

Lemmings : Night is Comming

Manuel Développeur

EUDES Robin – FREBY Rodolphe – GINOUX Pierre-Henri –
SAMBE Adj – LABAT Paul

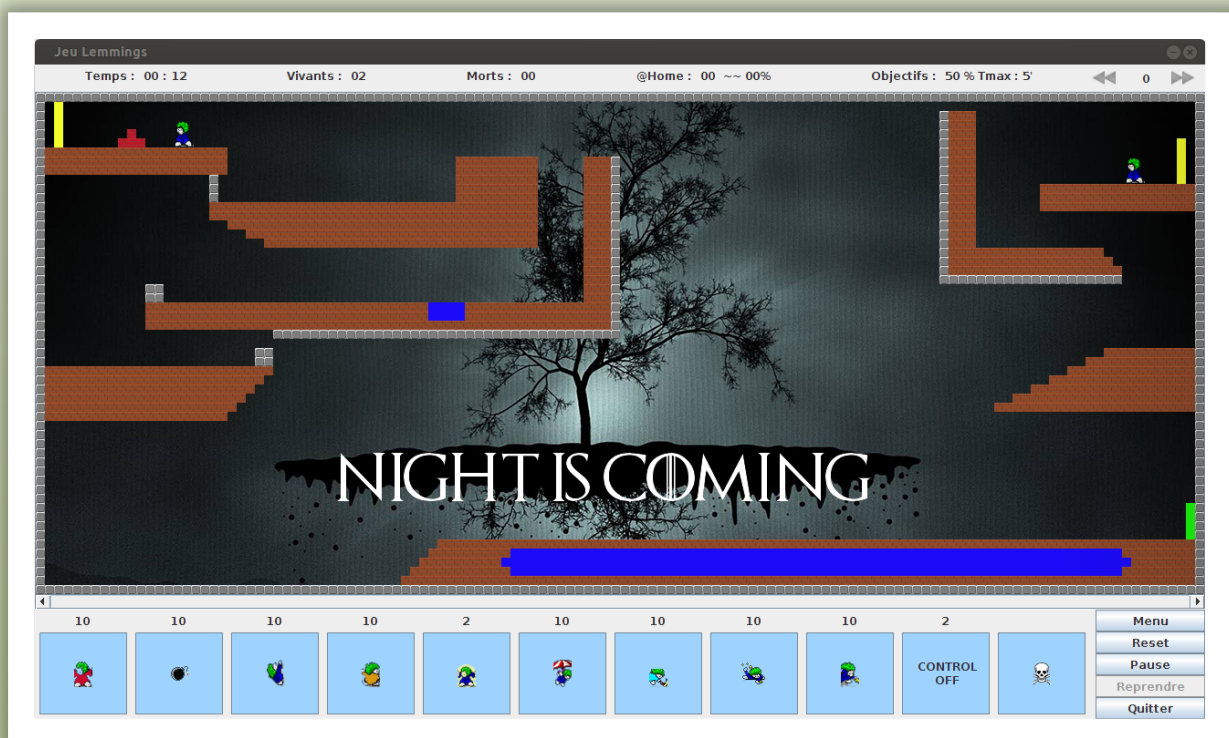
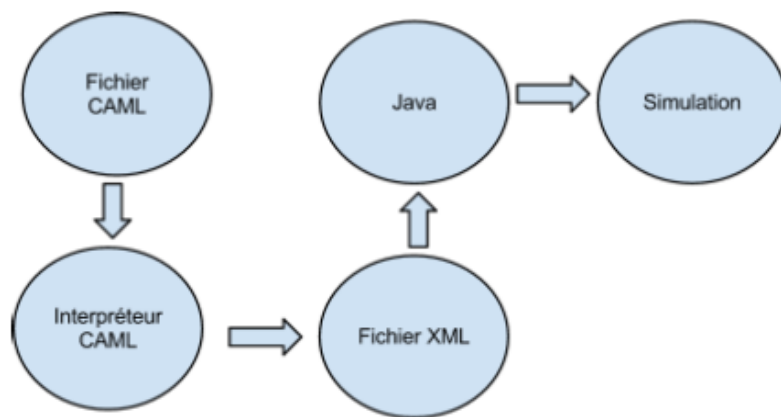


Table des matières

INTRODUCTION	3
PARTIE 1	4
I. STRUCTURE DU FICHIER OCAML	4
PARTIE 2	7
I. STRUCTURE DU LOGICIEL.....	7
II. ORGANISATION DU CODE JAVA.....	8
II.1 <i>jus.ricm.contexte</i>	9
II.2 <i>jus.ricm.lemmings</i>	9
II.3 <i>jus.ricm.automate</i>	9
II.4 <i>jus.ricm.gestionBouton</i>	9
II.5 <i>jus.ricm.graphique</i>	10
II.6 <i>jus.ricm.interpreteur</i>	10
II.7 <i>jus.ricm.ordonnanceur</i>	10
II.8 <i>jus.ricm.sound</i>	11
III. ORGANISATION DES CLASSES	11
IV. PRINCIPE DE FONCTIONNEMENT DU SIMULATEUR	12

Introduction

Le présent manuel donne toutes les instructions nécessaires pour la compréhension du logiciel Lemmings : Night is coming. Ce logiciel a pour objectif de simuler des lemmings grâce à des automates, afin de produire un jeu. Ceci est effectué par une saisie des données de l'utilisateur dans un fichier OCAML de manière à définir tous les paramètres qui seront nécessaires au jeu. Leur évolution sera faite par la suite par un programme codé en Java, et visualisé à l'aide d'une interface graphique.



Partie 1

I. Structure du fichier Ocaml

Le fichier OCAML est rempli par l'utilisateur. Cependant, il doit veiller à respecter la définition des types tels qu'ils ont été définis ainsi que la grammaire du langage. Le non-respect de ces procédures entrainera une erreur d'exécution lors de la génération du fichier XML.

Ci-dessous, la définition des types :

```
(***** Contexte *****)

type objectif = int;;
type nombreLemmings = int;;
type tempsMax = int;;

type contexte = objectif * nombreLemmings * tempsMax ;;

(***** Automate *****)

type nature = string ;;
type numero = string ;;
type etat = string ;;
type action = string ;;
type condition = string ;;

type transition = etat * etat * action * condition list ;;

type automate = numero * transition list;;

type comportement = nature * automate list ;;
(***** Map *****)

type type = int;;
type coordonneesX = int;;
type coordonneesY = int;;
type sens = string ;;
type taille = int;;

type Bloc = type * coordonneesX * coordonneesY * sens * taille ;;

type map = Bloc list ;;

(***** Boutons
```

```

*****
type nom = string ;;
type nombreCoups = int ;;

type bouton = nom * nombreCoups ;;

type liste_boutons = bouton list;;

(***** jeu
*****
type jeu = contexte * comportement list * map * liste_boutons ;;

```

Édition du fichier XML

La fonction ***ecrituretout*** permet la génération du fichier XML. Cette fonction écrit dans le fichier XML toutes les informations saisies au préalable par l'utilisateur tout en respectant les types. Ni un parser, ni une redéfinition des types sous forme de token n'est nécessaire.

Néanmoins, deux instructions sont requises pour la génération du fichier XML :

— ***open string*** ;;

On fait appel au module String qui nous fournit l'ensemble des fonctions nécessaires pour la manipulation de chaînes de caractères. Parmi-elles, nous avons la fonction ***length*** qui permet de calculer la taille d'une chaîne de caractère donnée en entrée par l'utilisateur.

— ***let sortie = open_out "Fichier.xml";;***

Cette ligne indique le fichier XML à créer ou à écraser s'il existe déjà.

La fonction ***ecrituretout*** regroupe les fonctions suivantes :

ecriturejeu : permet de spécifier l'objectif du jeu, le nombre de lemmings ainsi que la durée du jeu

ecrituretoutComportement : décrit la liste des comportements des lemmings

ecrituremap : affiche l'ensemble des blocs de la map

ecrituretoutBouton : affiche les boutons de l'interface graphique

Ces fonctions utilisent elles-mêmes des fonctions intermédiaires qui sont adaptées à d'éventuelles modifications (par exemple ajout de nouveaux attributs).

Voici un exemple de fichier XML généré :

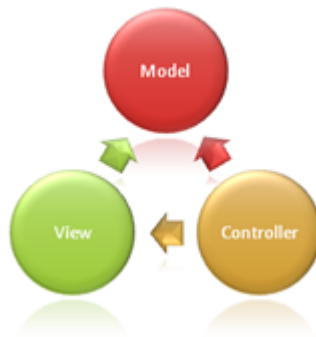
```
▼<Jeu>
  <Fenetre objectif="50" nbLemmings="10" tempsMax="5"/>
  ▼<ListeComportements>
    ▼<Comportement nature="base">
      ▶<Automate type="1">...</Automate>
    </Comportement>
    ▶<Comportement nature="parapluie">...</Comportement>
    ▶<Comportement nature="grimpeur">...</Comportement>
    ▼<Comportement nature="bloqueur">
      ▼<Automate type="4">
        ▼<Transition etat_init="BLOQUE" etat_suiv="BLOQUE" action="bloquer">
          <Condition valeur="isSol"/>
        </Transition>
        ▼<Transition etat_init="BLOQUE" etat_suiv="RETOUR" action="retourner">
          <Condition valeur="isVide"/>
        </Transition>
      </Automate>
    </Comportement>
    ▶<Comportement nature="creuserH">...</Comportement>
    ▶<Comportement nature="creuserV">...</Comportement>
    ▶<Comportement nature="creuserD">...</Comportement>
    ▶<Comportement nature="escalier">...</Comportement>
  </ListeComportements>
  ▶<map>...</map>
  ▼<ListeBoutons>
    <Bouton nom="parapluie" nombre="10"/>
    <Bouton nom="grimpeur" nombre="10"/>
    <Bouton nom="bloqueur" nombre="10"/>
    <Bouton nom="creuseurH" nombre="10"/>
    <Bouton nom="creuseurV" nombre="10"/>
    <Bouton nom="creuseurD" nombre="10"/>
    <Bouton nom="Escalier" nombre="10"/>
    <Bouton nom="Luciolle" nombre="10"/>
    <Bouton nom="supp" nombre="10"/>
  </ListeBoutons>
</Jeu>
```

Dans le cas contraire, un message d'erreur est généré pour indiquer le bon type à utiliser.

Partie 2

I. Structure du logiciel















L'architecture du logiciel repose sur le modèle MVC (Modèle-Vue-Contrôleur). Elle est découpée de la façon suivante :



Le **modèle** assure la gestion des événements et le fonctionnement des automates.
Le **contrôleur** déclenche les événements et informe des changements au modèle.
La **vue** gère l'affichage du monde.

II. Organisation du code Java

Le code est découpé en plusieurs packages.

- ▼  > src
 - ▶  jus.ricm.automate
 - ▶  jus.ricm.contexte
 - ▶  jus.ricm.gestionBouton
 - ▶  jus.ricm.graphique
 - ▶  jus.ricm.interpreteur
 - ▶  jus.ricm.lemmings
 - ▶  > jus.ricm.ordonnanceur
 - ▶  jus.ricm.sound
- ▶  JRE System Library [JavaSE-1.7]
- ▶  Referenced Libraries
 - ▶  img
 - ▶  library
 - ▶  music

II.1 jus.ricm.contexte

Ce package comprend la classe Contexte. Cette classe contient l'ensemble des variables qui sont nécessaires au fonctionnement du programme et qui doivent être uniques. De plus, celles-ci sont accessibles depuis n'importe quelle classe. Grâce au contenu de cette classe, nous pouvons exécuter le jeu, le mettre en pause, etc.

La classe Contexte comprend également une méthode qui permet de réinitialiser le contexte, qui peut être par exemple utile lorsque l'on veut faire une nouvelle partie.

II.2 jus.ricm.lemmings

Il contient la classe Lemmings. Cette classe représente tous les petits lemmings qui se déplaceront dans la fenêtre de jeu. Elle regroupe aussi tous les attributs permettant de représenter un lemming : sa position dans l'espace, son automate courant, son état courant dans l'automate. Les conditions particulières qui s'appliquent sur lui, et son état dans le jeu (vivant, mort ou arrivé à la sortie). Elle contient également les méthodes permettant de définir les événements qui s'appliquent à un lemming, et ainsi d'envoyer ces données dans l'automate, afin de trouver son état futur, et l'action qu'il doit effectuer.

II.3 jus.ricm.automate

Contient les classes Automate et AutomateSuivantCondition.

La première permet de représenter un automate. Elle contient des méthodes pour créer un automate, l'afficher et pour effectuer une transition d'un état vers un autre.

La seconde permet de représenter une transition au travers d'un couple liant un état d'arrivée, une liste de conditions pour atteindre le-dit état d'arrivée, et l'action associée à ce changement d'état.

II.4 jus.ricm.gestionBouton

Ce package contient l'ensemble des écouteurs de bouton. Ils permettent d'assigner une action particulière à chaque type de bouton. De cette façon, il suffit de modifier une classe pour modifier le comportement d'un ensemble de boutons qui utilisent ce type d'écouteur.

II.5 jus.ricm.graphique

Dans ce package sont présentes les classes gérant l’affichage du jeu. Une classe Bloc permet de définir ce qu’est un bloc et tous les attributs qui seront nécessaires aux lemmings pour se déplacer ; une classe Map qui définit les méthodes permettant de se déplacer dans une matrice de bloc et de la modifier de façon rapide. Il y a également trois classes principales. Les classes Menu, FenetreJeu et FenetrePrincipale définissent les fenêtres du logiciel qui seront vues par l’utilisateur. Il existe également deux classes qui permettent de définir le sélecteur de fichier XML qui est lancé juste avant le démarrage de la partie. Enfin, la classe Map_canvas définit les méthodes d’affichages du décor, des lemmings, ainsi que les animations diverses et la méthode faisant apparaître les lemmings à intervalle régulier.

II.6 jus.ricm.interpreteur

Il gère le fichier XML donné en entrée. Dans ce package, la classe ParserXML contient une méthode qui permet de parcourir l’ensemble du fichier XML afin de récupérer tous ses nœuds et attributs. Tous les objets concernant le contexte (l’objectif que l’utilisateur doit atteindre, le nombre de lemmings dans le jeu et la durée du jeu), les automates des lemmings, la map et les boutons y sont créés et initialisés.

II.7 jus.ricm.ordonnanceur

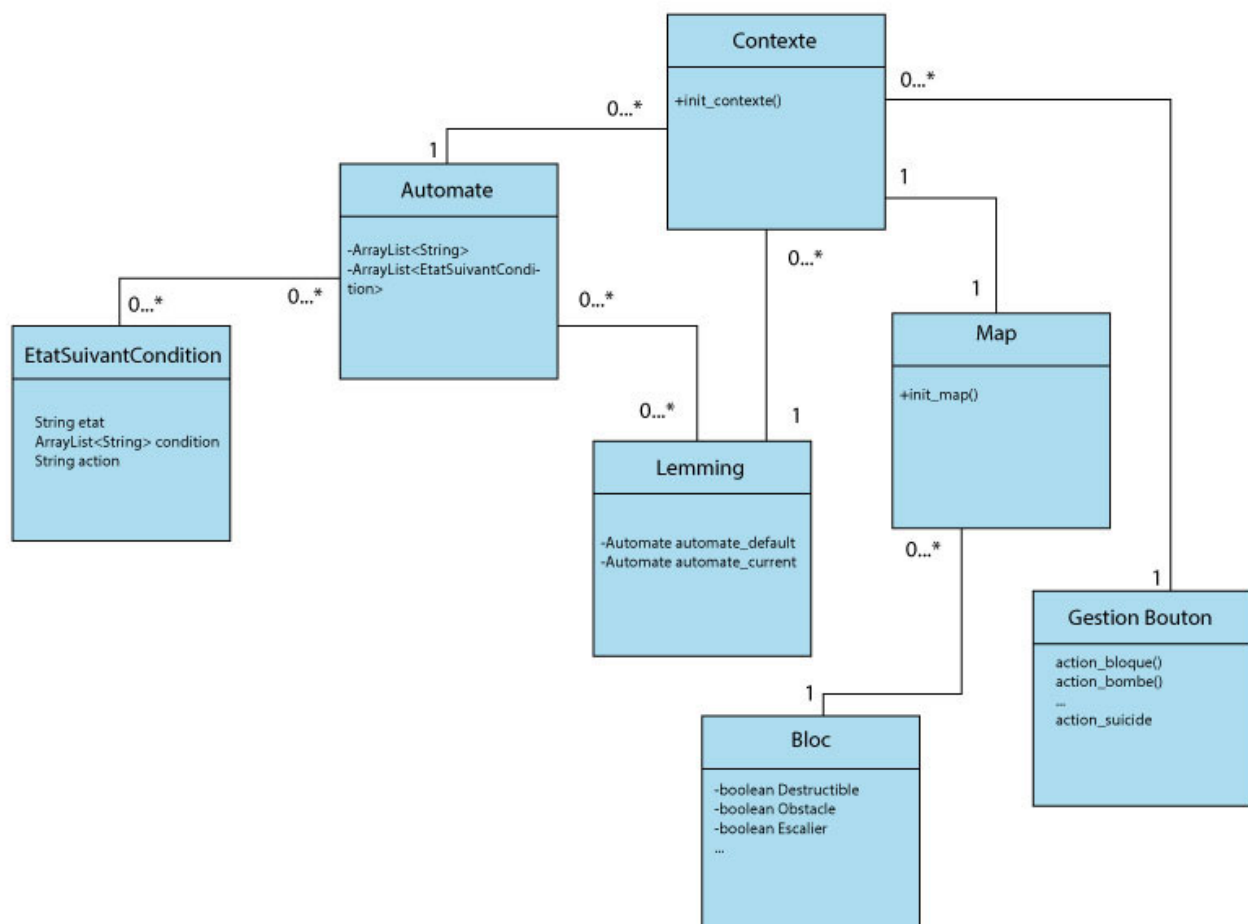
Dans ce package se trouve une unique classe. Elle définit le thread principal qui constitue le moteur de jeu. Elle contient également les méthodes qui permettent de savoir si le jeu est fini (en fonction du temps, du nombre de lemmings encore en vie et l’objectif que l’utilisateur doit atteindre). La méthode effectuant le traitement des lemmings est présente dans cette classe. Elle demande à chaque lemming d’interagir et de se déplacer dans son environnement, au moyen des méthodes définies dans la classe Lemmings.

II.8 jus.ricm.sound

Ce package contient la classe sound. Elle implémente Runnable et cloneable. La majorité des initialisations et des chargements sont effectués dans le constructeur. On charge un buffer dans le constructeur, ce qui permet par la suite de ne pas avoir à recharger le fichier son. On instancie donc dans le contexte un objet son en static, qui sera par la suite cloné à chaque utilisation. Ensuite, on lance un thread qui exécute le son demandé. Ce système nous permet de parcourir en une seule fois les fichiers audio pour une optimisation maximale du temps d'exécution.

III. Organisation des classes

L'organisation des classes se présente comme suit. Il s'agit d'une version simplifiée, qui a pour but de simplifier la compréhension de la structure dans son ensemble.



IV. Principe de fonctionnement du simulateur

Une fois le chargement du fichier XML fait, la carte ainsi que les boutons sont initialisés.

Ensuite, le simulateur traite à chaque cycle 5 étapes :

- Mise à jour de l’affichage :
- Mise à jour du temps
- Mise à jour des informations concernant le jeu, telle que le nombre de lemmings en vie, le temps de jeu.
- Traitement de chaque lemming un à un :
 - Analyse de leur environnement proche afin de déterminer une liste de conditions liées aux lemmings
 - Recherche de la transition dans l’automate adaptée en comparant la liste de condition représentant l’environnement du lemming avec la liste de condition nécessaire pour une transition.
 - Application de l’action associée à la transition trouvée, avec la mise à jour des attributs des lemmings si nécessaire.
 - Vérification des conditions de victoire.

La classe Ordonnanceur possède une méthode ***gameLoop()*** permettant de mettre à jour l’affichage, de parcourir les lemmings, d’appeler la méthode effectuant les transitions et les actions liées aux automates en fonction des conditions.

```

/**
 * Definition du Thread principal. Permet de faire tourner le jeu.
 */
Thread updateThread = new Thread() {
    @Override
    public void run() {
        //while avec condition qui permet d'arreter le thread
        while (Contexte.runnable) {
            Contexte.canvas.repaint(); // refresh le canvas, rappel paintComponent()

            // si le jeu n'est pas en pause et que le contexte est initialiser
            if(!Contexte.pause && Contexte.initContexte){
                Contexte.jeu.update_time(); //maj du temps
                Contexte.jeu.update_compteurs_type(); // maj des informations
                //ajout d'un nouveau lemmings au jeu tous les 100 cycle (environ 4sec)
                i++;
                if(i%100 == 0){
                    Contexte.canvas.born(Contexte.n_max,Contexte.spawnX1,Contexte.spawnY1,
                    Contexte.spawnX2, Contexte.spawnY2);
                }

                //mise a jour des lemmings
                traiteLemmings();

                Contexte.time = System.currentTimeMillis();
                //verifie si on gagne
                isVictoire();
            }

            try {
                // Delay and give other thread a chance to run
                Thread.sleep(Contexte.UPDATE_PERIOD); // milliseconds
            } catch (InterruptedException ignore) {}
        }
    }
}

```

```

/**
 * Boucle principale de jeu
 */
public void gameLoop()
{
    updateThread.start(); // called back run()
}

```

```

/**
 * Traitement du lemming...
 */
private void traiteLemmings()
{
    int j;
    if(Contexte.initContexte) // si contexte initialise
    {
        //parcour du tableau jusqu'au nombre de lemmings max
        for(j=0;j<Contexte.nbLemmings ;j++)
        {
            if(Contexte.tabLemmings[j].isVivant()) // si lemmings vivant, on le traite
            {
                Contexte.tabLemmings[j].transition(); //on applique les
transitions
            }
        }
    }
}

```

La classe Lemmings quant à elle possède une méthode **transition** qui est appelée dans l'Ordonnanceur. Celle-ci analyse totalement l'environnement autour du lemming (vérification de la présence du sol, du vide, d'un mur...) et crée ainsi une liste de condition représentant de manière symbolique cet environnement. Par la suite, la méthode **transition** va chercher l'état suivant dans l'automate en fonction de l'environnement entourant le lemming et d'une liste de conditions nécessaires au changement d'état, ainsi que l'action associée. Enfin, la méthode effectue l'action en mettant à jour les attributs du lemming qui auraient pu être modifiés par l'action.

```

public int transition(){
    ArrayList<String> listeEnvLem = new ArrayList<String>();
    //Liste de condition du lemmings, elle permet de stocker
    //l'analyse de l'environnement du lemmings

    if(this.isEsca()){ //Verification de la possibilite de construire un escalier
        listeEnvLem.add("isEsca");
    }
    if(this.isVide()){ //Verification de la présence du vide
        listeEnvLem.add("isVide");
    }else{
        listeEnvLem.add("isSol");
    }
    if(this.isMortel()){ //Verification d'une chute mortelle
        listeEnvLem.add("isMortel");
    }
    if(this.isEau()){ //Verification de la presence de l'eau

```

```

        listeEnvLem.add("isEau");
    }
    if(this.isObstacle()){
        //Verification de la presence d'un obstacle sur le chemin d'un lemmings
        listeEnvLem.add("isObstacle");
    }else{
        listeEnvLem.add("isCorniche");
    }
    if(this.hautMap()){
        //Verification de la presence du haut de la carte au dessus du lemmings
        listeEnvLem.add("hautMap");
    }
    if(this.isBloqueur()){
        //Verification de la presence d'un bloqueur sur le chemin du lemmings
        listeEnvLem.add("isBloqueur");
    }
    if(this.isCassableH()){ //Verification de la possibilite de creuser
horizontallement
        listeEnvLem.add("isCreusableH");
    }
    if(this.isCassableV()){ //Verification de la possibilite de creuser verticalement
        listeEnvLem.add("isCreusableV");
    }
    if(this.isCassableD()){ //Verification de la possibilite de creuser en diagonale
        listeEnvLem.add("isCreusableD");
    }
    if(this.isEscalier()){ //Verification de la presence d'un escalier
        listeEnvLem.add("isEscalier");
    }
    if(isFinH()){
        //Verification de la fin d'une action consistant a creuser horizontalement
        listeEnvLem.add("finH");
    }
    if(isFinV()){ //Verification de la fin d'une action consistant a creuser
verticalement
        listeEnvLem.add("finV");
    }
    if(isFinE()){ //Verification de la fin d'une action consistant a poser un escalier
        listeEnvLem.add("finE");
    }
    if(isFinD()){ //Verification de la fin d'une action consistant a creuser en
diagonale
        listeEnvLem.add("finD");
    }
}

```

```

String action;
EtatSuivantCondition listerecup;
// On va maintenant chercher la bonne transition dans l'automate
// et ainsi mettre a jour l'etat courant du lemmings et effectuer
// l'action trouvee
listerecup = this.automate_current.transition(listeEnvLem, etatcourant);
if(listerecup == null){
    action = "retourner";
}else{
    action = listerecup.getAction();
    // On cherche l'indice du nouvel etat
    this.setEtatcourant(this.automate_current.etatExiste(
        this.automate_current.getListeEtats(), listerecup.getEtat()));
}
this.actionL = action;
if(this.bombe){ // Si le lemmings est une bombe, on le rapproche de l'explosion
    this.cpt_bombe++;
}

if(this.cpt_bombe == 240){ // Verification de la condition d'explosion du
lemmings
    if(Contexte.mute)
    {
        Sound sonClonedBombe = (Sound)
Contexte.bombe.cloneObj();
        sonClonedBombe.SetFile(Contexte.bombe.GetFile());
        sonClonedBombe.SetInfo(Contexte.bombe.GetInfo());
        sonClonedBombe.SetAudioFormat(
            Contexte.bombe.GetAudioFormat());
        sonClonedBombe.SetAudiobuffer(
            Contexte.bombe.GetAudiobuffer());
        Thread t_sonClonedBombe = new Thread(sonClonedBombe);
        t_sonClonedBombe.start();
    }
    this.vie = "mort";
    // Tous les blocs du lemmings sont detruis, utile pour la mort d'un
    //bloqueur
    Contexte.map.getBloc(this.coord_x, this.coord_y).toVide();
    Contexte.map.getBloc(this.coord_x, this.coord_y + 10).toVide();
    Contexte.map.getBloc(this.coord_x, this.coord_y + 20).toVide();
    Contexte.map.getBloc(this.coord_x - 10, this.coord_y).toVide();
    Contexte.map.getBloc(this.coord_x - 10, this.coord_y + 10).toVide();
    Contexte.map.getBloc(this.coord_x - 10, this.coord_y + 20).toVide();
    Contexte.map.getBloc(this.coord_x - 20, this.coord_y).toVide();
    Contexte.map.getBloc(this.coord_x - 20, this.coord_y + 10).toVide();
    Contexte.map.getBloc(this.coord_x - 20, this.coord_y + 20).toVide();
}
// On va maintenant en fonction de l'action a traiter, appeler la bonne methode

```



```

switch (action){
case "marcher" :
    this.marcher();
    break;
case "tomber" :
    this.tomber();
    break;
case "atterrir" :
    this.atterrir();
    break;
case "tourner" :
    this.tourner();
    break;
case "grimper" :
    this.grimper();
    break;
case "bloquer" :
    break;
case "retourner" : // Cas special, on revient a l'automate precedent du
Lemmings
    // Utile pour les automates temporaires tels que creuser.
    this.retourner();
    break;
case "creuserH" :
    this.creuserH();
    break;
case "creuserV" :
    this.creuserV();
    break;
case "creuserD" :
    this.creuserD();
    break;
case "mourir" :
    this.mourir();
    break;
case "monter" :
    this.monter();
    break;
case "construire" :
    this.construire();
    break;
}
// On renvoie l'etat courant (-1 si etat puit)
return this.etatcourant;
}

```

Ci-dessous, un exemple de quelques une des méthodes de tests de l'environnement du lemming, ainsi que quelques une des méthodes pour la résolution des actions :

```
//Verification de la presence de l'eau
public boolean isEau(){
    // Premier test pour le bruitage, le lemmings est sur l'eau
    if (Contexte.map.getBloc(this.coord_x + 1, this.coord_y).getEau()
        && Contexte.map.getBloc(this.coord_x + 1,
            this.coord_y + 20).getEau()
        && Contexte.map.getBloc(this.coord_x + 1,
            this.coord_y + 10).getEau()) {
        if(Contexte.mute)
        { //Si le son n'est pas mute, on fait le bruitage
            Sound sonClonedEau = (Sound) Contexte.eau.cloneObj();
            sonClonedEau.SetFile(Contexte.eau.GetFile());
            sonClonedEau.SetInfo(Contexte.eau.GetInfo());

            sonClonedEau.SetAudioFormat(Contexte.eau.GetAudioFormat());
            sonClonedEau.SetAudiobuffer(Contexte.eau.GetAudiobuffer());
            Thread t_sonClonedEau = new Thread(sonClonedEau);
            t_sonClonedEau.start();

        }
    }
    // Veritable test qui renvoie True si le lemmings est sur de l'eau, False sinon
    return (Contexte.map.getBloc(this.coord_x + 1, this.coord_y).getEau()
        && Contexte.map.getBloc(this.coord_x + 1,
            this.coord_y + 20).getEau()
        && Contexte.map.getBloc(this.coord_x + 1,
            this.coord_y + 10).getEau());
}
```

```
// Verification de la presence de vide sous le lemmings
public boolean isVide(){
    return (Contexte.map.getBloc(this.coord_x + 1, this.coord_y).getVide()
        && Contexte.map.getBloc(this.coord_x + 1,
            this.coord_y + 20).getVide()
        && Contexte.map.getBloc(this.coord_x + 1,
            this.coord_y + 10).getVide());
}
```

```

public void marcher(){ //Methode pour faire marcher le lemming
    if (sens == 1){ //on va vers la droite
        this.coord_y = this.coord_y + 1;
    }
    else { // on va vers la gauche
        this.coord_y = this.coord_y - 1;
    }
}

```

La classe Automate quant à elle possède une méthode permettant de trouver la transition adaptée pour un lemming. Elle prend en paramètre une liste de condition (celle du lemming) et renvoie un objet EtatSuivantCondition, qui est un couple comprenant une liste de String représentant les conditions pour atteindre un état, un String pour le nom de l'état d'arrivée, et un String pour l'action.

```

public EtatSuivantCondition transition(ArrayList<String> param, int etatcourant){
    EtatSuivantCondition res = null;
    ArrayList<EtatSuivantCondition> listerecup =
this.listeEtatCond.get(etatcourant);
    for(int i = 0; i < listerecup.size(); i++){
        ArrayList<String> listeCond = listerecup.get(i).getCondition();
        int nbr = 0;

        for(int j = 0; j < listeCond.size(); j++){
            for(int k = 0; k < param.size(); k++){
                if(param.get(k).compareTo(listeCond.get(j)) == 0){
                    nbr++; // On a une condition identique a un
                        // positionnement du lemmings
                }
            }
        }
        if(nbr == listeCond.size()){ // Si le nombre de correspondance
            // est egal a la taille de la liste de condition, on a
            // trouve la bonne transition
            res = listerecup.get(i);
        }
    }
    return res;
}

```

Cette classe comprend également une méthode capable de renvoyer un entier représentant l'indice de l'état dans l'automate passé en paramètre :

```
public int etatExiste(ArrayList<String> listEtat, String elemRecherche){  
    int position = -1; // Valeur par défaut  
    for(int i = 0; i < listEtat.size(); i++){  
        // Parcours de la liste pour retrouver l'etat  
        if(listEtat.get(i).compareTo(elemRecherche) == 0){  
            // On a retrouve l'element recherche  
  
            position = i;  
        }  
    }  
    return position;  
}
```