit aléatoirement 15% de nos mémés en fermières. Lorsqu'on achète une *Mine* pour la première fois, on convertit aléatoirement 15% de nos mémés en minières. Lorsqu'on achète une *Factory* pour la première fois, on convertit aléatoirement 15% de nos mémés en ouvrières.

Les mémés sont trop vieilles pour faire trop de reconversions professionnelles. Si elles sont devenues fermières, elle ne peuvent pas devenir minières ou ouvrières ensuite.

Des mini-cookies animés près du curseur à chaque clic

À chaque clic sur le "big" cookie, un mini-cookie apparaît sur le curseur, fait un petit bon et s'en va vers le bas en disparaissant progressivement.

Vous trouverez des images de mini-cookies dans le fichier `icons.png` (les icônes font 48x48px sur l'image). Vous pouvez commencer par utiliser le premier mini-cookie (1ère colonne, 4ème rangée).

Pour plus de réalisme, ajoutez une rotation aléatoire pour que le mini-cookie généré ne soit pas toujours dans le même sens et ajustez de quelques pixels et de manière aléatoire la position d'apparition du mini-cookie.

Bonus 1 : lorsqu'un mini-cookie est généré, l'image est choisi aléatoirement parmi les 10 premières icônes de la 4ème rangée.

Bonus 2 : si vous êtes des fous, vous trouverez un moyen d'envoyer les mini-cookies aléatoirement sur les côtés avec des angles différents.

Des curseurs autour du "big" cookie

Tout est dans le titre. Vous pouvez le faire par étape :

- Une première sans animation : réussir à positionner les curseurs correctement autour du cookies. Max 50 cookies par cercle de curseurs. Inspirez-vous du jeu original.
- Une deuxième avec l'animation du curseur qui touche le cookie et qui passe d'un curseur à l'autre.
- Une troisième qui fait tourner tous les curseurs autour du cookies.

L'image du curseur : `cursor.png`

A. Les règles du jeu

Voici un résumé des règles principales et des mécanismes du jeu.

Le “big” cookie

Chaque clic sur le cookie produit de nouveaux cookies qui sont ajoutés dans le stock de la boulangerie.

Le nombre de cookies produits par clic peut être modifié au cours du jeu en achetant des améliorations.

Au début du jeu, chaque clic produit 1 nouveau cookie.

Les bâtiments de production

Ils permettent de produire automatiquement des cookies.

Chaque type de bâtiment produit un nombre différent de cookies. Plus la boulangerie possède de bâtiments de production, plus le nombre de cookies produits automatiquement est élevé.

À chaque fois que la boulangerie achète un bâtiment d’un certain type, le prochain bâtiment ce type coûtera 15% plus cher que le prix précédent.

Exemple :
La boulangerie achète une première ferme qui produit 8 cookies par seconde et coûte 1.100 cookies à l’achat. Pour acheter une seconde ferme, il faudra débourser 1.265 cookies.

Au début du jeu, la boulangerie ne possède aucun bâtiment de production. Il va falloir en acheter, et ce n’est pas gratuit ! On n’a rien sans effort… ;-)

La liste des Buildings se trouve **> ICI <**. Dans les données que je vous fournis, seuls les 5 premiers sont présents, à savoir :

- Cursor
- Grandma
- Farm
- Mine
- Factory

B. Les règles de l'interface

Voici un résumé des mécanismes liés à l'interface.

Mécanisme des tuiles du *Store*

Une tuile peut être bloquée (**locked**) ou débloquée (**unlocked**), et activée (**enabled**) ou désactivée (**disabled**). Lorsqu'elle apparaît elle est toujours bloquée et désactivée.

Le statut bloqué est le statut par défaut lorsqu'une tuile apparaît. Elle se débloque lorsque le stock de cookies est assez grand pour acheter le bâtiment. Une fois débloquée, elle ne se bloque plus jamais.

Le statut désactivé est le statut par défaut lorsqu'une tuile apparaît. Elle s'active lorsque le stock de cookies est assez grand pour acheter le bâtiment et se désactive lorsque le stock de cookies est trop petit.

Lorsqu'une tuile se débloque, elle s'active automatiquement et fait apparaître une nouvelle tuile (le prochain bâtiment de la liste) bloquée et désactivée.

Lorsqu'une tuile est bloquée (**locked**) :
- la silhouette de l'icône est affichée

Lorsqu'une tuile est débloquée (**unlocked**) :
- l'icône originale et en couleur est affichée

Lorsqu'une tuile est désactivée (**disabled**) :
- le texte du coût du bâtiment est de couleur corail (#ff6666)
- l'image de fond de la tuile est assombrie

Lorsqu'une tuile est activée (**enabled**) :
- le texte du coût du bâtiment est de couleur verte (#66ff66)

3 états possibles :
- débloqué (**unlocked**) et activé (**enabled**)
- débloqué (**unlocked**) et désactivé (**disabled**)
- bloqué (**locked**) et désactivé (**disabled**)