Mondes Virtuels

Eric Maisel

Automne 2020

- Dépôt : `Réseau ENIB /home/TP/modules/maisel/MASTER`
- Contact : maisel@enib.fr
- Contenu :
  - CM : 2h
  - TP : 4h (restitution : fin Décembre)

Objectifs

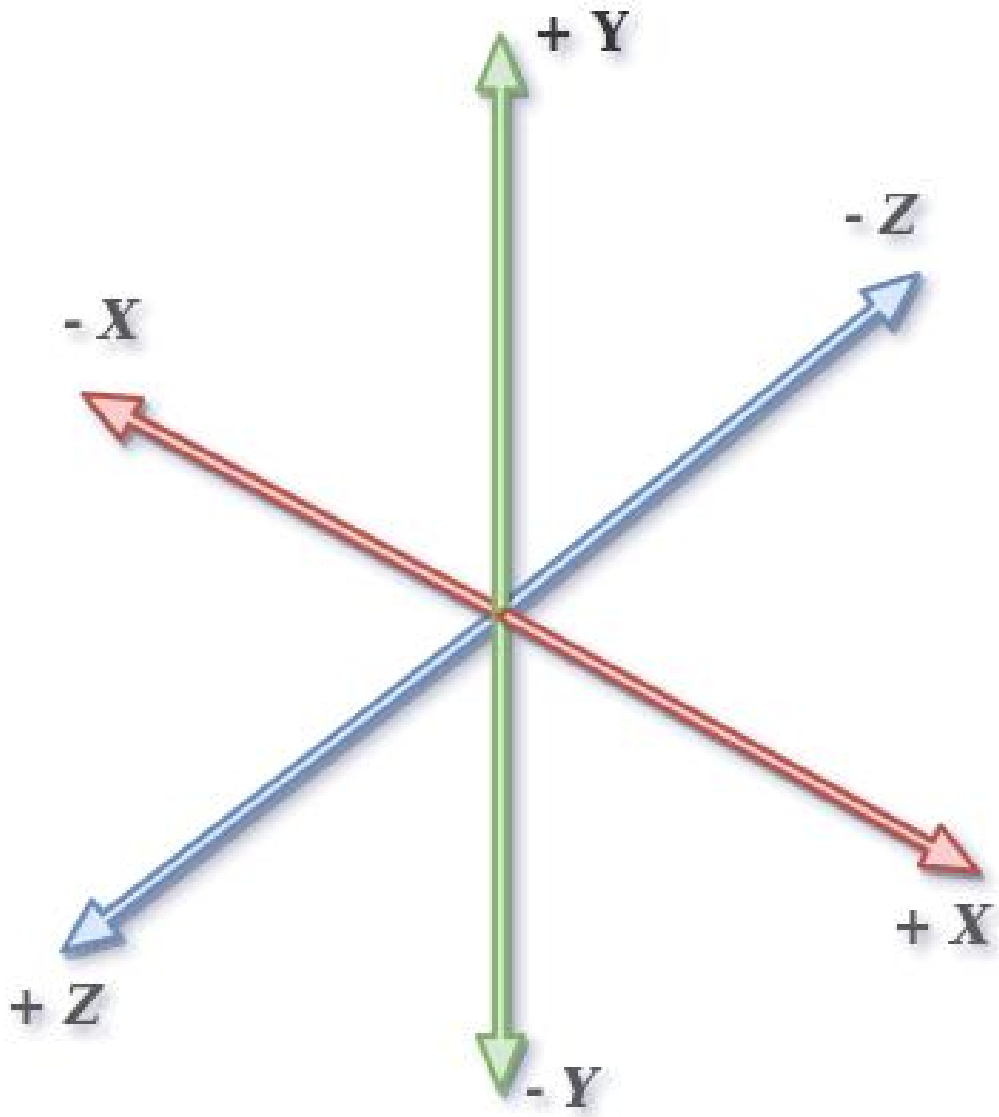
- Simulation d'un monde virtuel
- Le réel c'est quand on se cogne (Lacan)
- Le monde : une collection d'entités autonomes

Autonomie

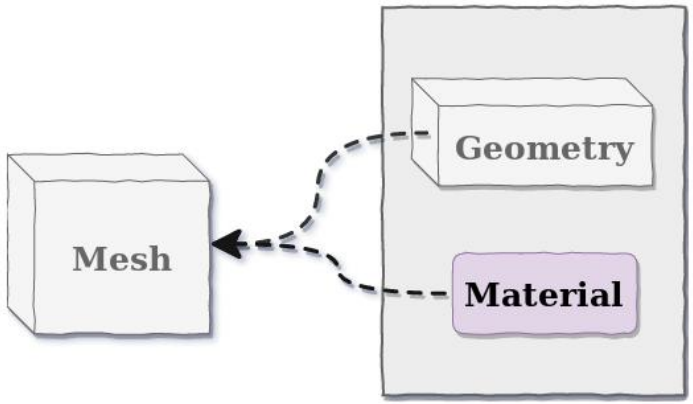
- Poursuivre un objectif en s'adaptant aux évolutions de son environnement
- $\Rightarrow$ percevoir son envt et agir en conséquence.

Les objets

Espace



Objets 3d



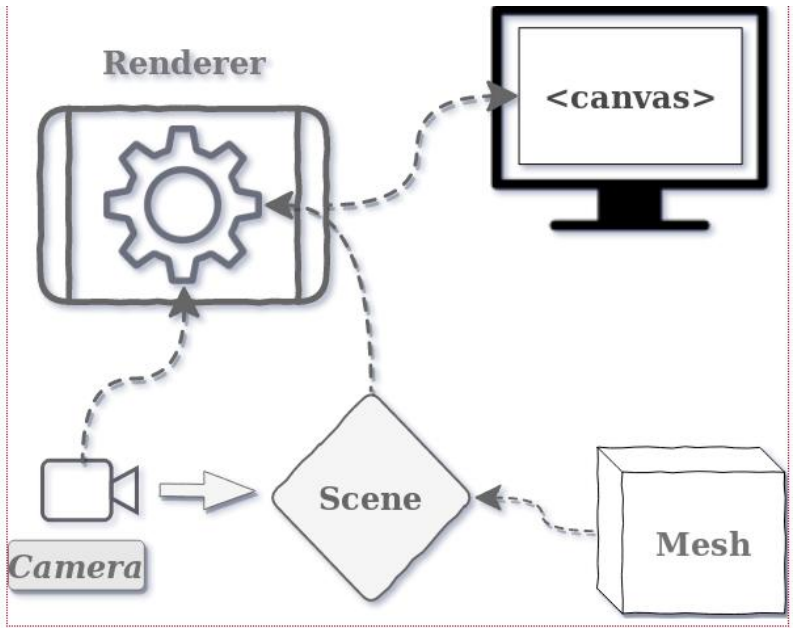
Description intrinsèque

- Forme : surface approchée par un maillage de triangles
- Matériau : paramètres décrivant les interactions lumière-matière (couleur, réflectance, ...)

Pose

- Translation
- Rotation

Calcul d'images



- **Scène** : collection d'objets 3d
- **Caméra** : position/orientation de l'utilisateur
- **Renderer** : procédure de calcul de l'image des objets de la scène depuis le point de vue spécifié par la caméra.

Le temps

Etat du monde

- Etat des objets du monde virtuel
  - Position, orientation
  - Aspect
  - Etat interne (vitesse, accélération, ..., des émotions, des intentions, ...)

Echantillonnage irrégulier

- Horloge : numéro de "frame"
- Horloge temps réel
  - Cohérence avec le temps physique
  - $t_i = t_{i-1} + \Delta_{i-1}$

Boucle de simulation

```
repeter
  traiter les événements utilisateur
  calculer le nouvel etat du monde, en afficher l'image
```

Organisation du code

Structuration en fichiers

- `index.html` : intégration du code dans une page Web
- `css/style.css` : fichier de définitions pour la mise en page
- `index.js` : script principal
- `assets` : répertoire des images et modèles 3d

Structuration en fonctions

- `init` : initialisation de l'infrastructure de restitution
- `creerMonde` : création des objets du monde
- `animer` : calcul de l'état du monde et d'une image

Exemple de code

Variables globales

```
var scene, objets3d, renderer, horloge, chrono ;
var camera, controleur ;
```

Initialisations

```
function init(){
  scene      = creerScene() ;
  objets3d   = {} ;
  camera     = creerCamera() ;
  controleur = creerControleurCamera(camera) ;
  chrono     = creerChronometre() ;
}
```

Exemple de code

Création des objets 3d

```
function creerMonde(){
  ...
  var geo = new THREE.BoxGeometry(1,1,1) ;
  var mat = new THREE.MeshStandardMaterial({color:'red'}) ;
  var mesh = new THREE.Mesh(geo,mat) ;
  mesh.rotation.set(Math.PI/2.0, Math.PI/3.0,0.0) ;
  mesh.position.set(5,2,5) ;
  scene.add(mesh) ;
  objets3d['cube-01'] = mesh ;
  ...
}
```

Exemple de code

Création des objets 3d

```
function animer(){
  requestAnimationFrame(animer) ;
  var dt = chrono.getDelta() ;
  horloge += dt ;
  controleur.update(dt) ;
  objets3d['cube-01'].rotation.set(0,t,0) ;
  renderer.render(scene,camera) ;

}
```

Entité-Composants-Systèmes

Entité

- une entité est un objet générique
- une entité est définie par :
  - un identifiant
  - une liste de composants

Composant

Des données pour définir :

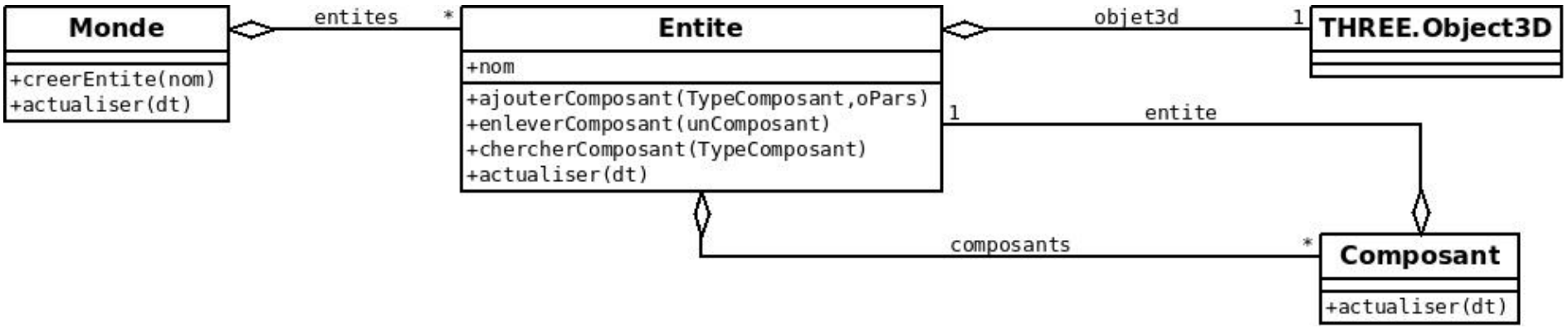
- un aspect d'une entité
- les interactions avec le monde

Système

Du code exécuté en continu

- qui réalise des actions globales sur les entités qui possèdent des les composants qui correspondent à ce système.

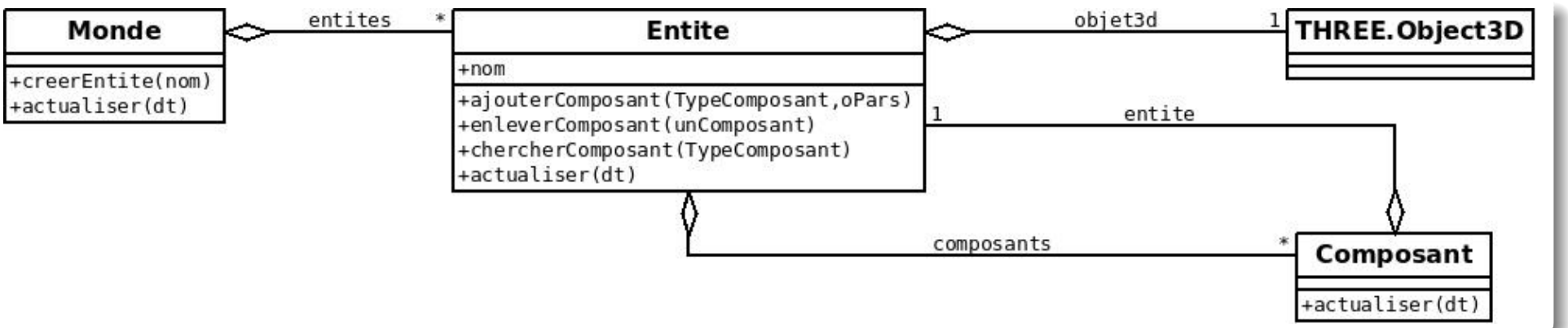
Organisation du code



- Abstraction des objets 3d par le type `Entite`
- Ajoût de fonctionnalités/comportements $\Rightarrow$ Ajoût de `Composant`
- Evolution de `Object3D` $\Rightarrow$ Exécution de `Entite.actualiser`

- Calcul des informations globales
- Calcul des états de chaque entité

Organisation du code




```
for entite in monde.entites :
  for composant in entite.composants :
    composant.actualiser(dt)

for entite in monde.entites :
  entite.actualiser(dt)
```

De la cinématique ...

Contrôle en position

$$P(t) = f(t)$$

Contrôle en vitesse

$$P(t + dt) = P(t) + \vec{v}(t)dt$$

- $\vec{v}(t) = cte \Rightarrow$ mvt rectiligne uniforme
- Comment faire varier $\vec{v}(t)$? i.e calculer $\Delta\vec{v}(t)$.

Contrôle en accélération

$$P(t + dt) = P(t) + \vec{v}(t)dt \quad (1)$$

$$\vec{v}(t + dt) = \vec{v}(t) + \vec{\gamma}(t)dt \quad (2)$$

De la cinématique ...

Une implémentation

Constructeur

```
function Entite(nom){
    this.nom          = nom ;
    this.objet3d      = null ;
    this.masse        = 1.0 ;
    this.vitesse      = new THREE.Vector3(0.0,0.0,0.0) ;
    this.acceleration = new THREE.Vector3(0.0,0.0,0.0) ;
}
```

De la cinématique ...

Une implémentation

Accès abstraits

```
Entite.prototype.setObjet3d = function(obj){
    this.objet3d = obj;
}

Entite.prototype.setPosition = function(x,y,z){
    if(this.objet3d) this.objet3d.position.set(x,y,z) ;
}
...
```

De la cinématique ...

Une implémentation

Méthode de mise à jour de l'état

```
Entite.prototype.actualiser = function(dt){
  if(this.objet3d){
    this.objet3d.position.addScaledVector(this.vitesse,dt) ;
    this.vitesse.addScaledVector(this.acceleration,dt) ;
  }
}
```

... à la dynamique

Problème

- Comment calculer $\vec{\gamma}$?
- Force : vecteur qui permet de modifier un mouvement
- Force : lien entre le mouvement d'un objet et son environnement.

Deuxième loi de Newton

$$\vec{F}(t) = m\vec{\gamma}(t)$$

... à la dynamique

Une implémentation

```
Entite.prototype.appliquerForce = function(f){
    this.acceleration.addScaledVector(this.acceleration,
                                      1.0/this.masse) ;
}

Entite.prototype.actualiser = function(dt){
    this.objet3d.position.addScaledVector(this.vitesse,dt) ;
    this.vitesse.addScaledVector(this.acceleration,dt) ;
    this.acceleration.set(0.0,0.0,0.0) ;
}
```

Implémentation de composants

Classe de base

```
function Composant(entite){
  this.entite = entite ;
}

Composant.prototype.actualiser = function(dt){}
```

Implémentation de composants

Exemple de sous-classe (1)

```
function CompAlea(entite, opts){
            Composant.call(this,entite) ;
            this.force = new THREE.Vector3(opts.force || 1.0,
                                           0.0, 0.0) ;
            this.alea  = opts.alea  || 0.5 ;
}

CompAlea.prototype = Object.create(Composant.prototype) ;
CompAlea.prototype.constructor = CompAlea ;

CompAlea.prototype.actualiser = function(dt){
    if(Math.random()<this.alea){
        this.entite.appliquerForce(this.force) ;
    }
}
```

Implémentation de composants

Exemple de sous-classe (2)

```
function CompFrott(entite, opts){
    Composant.call(this,entite) ;
    this.force = new THREE.Vector3(0.0,0.0,0.0) ;
    this.k  = opts.k  || 0.1 ;
}

CompFrott.prototype = Object.create(Composant.prototype) ;
CompFrott.prototype.constructor = CompFrott ;

CompFrott.prototype.actualiser = function(dt){
    this.force.copy(this.entite.vitesse) ;
    this.force.multiplyScalar(-this.k) ;
    this.entite.appliquerForce(this.force) ;
}
```

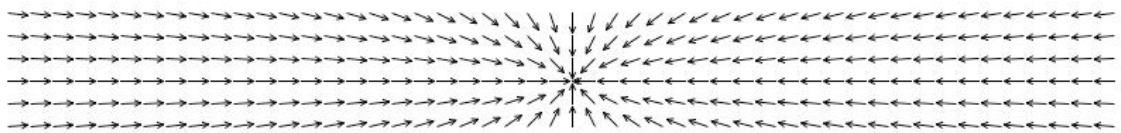
Exemple simple d'utilisation des entités et composants

```
function creerMonde(){
   ...
   var e = creerEntite("e1") ;
   e.setObjet3d(cube) ; // cube crée auparavant
   e.ajouterComposant(CompAlea,{}) ;
   e.ajouterComposant(CompFrott,{}) ;
   ...
}
```

Champs

Champs

Champs de forces



```
function CompChampForce2d(entite,opts){
  Composant.call(this,entite) ;
  this.champ = opts.champ ;
  this.f = new THREE.Vector3(0.0,0.0,0.0) ;
}
CompChampForce2d.prototype.actualiser = function(dt){
  if(this.entite.objet3d)
    this.f = this.champ.lookup(this.entite.objet3d.position);
  else this.f.set(0.0,0.0,0.0) ;
  this.entite.appliquerForce(this.f) ;
}
```

Champs

Champs de forces

Constructeur

```
function Champ2d(lx,lz,resolution){
  this.nbCols = lx / resolution ;
  this.nbLigs = lz / resolution ;
  this.t      = [] ;
}
```

Initialisation du champ

```
Champ2d.prototype.initAlea = function(){
  for(var j=0; j<this.nbLigs; j++)
    for(var i=0; i<this.nbCols; i++){
      var theta = 2*Math.PI*Math.random() ;
      var v = new THREE.Vector3(Math.cos(theta),
                                0,Math.sin(theta));
      this.t.push(v);}}
```

Accès aux éléments du champ

```
Champ2d.prototype.lookup = function(P){
  int c = int(clamp(P.x/this.resolution,0,this.nbCols-1)) ;
  int l = int(clamp(P.z/this.resolution,0,this.nbLigs-1)) ;
  return this.t[l*this.nbCols + c] ;
}
```

Champs

Champs de potentiel

- Φ : champs scalaire, potentiel
- $\nabla\Phi = \frac{\delta\Phi}{\delta x}\vec{i} + \frac{\delta\Phi}{\delta y}\vec{j} + \frac{\delta\Phi}{\delta z}\vec{k}$: gradient du champs
- $\vec{F}(P) = -k\nabla\Phi$

Environnement

- V : champ de potentiel
- γ : champ de résistance au mouvement
- σ : agitation aléatoire
- k : taux de croissance/dégradation
- D : taux de diffusion

Agents situés

- Masse
- Position
- Vitesse

Mouvement des agents

$$\frac{\delta \vec{v}(t)}{\delta t} = -\alpha \nabla \Phi(P) - \gamma(P)\vec{v}(t) + \sigma(P)$$

Evolution des champs

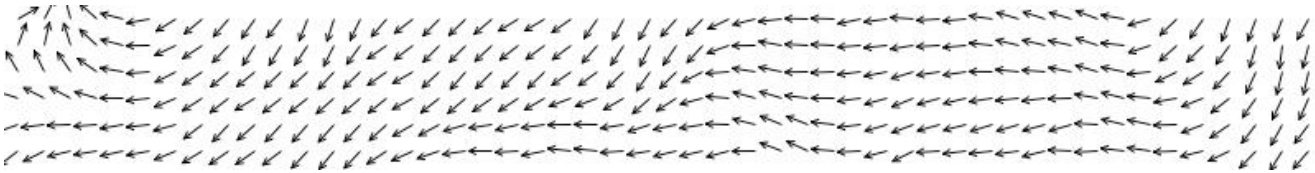
$$\frac{\delta \Phi(P)}{\delta t} = -k(P)\Phi(P) + \nabla(D(P)\nabla\Phi(P)) + \sum_i \lambda_i(\theta_i, \|P\vec{P}_i\|)$$

Evolution dans l'espace des états des agents

$$\frac{\delta \theta_i(t)}{\delta t} = \eta_i(\theta_i, t)$$

Champs

Champs de vitesse



- Champ $\vec{v}_d$: vitesse désirée (effet)
- $\vec{F}(t) = k(\vec{v}_d(P(t)) - \vec{v}(t))$

Accointances

Accointances

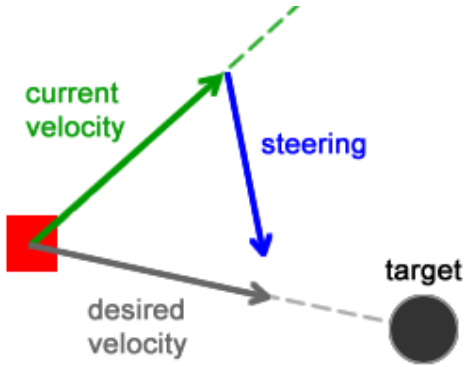
Accointance : "liaison avec d'autres personnes"

- Modification de l'état
- pour atteindre un objectif
- étant donnée un environnement décrit par des rôles

(envt, état) $\Rightarrow \vec{v}_d \Rightarrow \vec{F}$

Accointances

Passer par un point



- **Comportement** : `Seek`
- **Rôle(s)** :
  - Cible : C

$$\vec{F_s} = min(F_m, \|\vec{v_d} - \vec{v_c}\|) \frac{\vec{v_d} - \vec{v_c}}{\|\vec{v_d} - \vec{v_c}\|}$$

$$\vec{v_d} = v_m \frac{\vec{PC}}{\|\vec{PC}\|}$$

Accointances

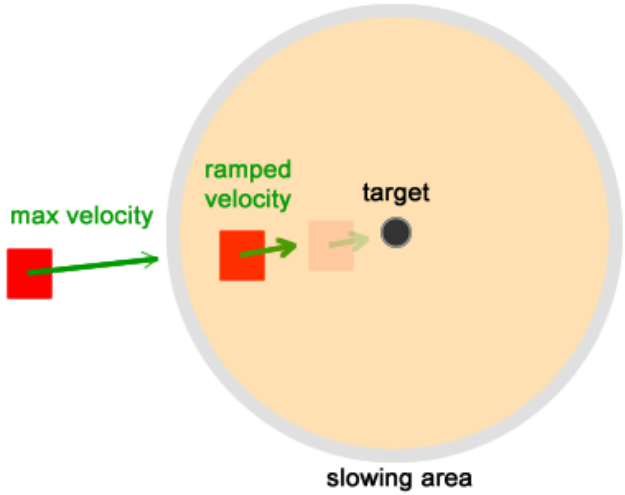
Passer par un point

Passage par une séquence de points

Suivi d'un objet mobile

Accointances

S'arréter en un point

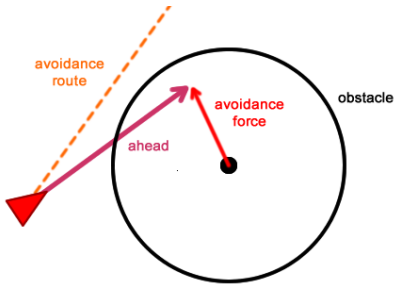


- **Comportement** : `Arrive`
- **Rôle(s)** :
  - Cible : C

Identique à `Seek` avec : $v_m = \begin{cases} v_0 & \text{si } d > d_0 \\ \frac{v_0}{d_0} d & \text{sinon} \end{cases}, d = \|\vec{PC}\|$

Accointances

Eviter des obstacles



- **Comportement** : `Evitement`
- **Rôle(s)** :
  - Obstacle : O

- **Objet mobile**
  - Position : P
  - Vitesse : $\vec{v}$
- **Obstacle**
  - Centre du volume englobant : C
  - Rayon du volume englobant : r

Accointances

Eviter des obstacles

- Palpeur : P' avec $P\vec{P}' = k\frac{P\vec{P}'}{\|P\vec{P}'\|}$
- $\|C\vec{P}'\| < r \Rightarrow P'' = (r+d)\frac{C\vec{P}'}{\|C\vec{P}'\|} - C$
- Comportement `Seek(P'')`

- Comment faire quand il y a plusieurs obstacles ?
- Et si on cherche à longer un mur ?

Accointances

Combiner des comportements

Par combinaison linéaire des forces correspondant aux comportements

- suivre une trajectoire ($\vec{F}_t$)
- éviter un obstacle O ($\vec{F}_o$)
- suivre le mur M_1 ($\vec{F}_m$)

Exemple

$$\vec{F} = 0.5\vec{F}_t + 0.25\vec{F}_o + 0.25\vec{F}_m$$

Capteurs virtuels

Perception

Par lancer de rayon

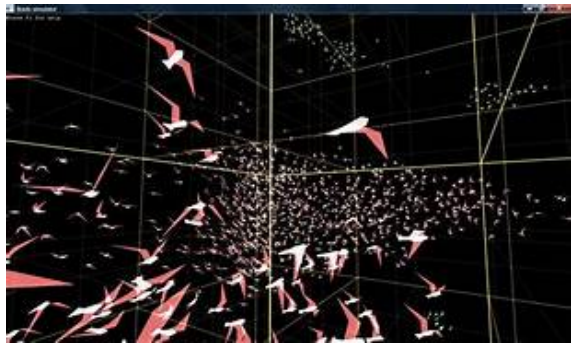
```
...
var raycaster  = new THREE.Raycaster(position, direction) ;
var intersects = raycaster.intersectsObjects(scene.children) ;
if(intersects.length > 0){
  pickedObject = intersects[0].object ;
  ...
}
```

Informations sur l'objet intersecté

- **object** : objet intersecté
- **point** : coordonnées du point intersecté
- **distance** : distance de l'origine du rayon au point d'intersection

Perception

Flocking



Problème

- Un ensemble d'entités qui forment un groupe
- Difficile à animer

Quoi ?

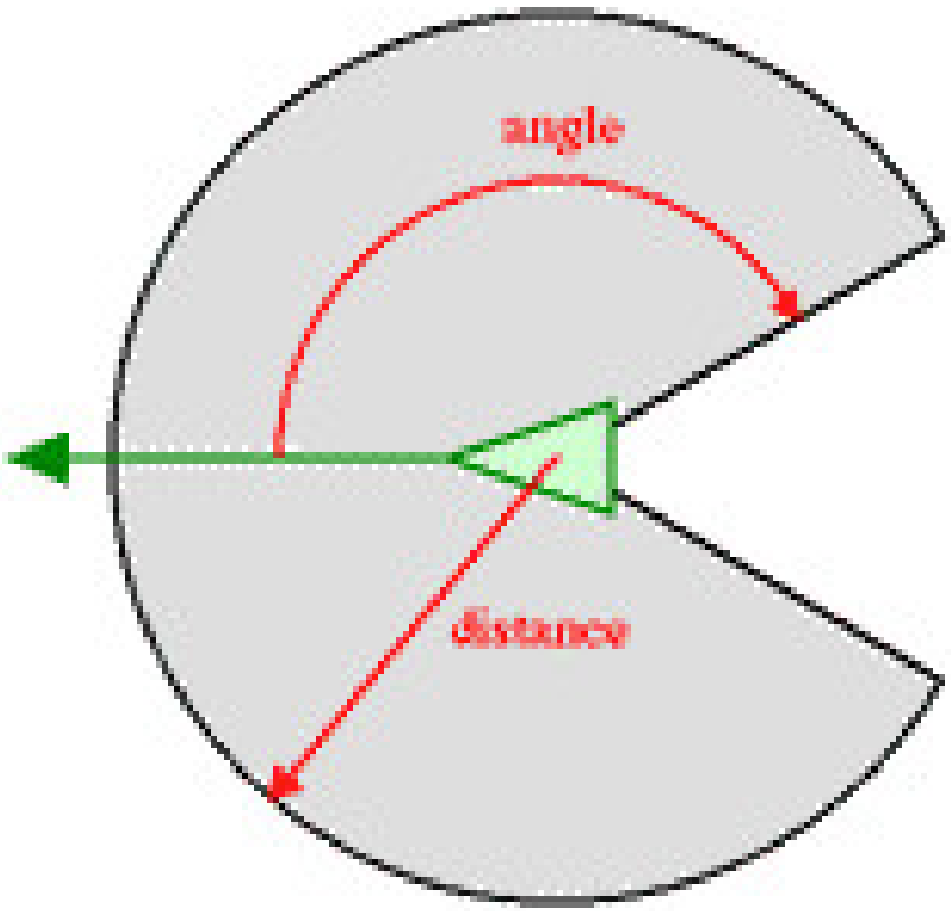
Garder la notion de groupes

Comment ?

Règles simples, identiques pour toutes les entités

Perception

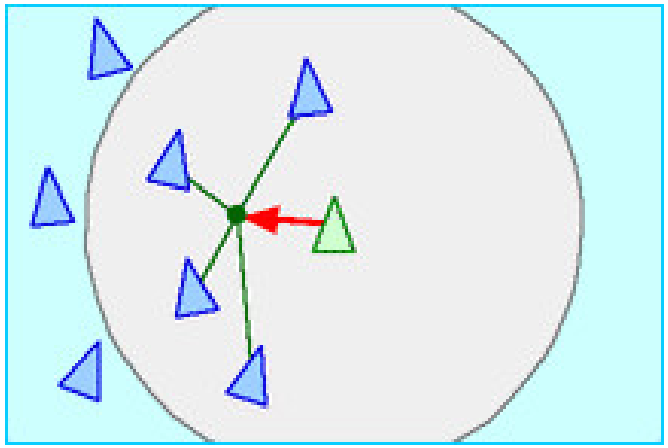
Flocking



Règle de cohésion

Les entités restent groupées

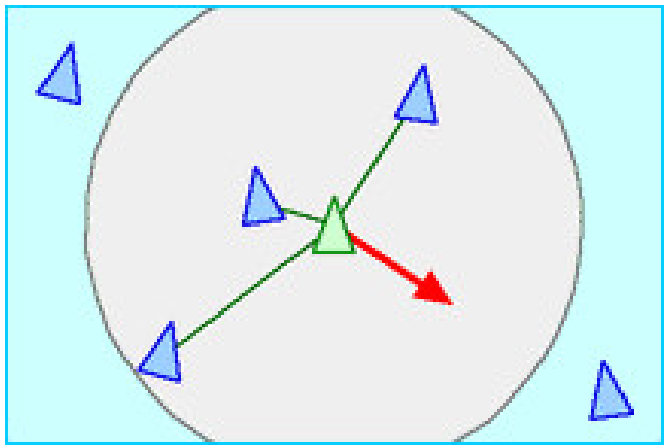
$$G = \sum_{i \in V} P_i, \vec{F}_c = \text{Seek}(G)$$



Règle de séparation

Les entités ne s'effondrent pas les unes sur les autres

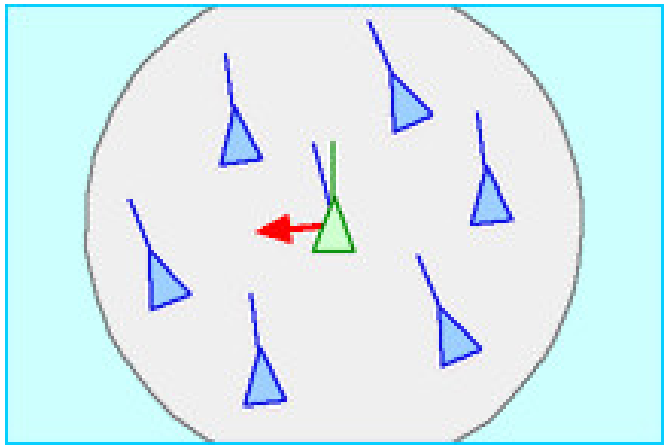
$$\vec{F_s} = \frac{1}{n} \sum_i \frac{\vec{P_iP}}{\|\vec{P_iP}\|}$$



Règle d'alignement

Les entités tendent à avoir la même orientation.

$$\vec{v}_m = \frac{1}{n}\sum \vec{v}_i, \vec{F}_a = k(\vec{v}_m - \vec{v})$$



$$\vec{F} = k_c \vec{F}_c + k_s \vec{F}_s + k_a \vec{F}_a$$

avec $k_c + k_s + k_a = 1$

Capteurs virtuels

Volumes de perception

Cylindre

- $C = \text{Cylindre}(C(x0, y0, z0), r, h)$
- $P(x, y, z) \in C \Leftrightarrow (x - x0)^2 + (z - z0)^2 < r^2 \wedge z - z0 < h$

Secteur angulaire

- $A = \text{Angle}(C, \vec{D}, h, \alpha)$
- $P \in A \Leftrightarrow \|\vec{PC}\| < h \wedge \frac{D \bullet \vec{CP}}{\|\vec{CP}\|]} > \cos(\alpha)$

Boite

- $B = \text{Boite}(x0, x1, y0, y1, z0, z1)$
- $P(x, y, z) \in B \Leftrightarrow x0 < x < x1 \wedge y0 < y < y1 \wedge z0 < z < z1$

Capteurs virtuels

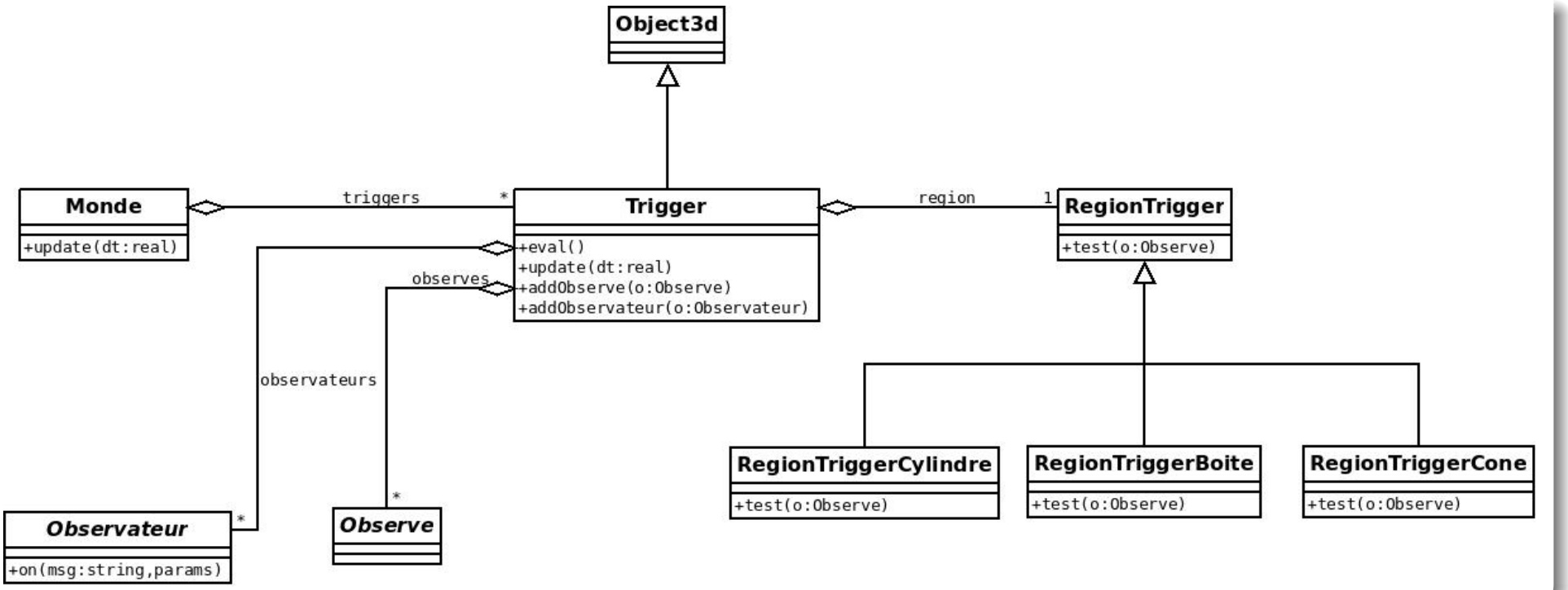
Triggers

Relations entre capteurs virtuels et Entités ?

- Utilisés pour exécuter de façon conditionnelle des actions
- Pattern 'observateur/observé' : les observateurs sont notifiés quand un observé entre/sort d'une région de l'espace

Capteurs virtuels

Triggers



- `update` : modifie les paramètres du capteur virtuel et du trigger
- `eval` : détermine s'il faut ou non envoyer une notification

Capteurs virtuels

Aura, focus, nimbus

Awareness

- Grandeur numérique (entre 0 et 1 ?)
- Donne l'importance d'une entité B pour une entité A.($Aw_A(B)$.

Canaux

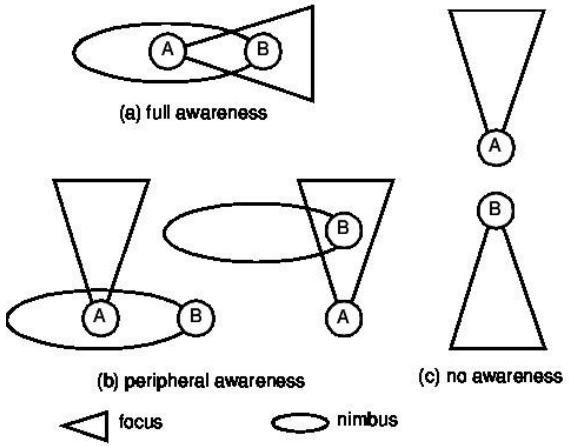
- Canaux : vue, ouïe, rayonnements E.M., ...
- Indépendance des canaux

Calcul de l'awareness de B pour A en fonction de

- ce que peut voir A ($\text{focus}_A(B)$)
- ce que donne à voir B ($\text{nimbus}_B(A)$)

Capteurs virtuels

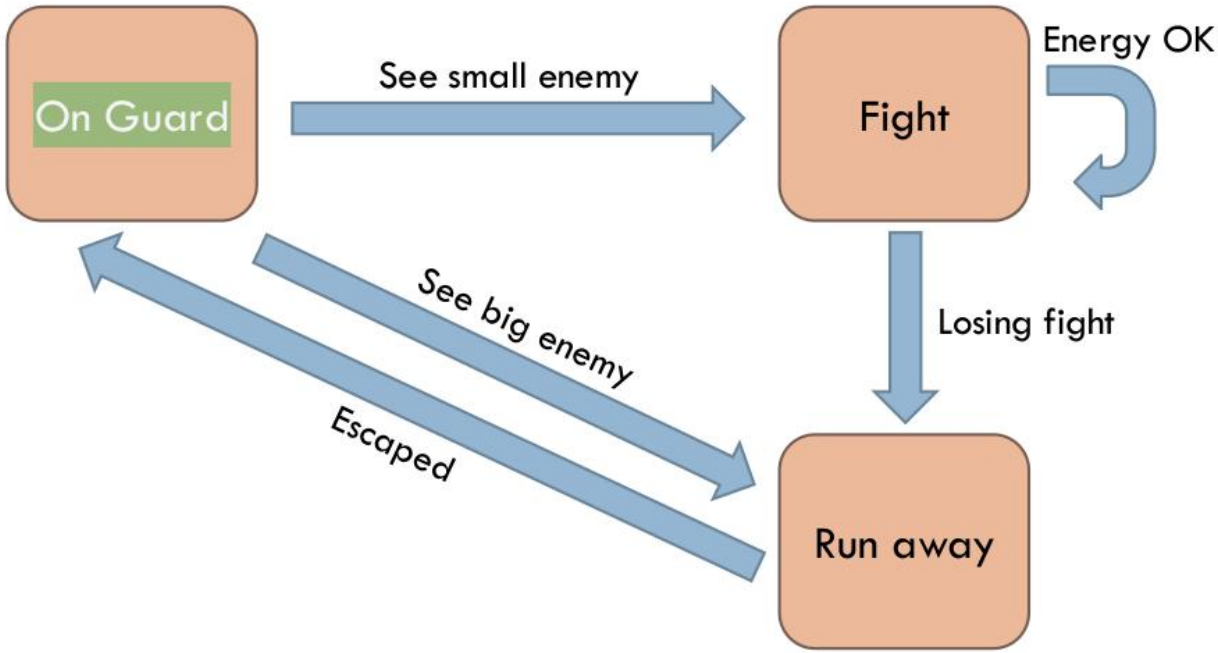
Aura, focus, nimbus



Exemple de règles d'évaluation

$B \in \text{focus}(A)$	$A \in \text{nimbus}(B)$	$Aw_A(B)$	Nature
oui	oui	1.00	perception centrale
oui	non	0.75	perception périphérique
non	oui	0.25	perception périphérique
non	non	0.00	pas de perception

Automates



- Description de comportements complexes
- Situation $\rightarrow$ Comportement

Automates

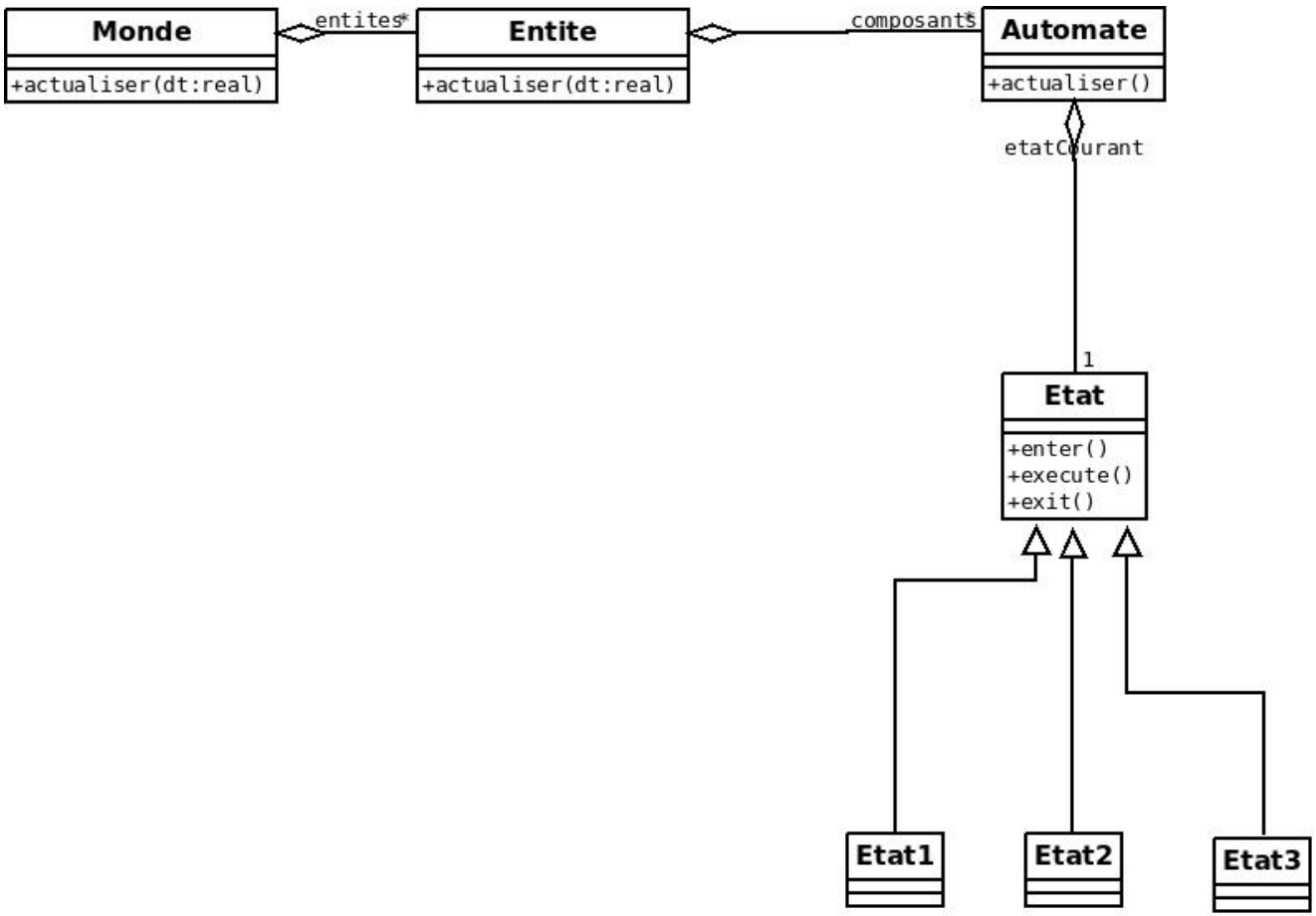
- Description de comportements complexes
- Situation $\rightarrow$ Comportement

- Très répandus
- Simple à comprendre
- Simple à implémenter
- Problème d'explicitation dans les cas complexes

Automates

```
var etat = 0 ; var encore = true ;
while(encore){
  switch(etat){
    case E0 : if(cond0){action0;etat=autreEtat;} else
              if(cond1){action1;etat=autreEtat;} else
              ...
              break ;
    case E1 : if(cond0){action0;etat=autreEtat;} else
              if(cond1){action1;etat=autreEtat;} else
              ...
              break ;
    .....
  }
}
```

Automates


Monde
+actualiser(dt:real)
entites*
Entite
+actualiser(dt:real)
composants*
Automate
+actualiser()
etatCourant
1
Etat
+enter()
+execute()
+exit()
Etat1
Etat2
Etat3

Automates

```
class FSM {
  constructor(states,s0){
    this.states = states ;
    this.currentState = s0 ;
    this.transition(s0) ;
  }

  get state() {return this.currentState ;}

  ...
```

Automates

```
  transition(state){
    const oldState = this.states[this.currentState] ;
    if(oldState && oldState.exit)oldState.exit.call(this);

    this.currentState = state ;
    const newState = this.states[state] ;
    if(newState && newState.enter)newState.enter.call(this);
  }

  update(){
    const state = this.states[this.currentState];
    if(state.update) state.update.call(this) ;
  }
}
```

Automates

```
class Comp extends Composant {
  constructor(entite,opts){
    super(entite) ;
    this.fsm = new FSM({
      idle : {
        enter : () => { ...},
        update : () => {if(COND1)this.fsm.transition('wait');}
      },
      wait : { ...}

    },'idle')
  }

  actualiser(dt){ this.fsm.update();}
}
```