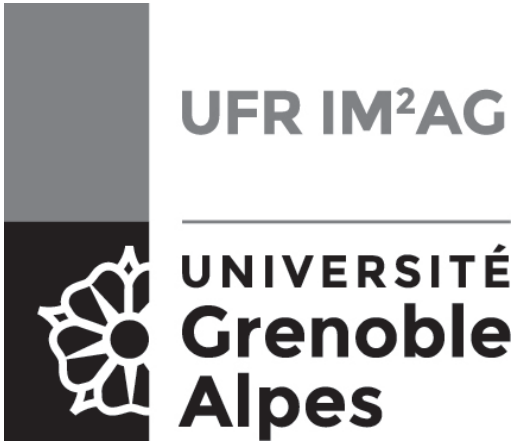


INF362
Interfaces graphiques et leur application en Java

Structure commune aux toolkits graphiques modernes

Construction de l'interface par assemblage de

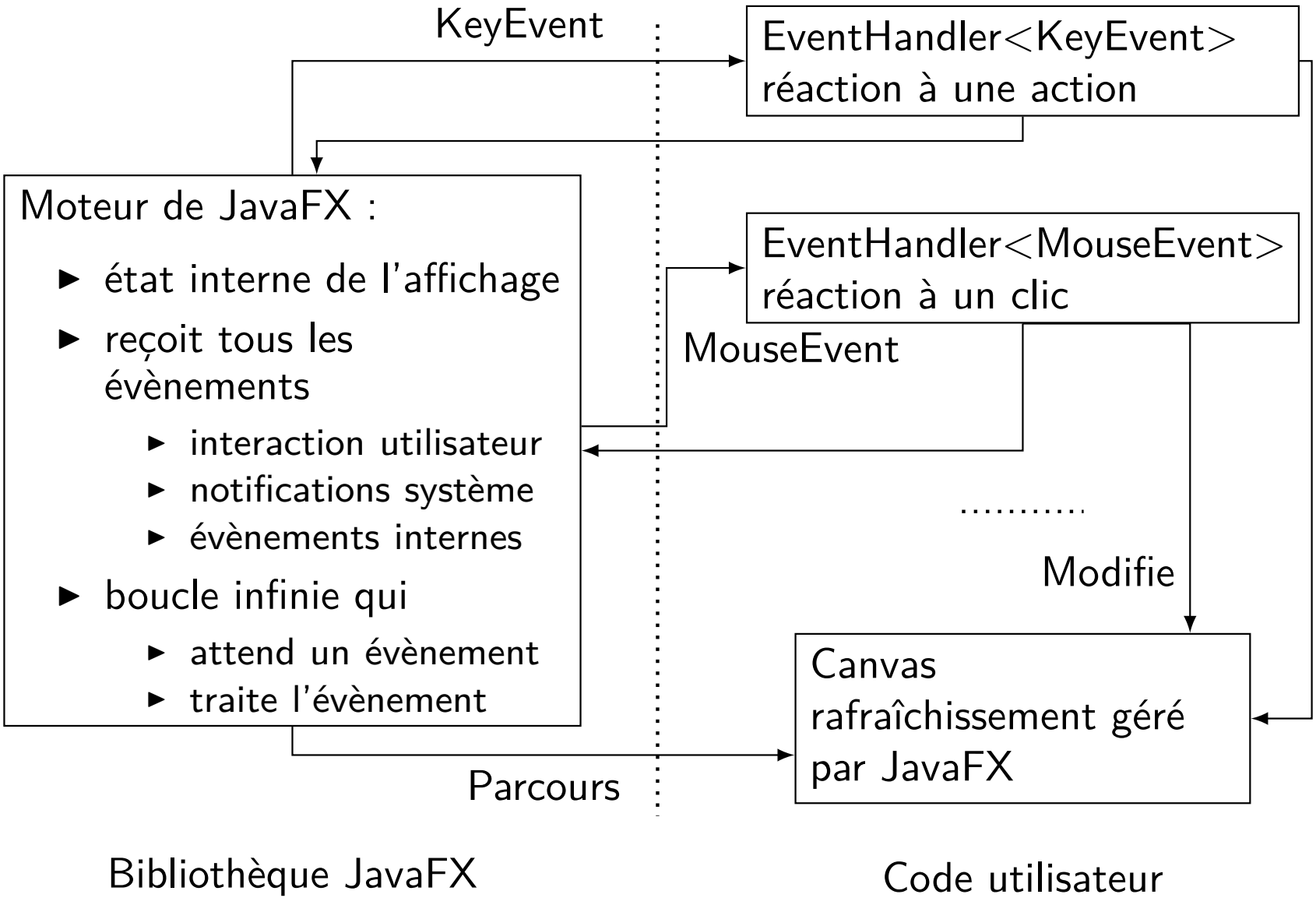
- composants graphiques organisés dans l'espace (conteneurs)
- fonctions de réaction exécutées lors de l'action sur les composants

Fonctionnement : modèle évènementiel

- boucle d'attente d'évènements
  - interactions utilisateur
  - notifications système
  - évènements internes

  généralement cachée dans la bibliothèque graphique
- gestion d'un évènement par appel de la fonction de réaction du ou des composant(s) concerné(s)

Moteur d'une interface graphique



Modèle évènementiel

Avantages

- simultanément à l'écoute de plusieurs sources (clavier, souris, ...)
- programmation plus simple et concise qu'une boucle de scrutation

Contraintes

- les fonctions de réaction doivent s'exécuter rapidement
  - pas de calcul lourd
  - pas de temporisation

  dans le cas contraire on gèle l'interface
- l'exécution est non séquentielle, difficile à comprendre
- il faut partager des objets et maintenir un état pour communiquer entre fonctions de réaction

Le dessin

Les étapes dans le dessin d'un composant sont masquées par JavaFX

- tout est mémorisé (objets graphiques, images des canvas, ...)
- un évènement interne déclenche le rafraîchissement

Modèle simple

- on peut modifier un composant partout dans le code
- déplacements, effets fournis par des méthodes des composants

mais difficultés cachées

- débogage difficile, les dessins n'apparaissent pas en direct
- images mémorisées potentiellement inefficace (ex : gros canvas)
- scènes dynamiques potentiellement difficiles à gérer
  (ex : ajouts/suppressions fréquents d'objets graphiques)

Les animations

Toute modification de l'affichage en fonction du temps est une animation

- déplacement d'un objet
- clignotement
- temporisation
- succession d'étapes

$\Rightarrow$ utile dans beaucoup d'application, jeux en particulier

L'affichage se fait de manière standard, donc

- il faut découper l'animation en étapes
- JavaFX répercute à l'écran une étape de l'animation lors du rafraîchissement (évènement interne)
- il faut un mécanisme pour provoquer, à intervalles réguliers, le passage à l'étape suivante de l'animation

Mise en place d'animations

JavaFX propose des classes toutes faites qui masquent le découpage en étapes et la progression dans l'animation

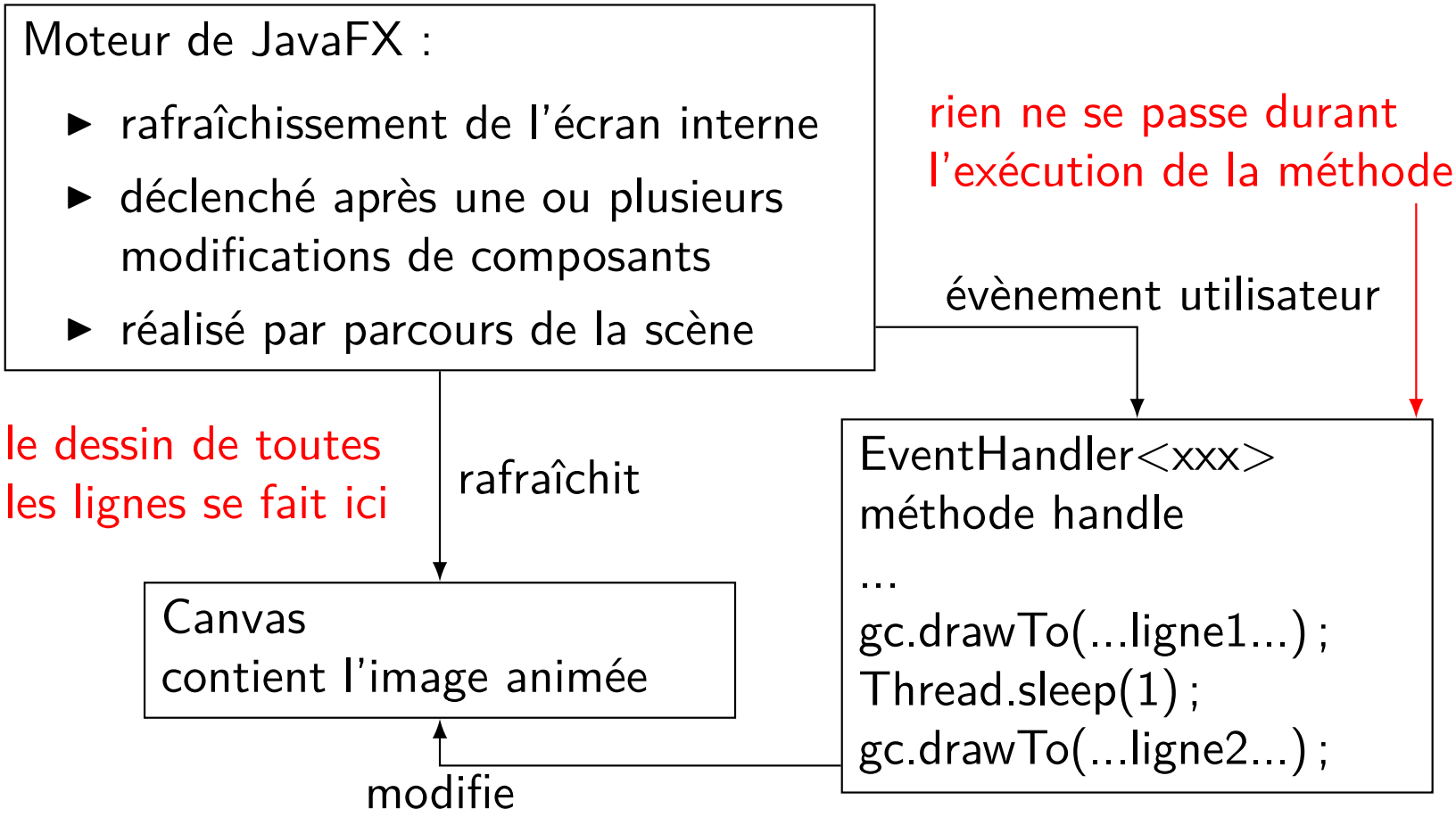
- `FadeTransition` : pour faire apparaître/disparaître un objet
- `TranslateTransition`, `PathTransition` : pour un déplacement
- `RotateTransition` : pour une rotation
- `ScaleTransition` : pour un zoom/dezoom
- ...

Composables en séquence (`SequentialTransition`) ou en parallèle (`ParallelTransition`)

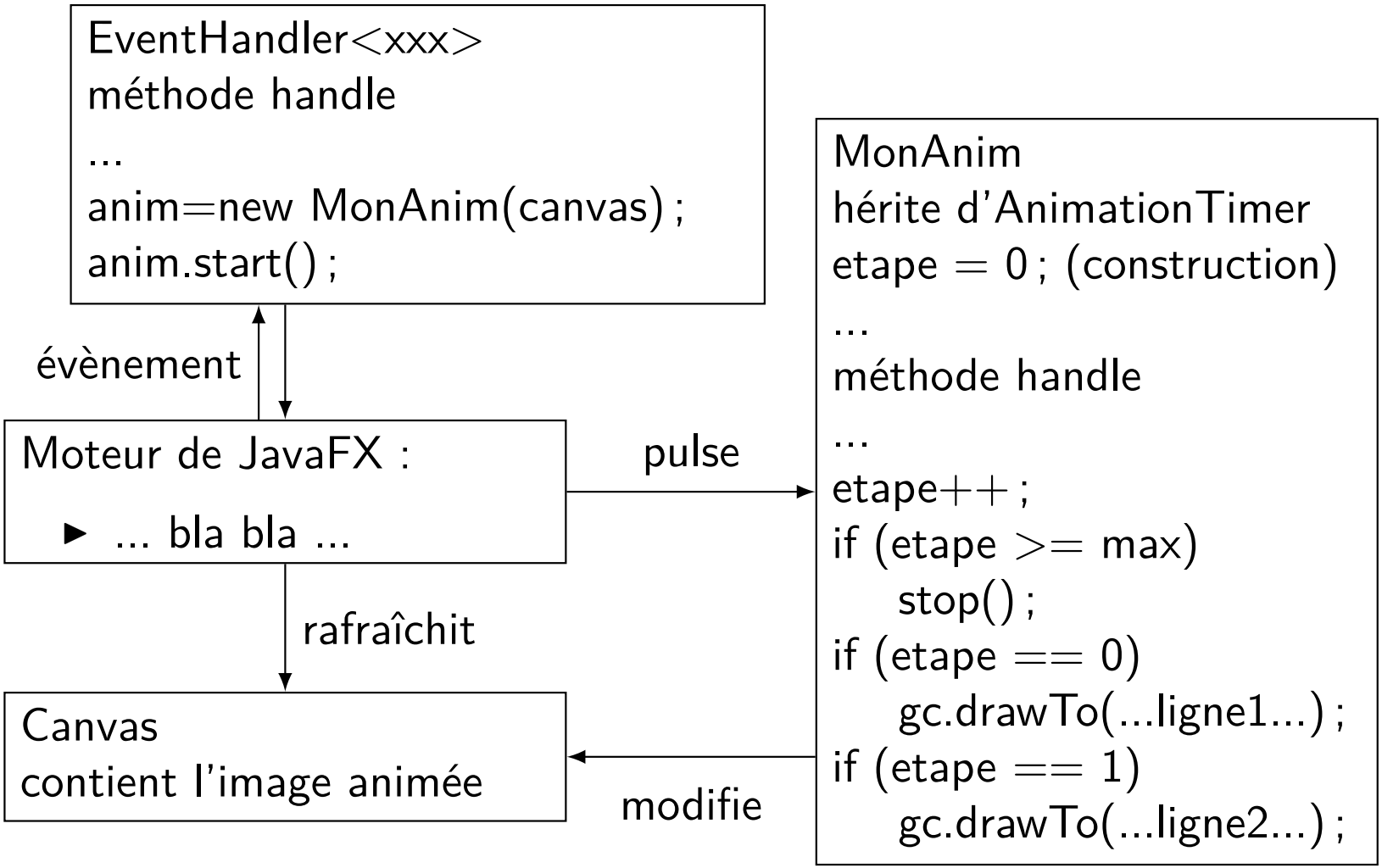
On peut aussi le faire à plus bas niveau, avec l'`AnimationTimer`

- appel de sa méthode `handle` toutes les 15ms
- avancée d'une étape de l'animation dans `handle`

Les animations, ce qu'on ne peut pas faire



Les animations



Gestion des animations

Les animations reposent donc sur

- une interaction (utilisateur) pour démarrer
- un `AnimationTimer` (ou un nouveau thread) pour déclencher chaque étape
- l'arrêt du timer (ou thread) à la fin de l'animation

Attention aux changements d'état de l'application durant une animation

- on ne souhaite pas forcément que celle-ci continue
- deux solutions
  - stopper l'animation avant le changement (un médiateur peut aider)
  - geler l'interface pendant l'animation

Exemple : dans un jeux de plateau, un pion se déplace

- clic sur *nouvelle partie* durant ce déplacement
- le déplacement ne doit pas se poursuivre sur le nouveau plateau

Structure d'une application graphique

Considérations générales

- découper l'application en modules
  - avancer par étapes
  - répartir le travail
- éviter les dépendances bidirectionnelles
  - pour tester les modules un par un
  - pour avancer de manière incrémentale

Considérations plus spécifiques aux interfaces

- multiples manières de représenter les données
- multiples sources d'entrée utilisateur

En plus de l'évolution technologique rapide

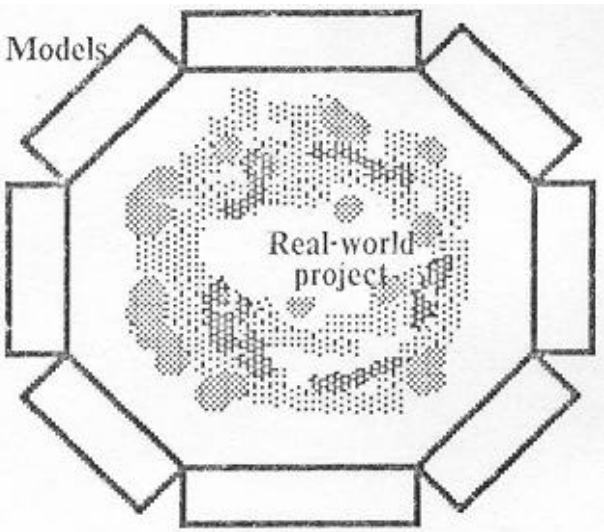
Des règles d'organisation et des *design patterns* peuvent aider

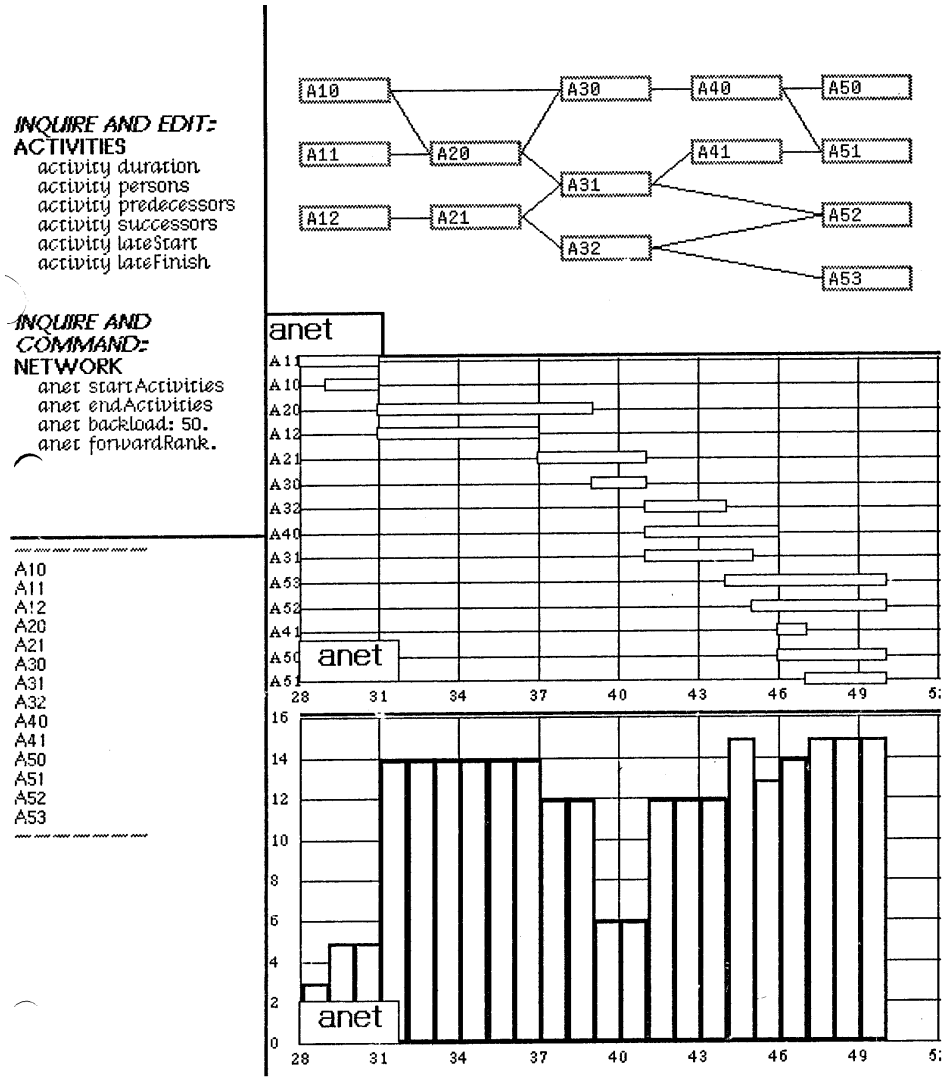
Organisation de l'application, modèle MVC

Trygve Reenskaug, 1979
d'abord Chose-Modèle-Vue-Editeur puis Modèle-Vue-Contrôleur

Objectif : modularité

- modèle(s) : ensemble de connaissances
- vue(s) : représentation graphique d'un modèle
- contrôleur(s) : lien entre l'utilisateur et le système





1979...

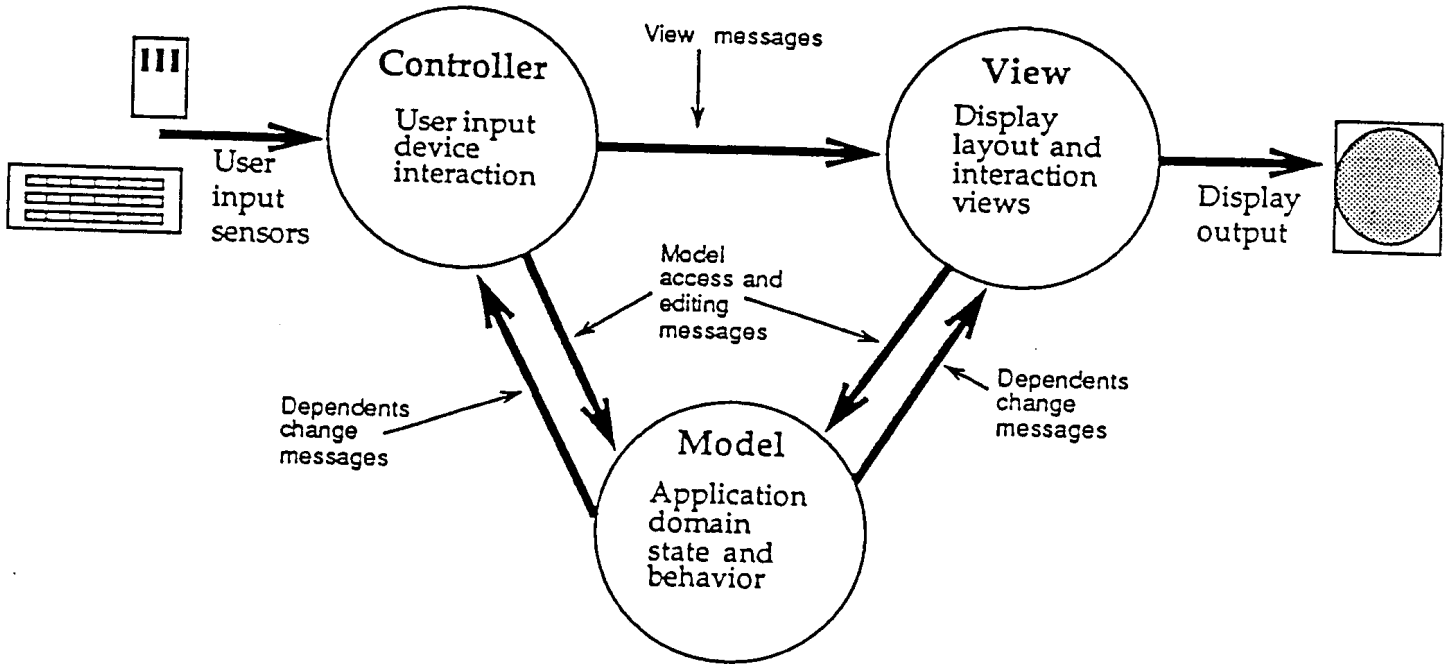
Apple II plus

Tandy TRS-80

Pas de souris (1981 pour les premières versions commerciales)
Pas d'internet (1983)

Organisation de l'application, modèle MVC

Glenn E. Krasner et Stephen T. Pope, 1988, reformulation du modèle de Reenskaug précisant, notamment, la communication entre composants



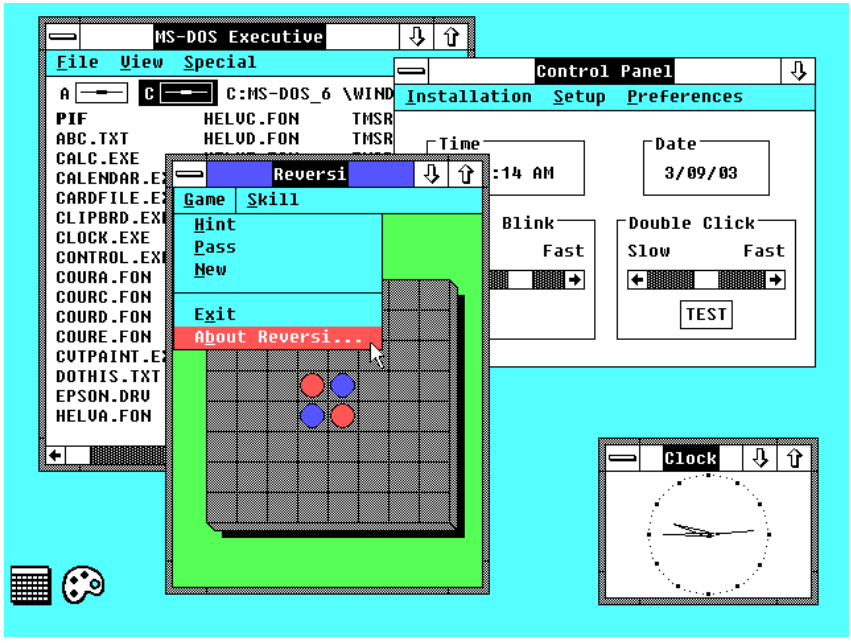
Relation asymétrique entre un modèle et ses vue(s)/contrôleur(s)

- les vue(s)/contrôleur(s) accèdent directement au modèle
- le modèle diffuse ses changements d'état à ses *dépendants*

Le même principe peut s'appliquer entre vue(s) et contrôleur(s)

... en 1988...

Acorn Archimedes



Microsoft windows 2

Pas de World Wide Web, http, html (1989)
Pas de Linux (1991)
Pas de Google (1998)

MVC aujourd'hui

Reste le principal modèle pour modulariser une interface graphique

- les vues et contrôleurs sont facilement remplaceables
- le travail peut être divisé plus facilement
- le design global est plus simple

MVC se fonde sur plusieurs *design patterns*

- adaptateurs
- observateurs

On en trouve une mise en œuvre dans la bibliothèque standard de java

L'implémentation de JavaFX y ajoute quelques éléments supplémentaires

- beaucoup de composants jouant le rôle de vue et/ou contrôleur sont également observables
- les propriétés permettent même d'être liées de manière bidirectionnelles à d'autres propriétés

Adaptateur

Convertit l'interface d'une classe en une interface attendue par son client

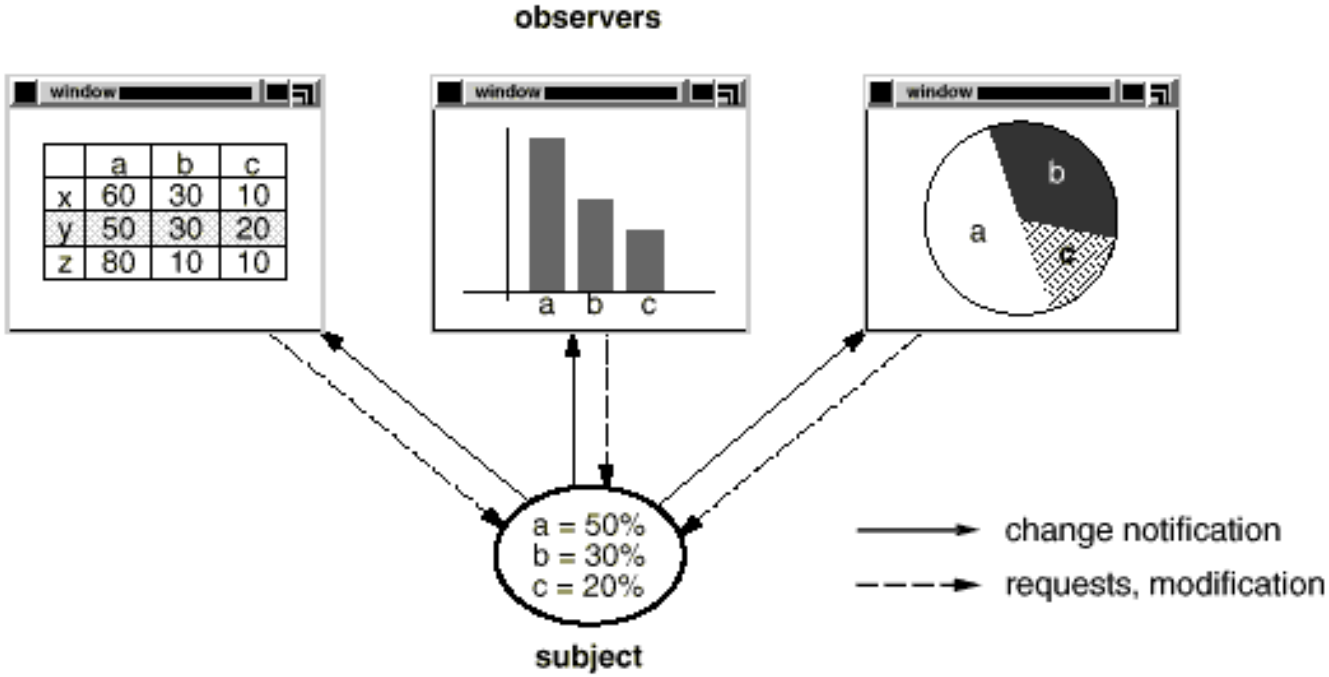
```
import javafx.event;

class AdaptateurNouveau implements EventHandler<ActionEvent> {
    Application app;

    public AdaptateurNouveau(Application b) { app = b; }

    public void handle(ActionEvent e) { app.erase(); }
}
```

Attachable à une entrée de menu ou de bouton via la méthode `setOnAction`

- nommage des méthodes globalement cohérent (`setOnKeyPressed`, `setOnMouseClicked`, ...)
- l'adaptateur étant court il est souvent défini de manière anonyme

Observateur

Aussi connu sous le nom *publish/subscribe*
Definit une relation de dépendance entre un ensemble d'objets observateurs et un objet observable, telle que l'objet observable peut prévenir tous ses dépendants d'un changement d'état le concernant.



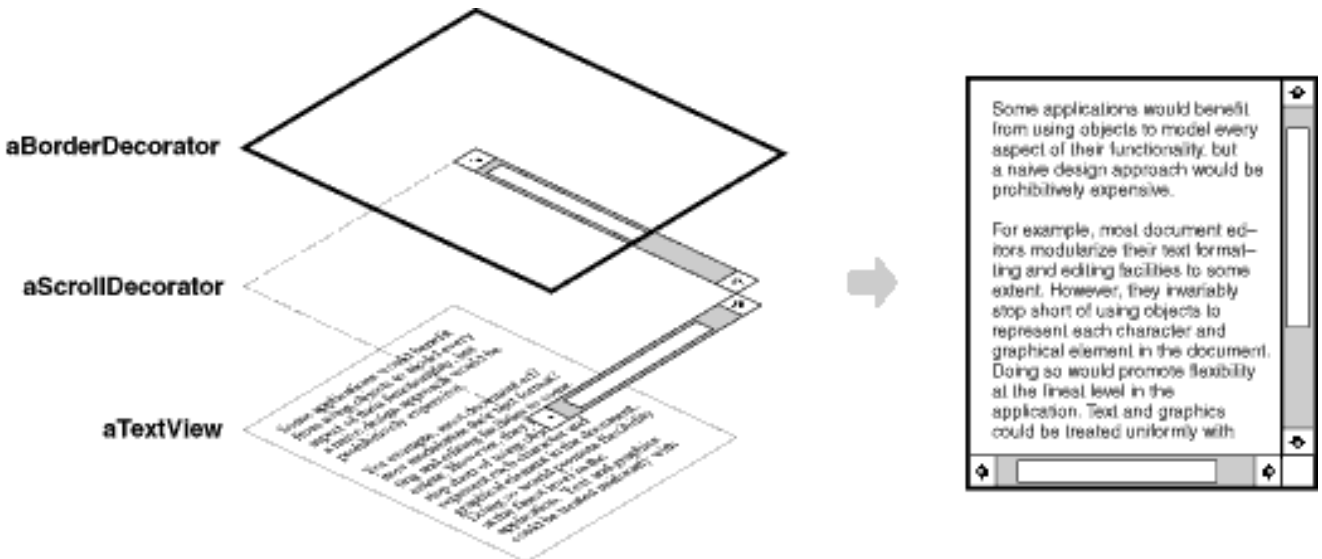
Classe `Observable` et interface `Observer` en java.

Observateur, un exemple d'implémentation

```
import java.util.*;

class Observable {
    java.util.List<Observer> changeListeners;

    public Observable() {
        changeListeners = new java.util.ArrayList<Observer>();
    }

    public void addObserver(Observer l) {
        changeListeners.add(l);
    }

    public void notifyListeners() {
        ListIterator<Observer> it;
        it = changeListeners.listIterator();
        while (it.hasNext()) {
            it.next().changeOccured();
        }
    }
}
```

Décorateur

Se retrouve assez fréquemment dans la bibliothèque standard de Java
Ajoute de nouvelles fonctionnalités à un objet existant en maintenant la même interface, une partie du travail étant faite par délégation.



Similaire à une spécialisation dynamique

- ajout/suppression de décorateurs à l'exécution sans toucher aux données sous-jacentes
- adaptation de l'application à l'environnement

Résumé sur les interfaces graphiques

Le code d'une application avec interface graphique est complexe

- volumineux
- peu factorisable
- avec beaucoup de dépendances entre objets

Quelques bonnes pratiques aident à avancer plus facilement

- respecter le modèle évènementiel
  - rafraîchir correctement
  - ne pas surcharger le système
  - ne pas geler l'interface
- utiliser le modèle MVC pour modulariser
- utiliser les autres design patterns
  - decorateur pour rendre l'interface dynamique
  - médiateur pour simplifier la gestion d'un ensemble de composants
  - ...

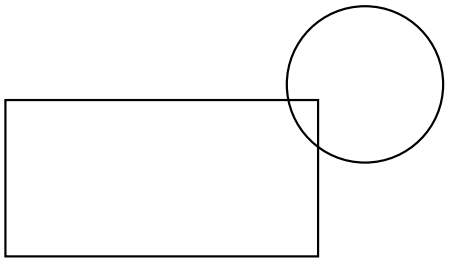
La gestion des collisions

Commune dans un jeu ou une simulation, mais complexe de

- détecter les collisions
- les traiter correctement
- raisonnablement efficacement

Déterminer si deux objets se touchent

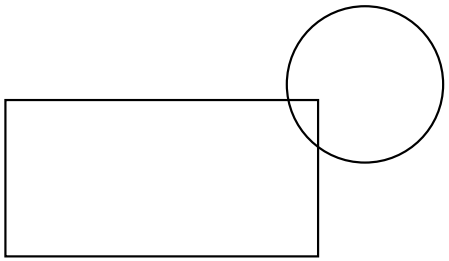
Simple : on détermine s'ils ont un ou plusieurs points en commun



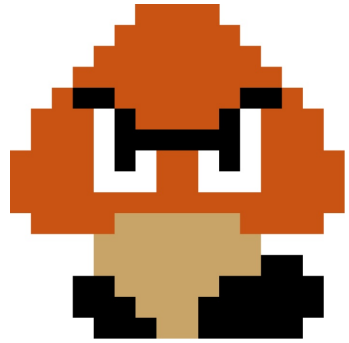
Facile

Déterminer si deux objets se touchent

Simple : on détermine s'ils ont un ou plusieurs points en commun



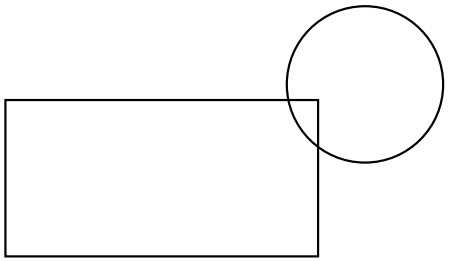
Facile



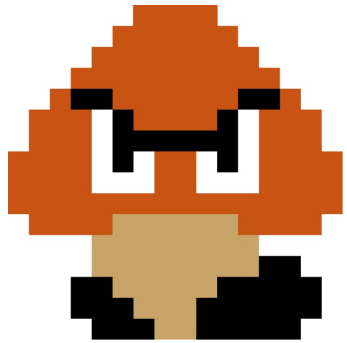
Difficile

Déterminer si deux objets se touchent

Simple : on détermine s'ils ont un ou plusieurs points en commun



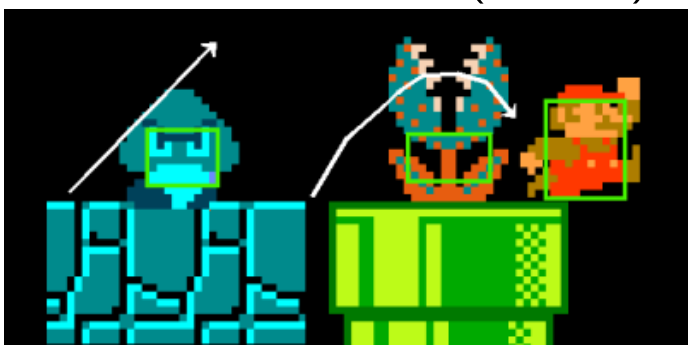
Facile



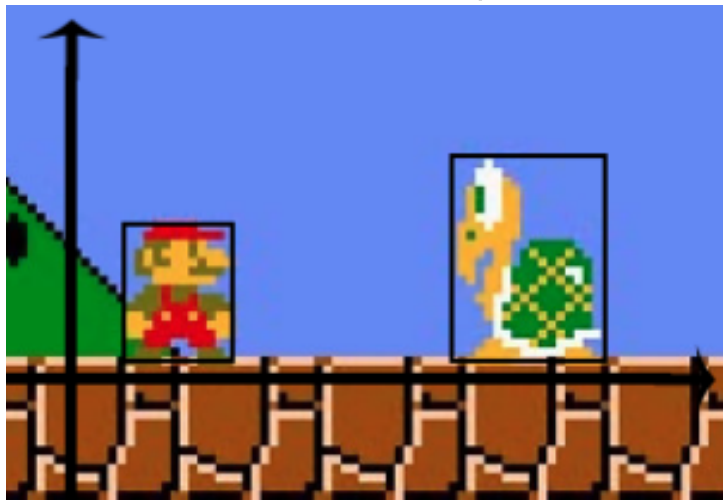
Difficile

Calcul général trop lourd, intersection d'unions de polyhèdres, on approxime avec des formes simples

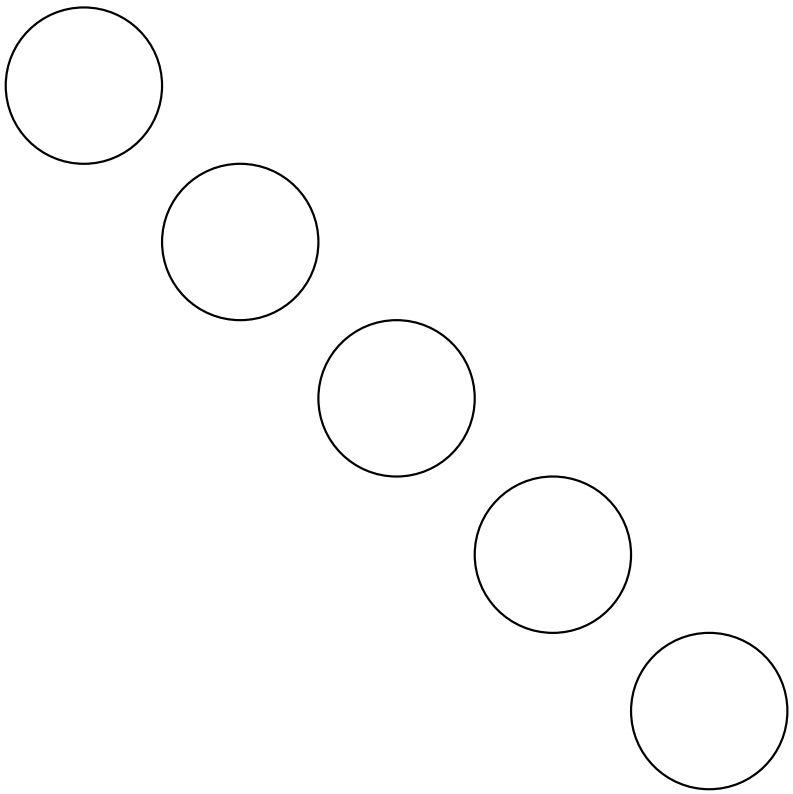
- la boite de collision (*hitbox*)



- la boite englobante (*bouding box*)



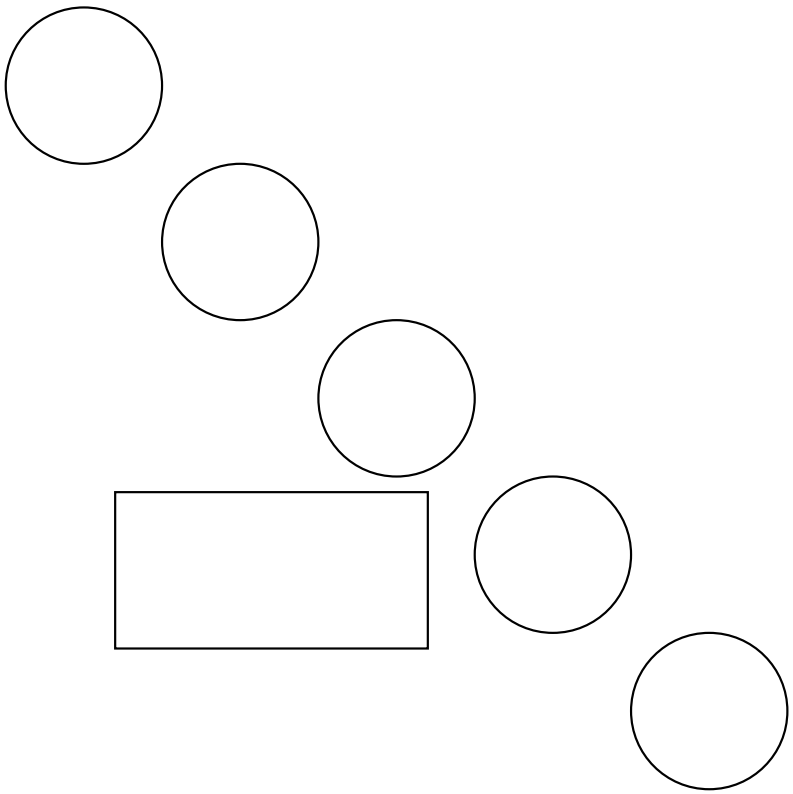
Detection d'une collision



Mouvement de la balle discrétisé

- guidé par les *pulses* et la vitesse

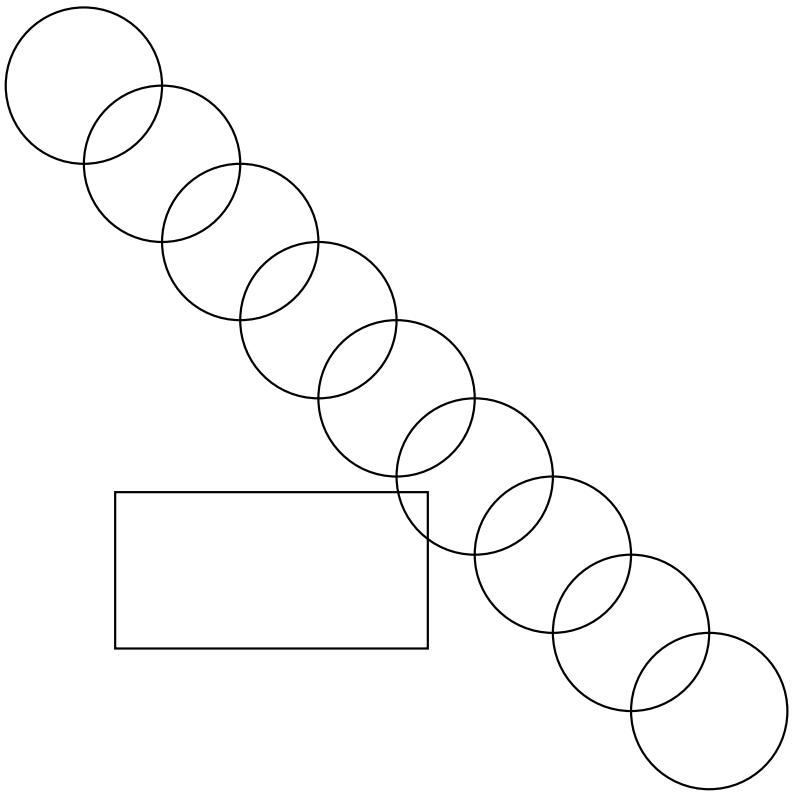
Detection d'une collision



Mouvement de la balle discrétisé

- guidé par les *pulses* et la vitesse
- les collisions sur les parties non parcourues ne sont pas vues

Detection d'une collision

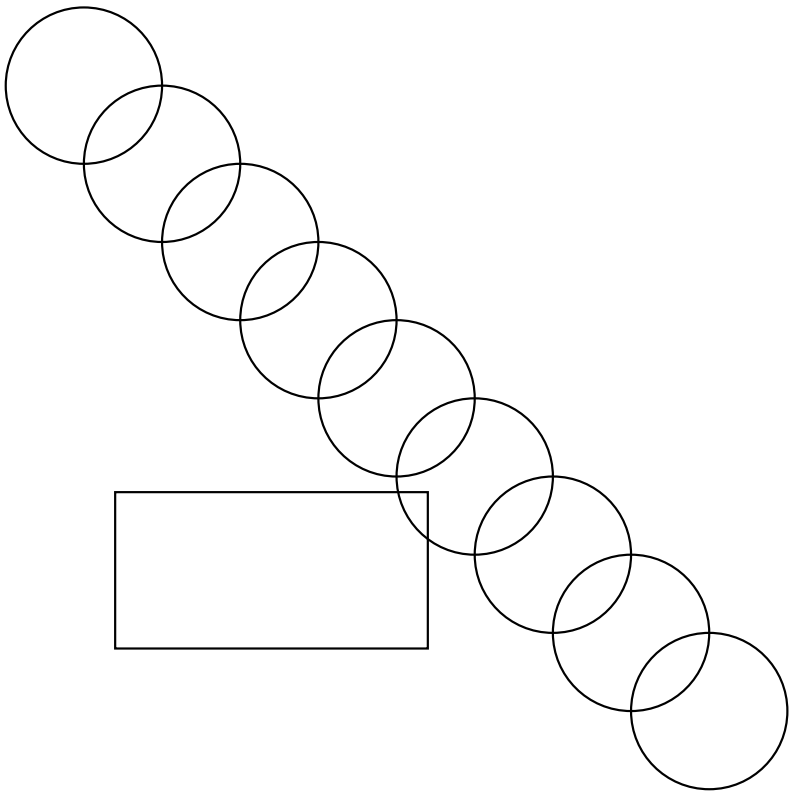


Mouvement de la balle discrétisé

- guidé par les *pulses* et la vitesse
- les collisions sur les parties non parcourues ne sont pas vues

On peut affiner les trajectoires

Detection d'une collision



Mouvement de la balle discrétisé

- guidé par les *pulses* et la vitesse
- les collisions sur les parties non parcourues ne sont pas vues

On peut affiner les trajectoires, mais

- FPS constant, on ralentit le jeu
- si on affine entre les *frames*, on alourdit le calcul

On ne détectera pas tout

Traitement d'une collision

Le résultat d'une collision dépend de

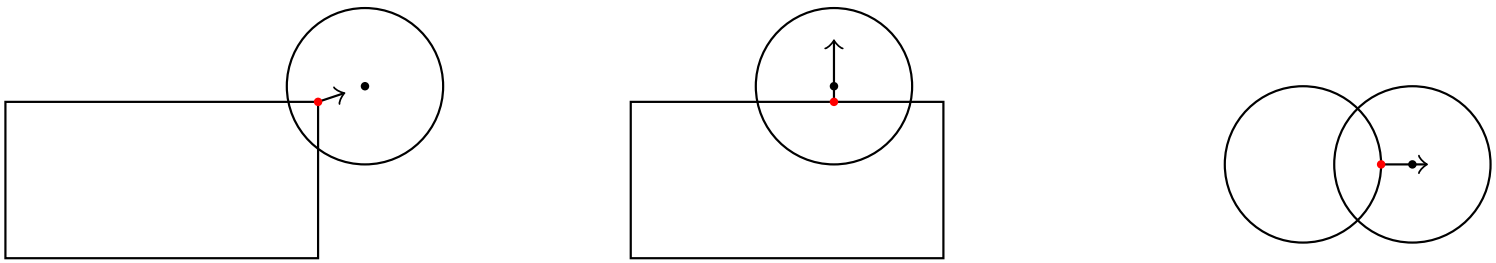
- la vitesse des objets qui s'entrechoquent
- de leur masse
- de l'élasticité de leur matériau
- de leur topologie
- ...

Ici, on simplifie

- vitesse constante
- pas de masse ou élasticité
- topologies simples : cercles, rectangles et demi-plans

Vecteur de réaction d'un objet percuté

Vecteur normal à la surface percutée par l'objet mouvant, avec un balle

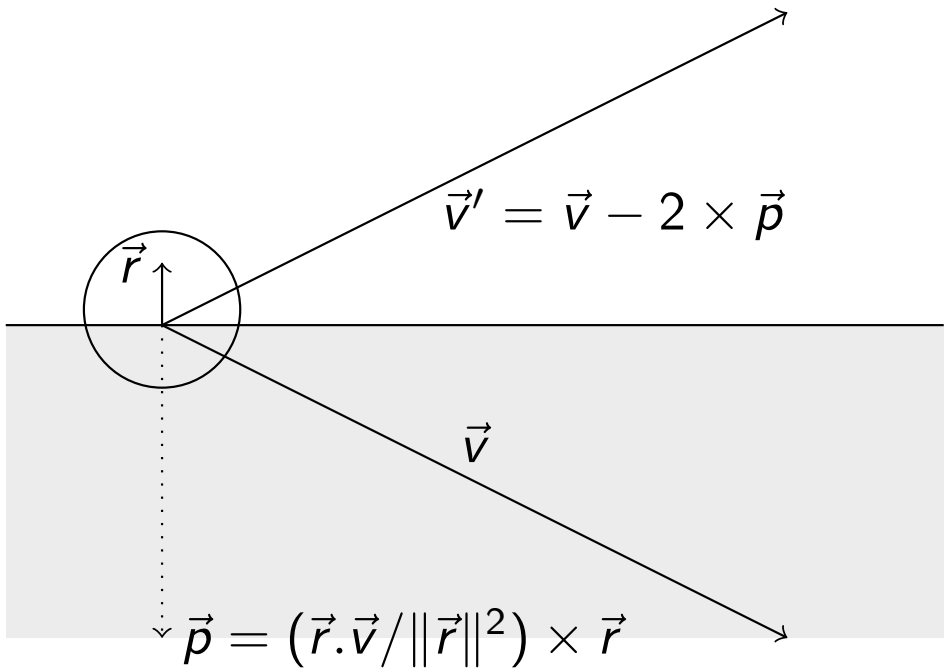


On veut

- déterminer l'importance de la collision, dans les trois cas :
  - rayon du cercle - distance entre son centre et le point d'impact
  - $\rightarrow$ valeur entre 0 et 0,5
- déterminer la direction la plus plausible :
  - vecteur entre le point d'impact et le centre du cercle

Prise en compte du vecteur de réaction

Pour avoir quelque chose de réaliste, nous voulons un rebond qui transforme la vitesse en son symétrique par rapport au plan d'impact

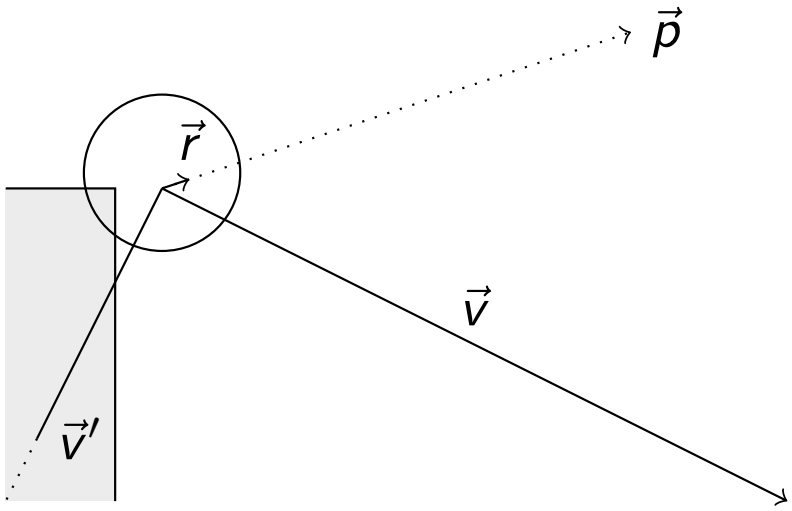


- on projette la vitesse sur l'axe passant par le point d'impact et colinéaire à la réaction
- on soustrait deux fois cette projection à la vitesse pour avoir le symétrique

On peut aussi déplacer d'office l'objet mouvant pour le sortir de l'objet avec lequel il entre en collision

Réaction anormale

Pour avoir quelque chose de réaliste, nous voulons un rebond qui transforme la vitesse en son symétrique par rapport au plan d'impact



- lorsque le produit scalaire $\vec{r}.\vec{v}$ est positif, la vitesse "éloigne" du point d'impact
- collision vue trop tard à cause de la discrétisation du déplacement

On peut tenter de corriger avec un schéma de résolution

- explicite : on affine la discrétisation
- implicite : on interpole avec la précédente position jusqu'à converger jusqu'au vrai point d'impact

ou on fait l'autruche et on ignore le problème.

Mise en place du traitement

Détermination des différentes vitesses

- pour avoir un jeu fluide en 60 FPS
- pour correspondre à une difficulté normale pour un humain

On ajuste ensuite les collisions

- on essaie avec les vitesses de base
- au besoin, multiplier les passes de mouvement et détection entre deux images
- on peut affiner les collisions détectées si elles sont anormales

éventuellement, on achète un processeur plus puissant...