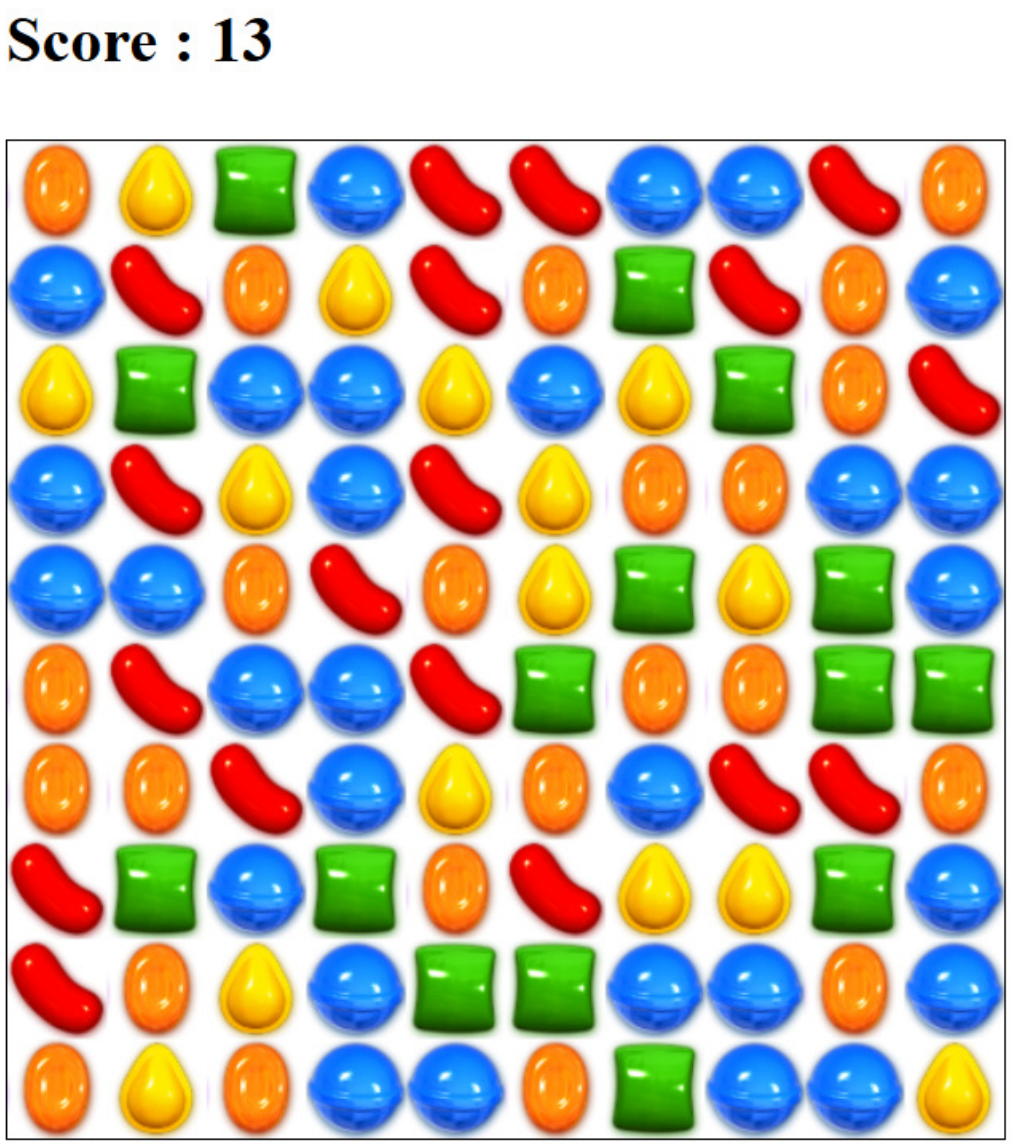
Candy Crush

Javascript

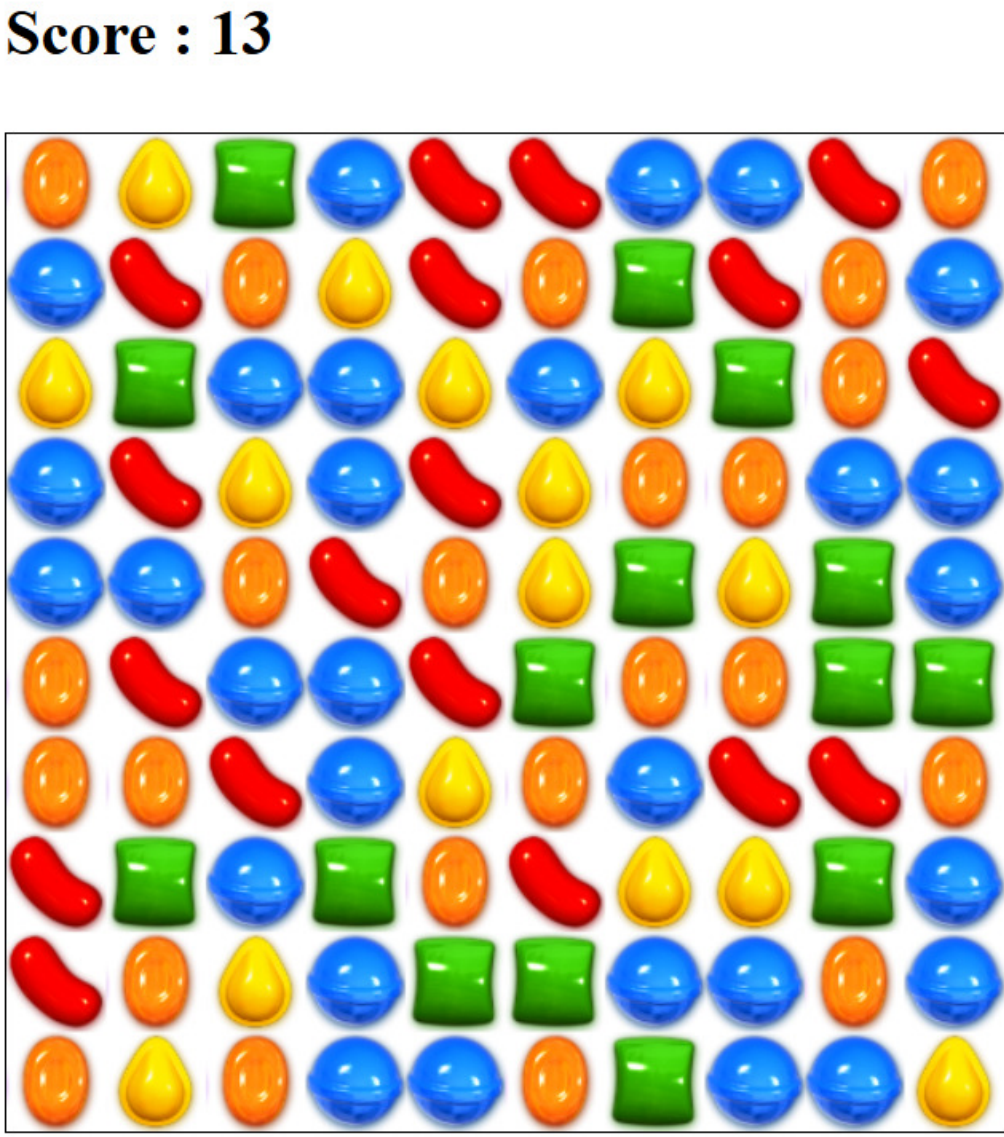


Candy Crush : MVC

1. Description du problème
2. Fonctions / Classe utilitaires
3. Modèle
4. Vue
5. Contrôleur
6. Gestion des évènements
7. Contrainte et description Génie Log. (diag classe etc..)

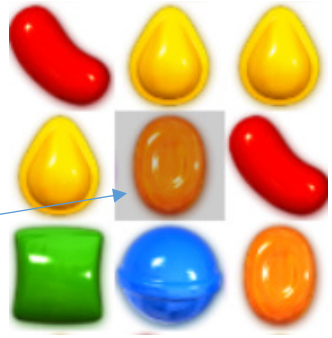
Candy Crush

- Lancement
  - Score : 0
  - Les bonbons tombent d’en haut
  - Animation
  - Aléatoire
  - 5 sortes / bonbons
  - Alignement : les bonbons disparaissent
  - D’autres bonbons viennent combler
- Stabilisation
  - plus d’alignements
  - La grille est pleine
  - Pas d’interaction avant



Candy Crush

- Interactions
  - L'objectif est de réaliser des lignes
  - Verticales et / ou horizontales
  - On sélectionne un bonbon / voisin
  - Sélection : grisé
  - Second bonbon :
  - Si pas d'échange possible : désélection
- Sinon :
  - Animation du swap
  - La ligne / colonne explose et re stabilisation (cf initialisation)
  - On compte les points



Remarque : Javascript et classes

```
class Rectangle {
  constructor(hauteur, largeur) {
    this.hauteur = hauteur;
    this.largeur = largeur;
  }
}
```

Candy Crush : le bonbon a son MVC !!!!

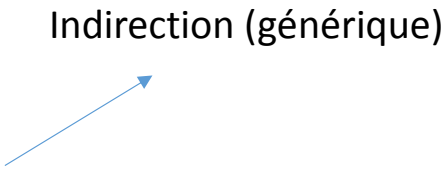
- Le modèle du bonbon : ses coordonnées, son état, sa destination
    - Les méthodes qui permettent de les lire / écrire (accesseurs)
- La vue du bonbon : son image, la méthode qui le dessine
- Le contrôleur du bonbon : ce qui modifie les valeurs dans le modèle, ce qui indique qu'il est ou pas en mouvement, ce qui décide ou pas de demander à la vue de mettre à jour etc..

ATTENTION !!

- Ne pas faire de modèle individuel de MVC
- On a une collection de bonbon
- Synchronisation

Candy Crush

- Objet graphique (Sprite)
- Classe LutinBonbon
  - Sert à afficher un bonbon
  - Constructor(*quel*, largeur, hauteur)
    - ***quel*** est le numéro du bonbon (1, 2, 3, 4, 5)
  - positionXY(x1,y1)
    - Change la position du lutin
  - marque(v)
    - Vrai/faux : si vrai alors quand on dessine le bonbon on dessine un cadre autours



Candy Crush

- Objet graphique (Sprite)
- dessin(contexte)
  - Dessine le sprite à sa position courante
  - Et dessine éventuellement un cadre s'il est marqué

- On souhaite gérer le déplacement en animation
- De la position courante vers une position 2
- On rajoute une position cible pour l'animation
- deplacementVers(x2,y2)
  - N'affiche rien : indique la cible du déplacement

Candy Crush

- Si la position cible n'est pas atteinte, alors le lutin est considéré comme
  - estEnMouvement() // vrai
- Il faut le faire un peu avancer
  - Donc modifier ses coordonnées

```
metAJour() { // met à jour les coordonnées du sprite si besoin
  if (! this.estEnMouvement() ) return // pas la peine
  if ( (this.x != this.versX) || (this.y != this.versY) ) {
            if (this.versX > this.x) this.x = this.x +5
            if (this.versX < this.x) this.x = this.x -5
            if (this.versY > this.y) this.y = this.y +5
            if (this.versY < this.y) this.y = this.y -5
          } else {
                  this.bouge = false;

          }
}
```

Candy Crush

- Donc pour animer un lutin
  - var bonbon = new LutinBonbon(1, 40, 40)
  - bonbon.position(20,20) // force la position mais n'affiche rien
  - bonbon.deplacementVers(x2,y2) // ne déplace rien, indique la cible à atteindre
  - bonbon.metAJour() // déplace un peu la position courante
  - bonbon.dessin(contexte) // dessine le sprite en position (effacer écran ?)
  - bonbon.metAJour() // déplace un peu la position courante
  - bonbon.dessin(contexte) // dessine le sprite en position
  - bonbon.metAJour() // déplace un peu la position courante
  - bonbon.dessin(contexte) ….
  - …. On arrête quand le sprite a sa position finale

Candy Crush

- Donc pour animer un lutin
  - var bonbon = new LutinBonbon(1, 40, 40)
  - bonbon.position(20,20) // force la position mais n'affiche rien
  - bonbon.deplacementVers(x2,y2) // ne déplace rien, indique la cible à atteindre

  - do {
    bonbon.metAJour() // déplace un peu la position courante
    bonbon.dessin(contexte) // dessine le sprite en position (effacer écran)
  } while (bonbon.estEnMouvement())

IMPOSSIBLE

Candy Crush

- En Javascript
  - Asynchrone

```
var bonbon = new LutinBonbon(1, 40, 40)
bonbon.position(20,20)
bonbon.deplacementVers(x2,y2)
anime() // fin des instruction du programme
// pas d'instruction en dessous

function anime(contexte) {
        bonbon.metAJour()
        bonbon.dessin(contexte)
        if (bonbon.estEnMouvement())
                setTimeout(anime, 100)
}
```

ASYNChrone

Candy Crush

- La Vue va contenir **tous** les sprites
  - Sans doute dans un tableau de sprites
- Chaque sprite/lutin de la vue a ses propres coordonnées (c'est un objet)
- Chaque sprite de la vue peut être en mouvement ou pas
- Quand on affiche la vue, on demande à chaque sprite de s'afficher
- Quand on demande à la vue de se mettre à jour, on demande à chacun de ses sprites de se remettre à jour
  - Un par un
  - PUIS on redessine toute la vue !

Bien séparer ce qui met à jour la position attribut
Et ce qui affiche !...
Une fonction ne fait pas les 2 choses

Candy Crush : pour la VUE

```
dessineTout() { // Vue
  effaceEcran()
  pour chaque sprite
     sprite.dessin(contexte)
  fpour
}
```

```
metAJour() { // Vue
  pour chaque sprite
     sprite.mettreAJour()
  fpour
}
```

```
estEnMouvement() { // Vue
  Vrai si au moins un sprite de la vue est en mouvement
}
```

Attention, c'est pour toute la vue
Il suffit qu'un sprite soit en mouvement
Pour que la vue soit en mouvement !

Candy Crush

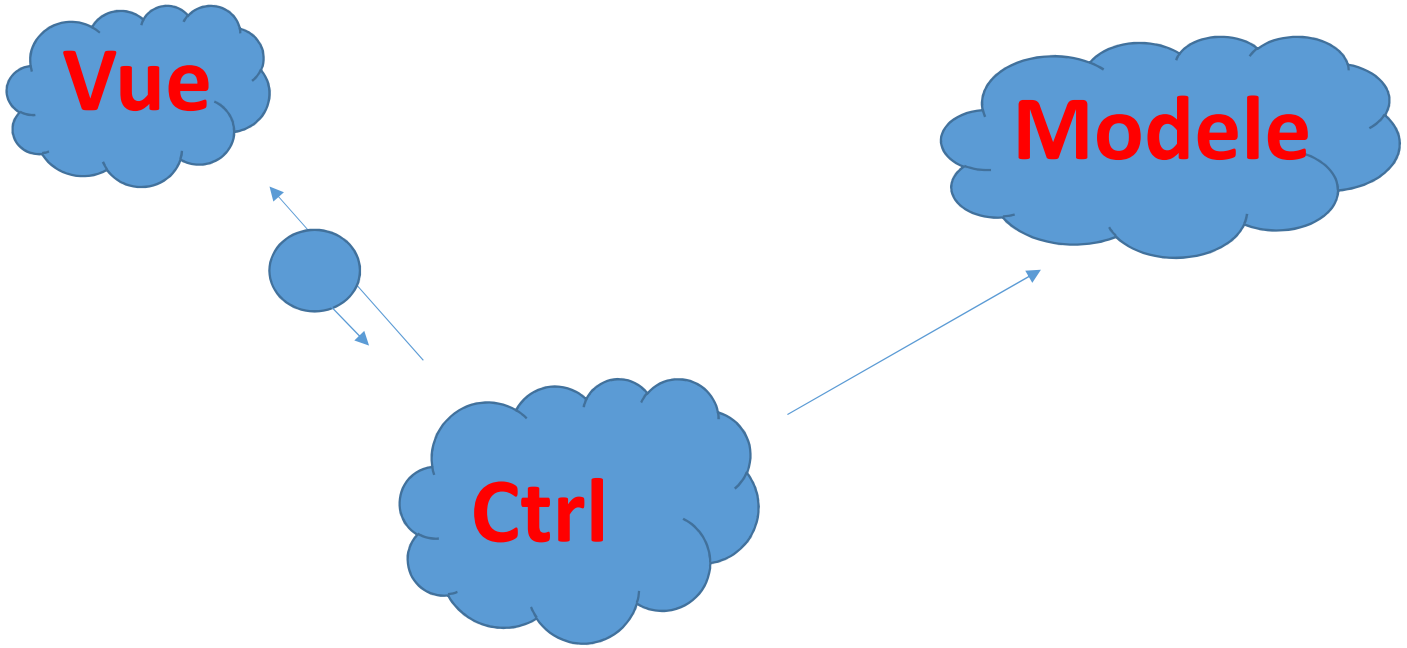
```
var vue = new Vue( …… )
// on rajoute des sprites à la vue
// ….
// on demande des déplacements à faire à certains sprites
animeVue()

function animeVue(contexte) {
        vue.metAJour() // itère sur la mise à vue de chaque Sprite
        vue.dessinTout(contexte)
        if (vue.estEnMouvement())
                setTimeout(animeVue, 10)
}
```

Faire la différence entre chaque Sprite et la Vue qui contient Tous les sprites

PASSER EN OBJET !

Candy Crush



Candy Crush

```
var vue = new Vue( …… )
// on rajoute des sprites à la vue
// ….
// on demande des déplacements à faire à certains sprites
vue.animeVue()

class Vue ….. {

  function animeVue(contexte) { // Méthode de Vue
        this.metAJour() // itere sur la mise à vue de chaque Sprite
        this.dessinTout(contexte)
        if (this.estEnMouvement())
                    setTimeout(this.animeVue, 10)
    }
}
```

IMPOSSIBLE

Candy Crush

```
var vue = new Vue( …… )
// on rajoute des sprites à la vue
// ….
// on demande des déplacements à faire à certains sprites
vue.animeVue()
// class Vue……. {
// méthode de Vue
function animeVue(contexte) {
        this.metAJour() // itere sur la mise à vue de chaque Sprite
        this.dessin(contexte)
        var that = this
        if (this.estEnMouvement())
                setTimeout( () => { that.animeVue(contexte) }, 10) }
 }
// } //
```

Candy Crush

Vue : objet qui sert à afficher les éléments de l'écran

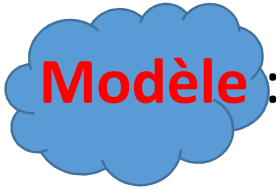
La vue ne contient QUE des objets graphiques et leur position

La vue ne contient rien du problème à résoudre

Son seul problème à résoudre c'est de gérer les objets graphiques

ET leurs déplacements.

Candy Crush



Modèle : objet qui sert à contenir les données du problème

Le modèle ne s'occupe pas de l'affichage (c'est la vue qui gère).

Le modèle conserve en mémoire les données du problème à un moment donné.

Ces données sont modifiées par le contrôleur.

Candy Crush et MVC

- Modèle :
  - État de la grille de bonbon
  - Indépendant de l'affichage
    - Donc indépendant des animations (qui ne concernent que la vue)
  - 5 types de bonbons
  - Le Score (stocké, pas d'affichage)

- Modèle : pensez à la sauvegarde / save / reload
  - Qu'est-ce qu'on met dedans ?
  - Ce qu'on y met, c'est le modèle
  - Donc pas l'animation etc..

Candy Crush et MVC

- Vue :
  - A minima, la vue doit pouvoir afficher le modèle
  - Mais elle peut faire plus que ça ! (Ctrl ?)
  - Animation secondaires, décorations, thème
  - Sons, bruitages

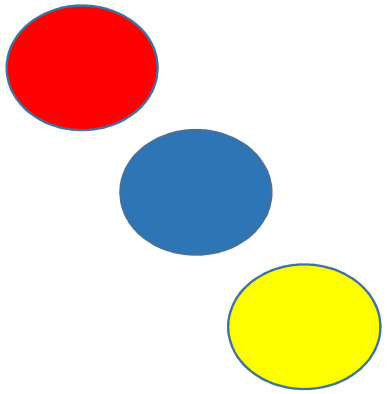
  - La vue contient clairement les *sprites* bonbon
  - Dans le cas de ce problème, un tableau de *sprites*
  - Pourquoi ne pas utiliser le modèle et éviter de refaire un tableau de *sprites* ?
    - Parce que l'animation a besoin d'informations non cruciales en plus
    - Que chaque bonbon peut être animé indépendamment
    - Parce que l'animation et le modèle peuvent ne pas être synchronisés

Candy Crush et MVC

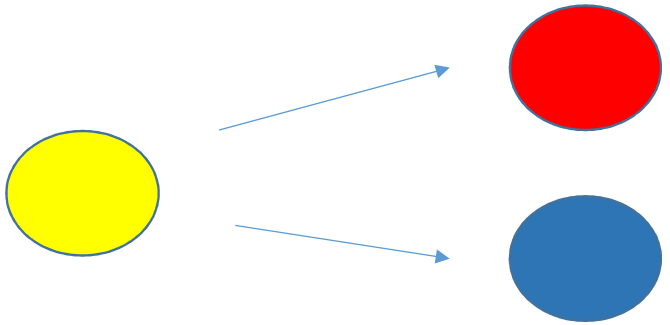
- Contrôleur :
  - Le chef d'orchestre des évènements
  - Il se déclenche sur des événements
  - Click souris : évènement évident
  - Le click souris va déclencher la réaction du modèle et de la vue
  - Il faut interdire les clicks souris pendant les phases
    - De stabilisation de la grille
    - Et donc d'animation
  - Donc les évènements à prendre en compte
    - Click + fin d'animation

Candy Crush et MVC

- En théorie
  - 3 objets
    - Un modèle
    - Une vue
    - Un contrôleur

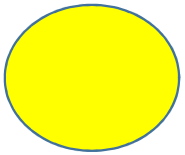


  - Ici le contrôleur est le chef d'orchestre
    - Initialisation : une instance de contrôleur
    - Le contrôleur créé un modèle et une vue

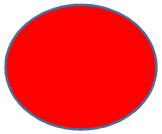


Évènement

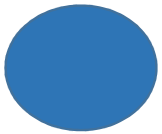
Déclenchement d’un évènement



Contrôleur



Modèle : contient un tableau d’objets qui représentent l’état de la grille + score



Vue : contient un tableau d’objets graphiques pour l’affichage

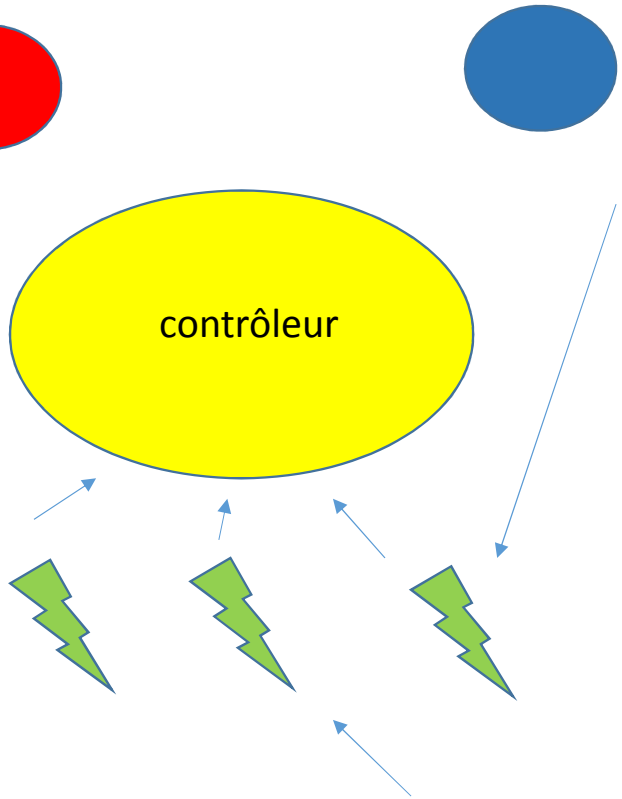
Le contrôleur répond à des évènements (ou déclenche des evt)
En réponse, il demande au modèle de se modifier
Et à la vue de se mettre à jour

Création du contrôleur

Fin animations
La vue a terminé ses animations
Elle représente bien le modèle

Click souris

On suppose que le jeu n'a pas de fin
(sinon, rajouter un evt fin / temps, score, nb bonbons etc..)



Modèle

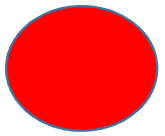


Tableau d'entiers ou « » si vide
Entier pour le score
Création : grille au hasard (sans trous)

Vue

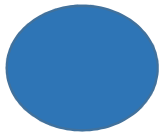
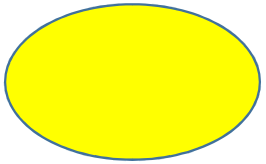


Tableau de lutins (vide au départ)
Mission : affiche les lutins et le score

La vue demande au modèle son contenu
Crée des lutins qui correspondent aux
Codes bonbon (1 -> rouge, 2 -> vert..)

new

Accès lecture



new

Création du contrôleur (méthode)

vue.metAJourAPartirModele()

1. Création du modèle

puis 2. Création de la vue

puis 3. Demande à la vue de se mettre à jour / modèle

puis 4. Lance l'animation de la vue

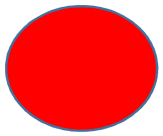
Cf animation
vue.animeVue()

setTimeout…

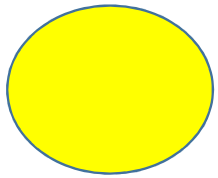
Evt fin animation

STOP

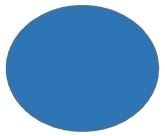
Modèle



Contrôleur



Vue



Attention gestion score :
bloquer jq pas trous + filtrage click

Fin animation

la vue est synchronisée sur le modèle (rien ne bouge plus)
on demande au modèle **s'il y a des alignements** ?

modele.explosionPossible() ?

OUI ! On demande au modèle de les faire exploser
puis on demande à la vue de se mettre à jour **puis** animation
STOP

vue.metAJourAPartirModele()

vue.animeVue()

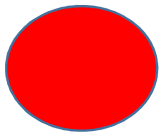
NON ! Le contrôleur appelle une méthode **repackGrille** qui
regarde et corrige dans le modèle les trous en comblant les colonnes verticalement : à chaque trou comblé, il demande à la vue de créer un lutin correspondant dont la cible doit être la position finale.
puis animation de la vue (pas de mise à jour ici, c'est le contrôleur qui)
STOP

Pas de trous
Jeux + score

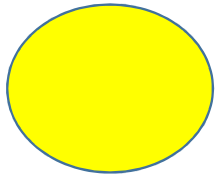
vue.animeVue()

accès

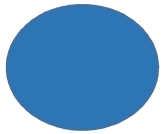
Modèle



Contrôleur



Vue



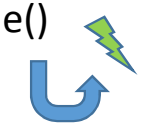
Click souris (var) pas d'evt tant que lancement pas terminé

(var) On détecte le premier / second click

vue.marqueLutin(x,y)

Premier click : on note la case sélectionnée **puis** on indique à la vue que le lutin de cette case est marqué **puis** on demande l'animation de la vue

vue.anime Vue()



STOP

Second click : on prend seulement si différent case courante, voisin ?

echangeCases(x1,y1,x2,y2)

On demande au modèle d'échanger les 2 cases (on ne touche pas à la vue)

explosionPossible() ?

Puis On demande au modèle si explosion possible ?

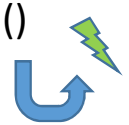
vue.enleveMarqueLutin(x,y)

NON : on demande du modèle de rééchanger les deux cases (on remet dans l'ordre)

vue.anime Vue()

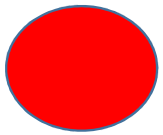
Puis déselection dans la vue **puis** affichage (ou animation)

vue.afficheVue()

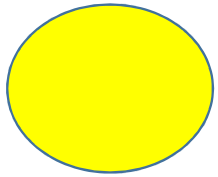


STOP

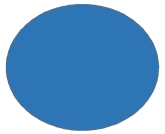
Modèle



Contrôleur



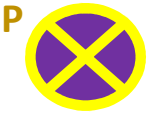
Vue



Click souris (var) pas d'evt tant que lancement pas terminé

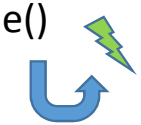
(suite)
OUI : indiquer l'échange des deux lutins à la vue **puis** animation de la vue

STOP



Vue.echangeLutins(x1,y1,x2,y2)

vue.animeVue()



Description

-> fonctions utilitaires

-> définition de VUE, MODELE, CONTROLEUR

-> Evt et gestion

-> partie GL

diagramme de classe

définition des méthodes (sémantique)

etc..

Organisation objet

```
class LutinBonbon  { //

        constructor(quel, largeur, hauteur ) {
                // quel : type de l'objet (1, 2, 3, 4, 5) - en gros un tag sur le bonbon
                // largeur, hauteur : zoom à l'affichage
        }

        positionXY(x1, y1) { // change la position
        }

        typeBonbon() { // renvoie le type du sprite
        }

        marque(v) { // true ou false : indique si l'objet est sélectionné ou pas
        }

        dessin(context) { // dessine le sprite avec sa position courante
        }
```

Organisation objet

```
        deplacementVers(x2, y2) { // prévoit un déplacement vers x2,y2
        }

        estEnMouvement() { // renvoie vrai si le sprite bouge encore
        }

        metAJour() { // met à jour les coordonnées du sprite si besoin
        }
} // class LutinBonbon
```

Organisation objet

```
class Vue { // contient le modèle de la vue

        constructor(txy, monControleur, modele, tailleLutin) { // on passe la taille et un objet contrôleur en argument
        }

        metAJourAPartirDuModele() { // la vue va aller chercher son état dans son propre modèle
        }

        nouveauLutin(x,y, quel) {
        }

        echange2lutins(x1,y1,x2,y2) { // intervertit deux cases dans la vue
        }

        afficheVue(contexte) { // dessine la vue sans animation
        }

        animeVue(contexte) { // anime les bonbons et quand c'est terminé, appelle le contrôleur
        }
}
```

Organisation objet

```
class Modele { //

        constructor(taille) { // taille est le coté du carré en nombre de cases
        }

        echange2cases(x1,y1,x2,y2) { // deux cases dans le modèle
        }

        faitExplosion() { // on fait les explosions du modèle
        }

        explosePossible() { // est-ce qu'il y a des explosions potentielles dans le modèle ?
        } // explose

}
```

Organisation objet

```
class Controleur { //

        constructor(tailleJeu, tailleLutin) {
        }


        finAnimation(contexte) { // est appelé automatiquement quand la vue s'est mise à jour avec l'animation
                // Quand la vue s'est stabilisée, ca veut dire que l'animation en cours
                // est terminée et que la vue reflète bien le modèle
                // à ce stade, il faut pour le contrôleur, observer le modèle et le modifier
                // et éventuellement relancer une animation de la vue

        }


        click(x,y) { // on vient de cliquer à la position x,y dans l'écran

        } // click(x,y)
```

Organisation objet

```
    repackGrille(contexte) { // fait tomber et rebouche les trous en créant de nouveaux bonbons
    }


    repackColonne(col) { // repack une colonne: donne faux si pas besoin
    } // repackColonne

}
```

```
<!DOCTYPE html>
<html lang="fr">
	<head>
		<meta charset="UTF-8" />
		<script src="mscript.js"></script>
		<title>test</title>
	</head>
	<body onload="init()">
		<canvas id="cvs" width="500" height="500" style="margin: 10px auto; border: solid 1px #000;"></canvas>
		<img id="button" onclick="init()">
	</body>
</html>
```

```
function captureClick(event) // on intercepte le click souris
{ // calcul des coordonnées de la souris dans le canvas
    if (event.target.id == "cvs") {
        var x = event.pageX - event.target.offsetLeft;
        var y = event.pageY - event.target.offsetTop;
         jeu.click(x,y)
    }
}

function init() { // démarrage du jeu
   // variable globale
    context = document.getElementById("cvs").getContext("2d");
    context.width = document.getElementById("cvs").width;
    context.height = document.getElementById("cvs").height;

    document.addEventListener("click", captureClick);

    var jeu = new Controleur(10, 50) // cree une grille de 10x10 avec des lutins de taille 50x50
                // le controleur cree le modele et la vue

    jeu.maVue.metAJourAPartirDuModele(); jeu.maVue.animeVue(context)

}
```