Dossier de Developpement

Ce document traitera de l’action de l’intelligence artificielle utilisée dans ce projet. Les décisions d’actions sont calculées par l’IA de manière semi-aléatoire et selon les caractéristiques d’un joueur.

L’intelligence Artificielle

# L’Intelligence Artificielle

*La déclaration de la classe Scene.h*

|  |
| --- |
| #ifndef SCENE\_H  #define SCENE\_H  #include <iostream>  #include <vector>  using namespace std;  #include "Bot.h"  class Scene  {  public:  Scene();  ~Scene();  protected:  private:  vector<Bot\*> externBots;  Bot\* localBot;  int\*\* map;  };  #endif // SCENE\_H |

Ifndef SCENE\_H est une structure conditionnelle permettant de déclarer le header *Scene* uniquement s’il n’a pas déjà été déclaré au préalable.

define SCENE\_H permet de déclarer le header de *Scene*.

#include <iostream> et #include <vector> permettent d’inclure des bibliothèques particulières de la bibliothèque standard.

using namespace std permet l’utilisation du domaine de nom *std*, et donc par la même occasion, des différents types et fonctions qui y sont contenus.

#include "Bot.h" permet d’inclure le header *Bot.h*.

class Scene représente le début de la déclaration de la classe *Scene*.

Scene() et ~Scene() sont respectivement, la fonction constructeur et la fonction destructeur de la classe *Scene*.

Dans la section private de la classe, on déclare les attributs qui seront accessibles uniquement dans la classe *Scene*.

vector<Bot\*> externBots; permet de déclarer un tableau vectoriel qui va contenir l’ensemble des personnages de type Bots se trouvant sur la carte du jeu.

Bot\* localBot; est un pointeur sur une variable de type Bot.

int\*\* map; est un pointeur sur un pointeur sur un nombre de map de type *int*.

La définition de la classe *Scene.cpp*

|  |
| --- |
| #include "Scene.h"  Scene::Scene()  {  localBot = new Bot();  while(1)  {  localBot->decisionAction();  system("PAUSE");  }  }  Scene::~Scene()  {  //dtor  } |

#include "Scene.h" permet d’inclure le header *Scene.h*.

Scene::Scene() permet de faire la définition de la classe *Scene*.

localBot = new Bot(); permet de créer une instance, nommée localBot, de la classe *Bot*.

while(1) permet de débuter une boucle infinie.

localBot->decisionAction(); permet d’utiliser la méthode *decisionAction()* de l’objet *localBot* par le pointeur sur *localBot*.

system("PAUSE"); permet de mettre le système de l’ordinateur en Pause pour ne pas surconsommer le CPU, ni la mémoire.

Scene::~Scene() permet de faire la définition du destructeur de la classe *Scene*.

Bot.h

|  |
| --- |
| #ifndef BOT\_H  #define BOT\_H  #include <iostream>  #include <vector>  #include <string>  using namespace std;  #include "StaticConstraint.h"  #include "VarConstraint.h"  #include "RecursiveConstraint.h"  #include "Constraint.h"  #include "Action.h"  #include "IA.h"  #define ACTION\_PRIM(id, weight) NewAction = new Action(id, weight); \  AIPrim->addAction(NewAction);\  NewAI = new IA();\  AISec.push\_back(NewAI);  #define ACTION\_SEC(id, weight) NewAction = new Action(id, weight);\  NewAI->addAction(NewAction);  // -- On Action's Contrainst --  #define STATIC(type, value) NewConstraint = new StaticConstraint(type, value); \  NewAction->addConstraint(NewConstraint);  #define VAR(type, value) NewConstraint = new VarConstraint(type, value); \  NewAction->addConstraint(NewConstraint);  #define RECURSIVE(type, value) NewRecConstraint = new RecursiveConstraint(type, value); \  NewAction->addConstraint(NewRecConstraint);  // -- On Constrainst's Contrainst --  #define REP(level) rep[level] = NewRecConstraint;  #define CSTATIC(type, value, level) NewConstraint = new StaticConstraint(type, value); \  rep[level]->addConstraint(NewConstraint);  #define CVAR(type, value, level) NewConstraint = new VarConstraint(type, value); \  rep[level]->addConstraint(NewConstraint);  #define CRECURSIVE(type, value, level) NewRecConstraint = new RecursiveConstraint(type, value); \  rep[level]->addConstraint(NewRecConstraint)  #define ADD 1  #define FACT 2  #define SUP 3  #define INF 4  #define WHITE 5  #define DIV 6  #define GESTUEL 1  #define TAPER 11  #define DORMIR 12  #define ORDONNER 13  #define SALUER 14  #define JOUER 15  #define SE\_NOURRIR 16  #define DON 2  #define CADEAU 21  #define CADEAU\_PIEGE 22  #define DEPLACEMENT 3  #define PROMENER 31  #define ATTIRER 32  #define EXPLORER 33  #define EXPRESSION 4  #define PARLER 41  #define CRIER 42  #define GEINDRE 43  #define RIRE 44  class Bot  {  public:  Bot();  ~Bot();  int decisionAction();  protected:  private:  IA\* AIPrim;  vector<IA\*> AISec;  int coordx;  int coordy;  int direction;  double sociability;  double actionTendency;  double temper;  double energy;  double hunger;  };  #endif // BOT\_H |

Le header de Bot permet déclarer le nom de la classe Bot et ses attributs : une coordonnée X et Y pour localiser le Bot sur une grille invisible, afin de pouvoir modifier certaines caractéristiques en fonction de l’emplacement du Bot. Les attributs sociabilité, tendance à l’action, tempérament, énergie et faim permettent de stocker les caractéristiques personnelles d’un Bot pour que l’Intelligence Artificielle se base sur ces variables.

IA\* AIPrim; représente un pointeur sur une variable de type IA.

vector<IA\*> AISec; définit un tableau vectoriel qui regroupe les pointeurs sur IA.

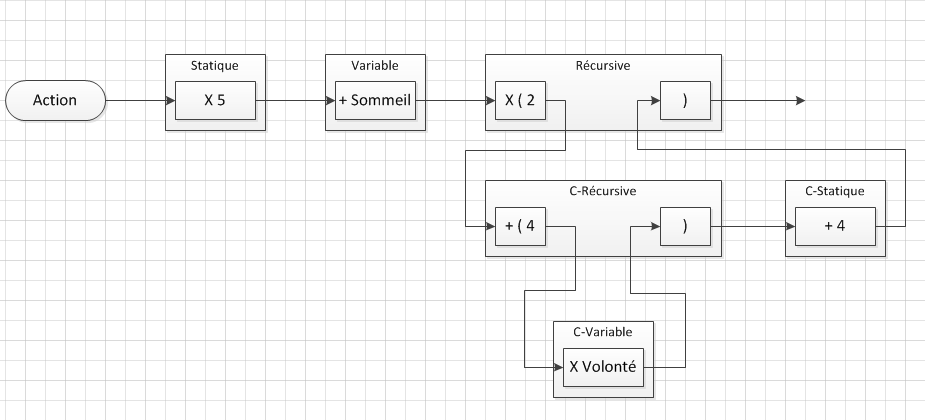
L’utilisation de constante de préprocesseur a permis d’économiser du temps et de la place dans le code, mais a surtout permis d’éviter une redondance du code.

Bot.cpp

|  |
| --- |
| #include "Bot.h"  Bot::Bot()  {  /\* -- Initialisation of Bot's Variable -- \*/  sociability = (rand() % 11) + 1;  actionTendency = (rand() % 11) + 1;  temper = (rand() % 11) + 1;  energy = (rand() % 11) + 1;  hunger = (rand() % 11) + 1;  /\* -- AI's Declaration -- \*/  AIPrim = new IA();  IA\* NewAI = NULL;  /\* -- Action's Declaration -- \*/  Action\* NewAction = NULL;  Constraint\* NewConstraint = NULL;  RecursiveConstraint\* NewRecConstraint = NULL;  vector<RecursiveConstraint\*> rep(1);  /\* -- Action List -- \*/  ACTION\_PRIM(GESTUEL, 1)  ACTION\_SEC(TAPER, 18)  RECURSIVE(ADD, -1) REP(0)  CVAR(FACT, &temper, 0)  CSTATIC(ADD, 5, 0)  CSTATIC(FACT, 4, 0)  ACTION\_SEC(DORMIR, 11)  RECURSIVE(ADD, 2) REP(0)  CVAR(FACT, &energy, 0)  ACTION\_SEC(ORDONNER, 10)  RECURSIVE(ADD, 0) REP(0)  CVAR(ADD, &temper, 0)  CVAR(ADD, &actionTendency, 0)  CSTATIC(FACT, 2, 0)  ACTION\_SEC(SALUER, 15)  ACTION\_SEC(JOUER, 1)  ACTION\_SEC(SE\_NOURRIR, 1)  ACTION\_PRIM(DON, 1)  ACTION\_SEC(CADEAU, 1)  ACTION\_SEC(CADEAU\_PIEGE, 1)  ACTION\_PRIM(DEPLACEMENT, 1);  ACTION\_SEC(PROMENER, 1)  ACTION\_SEC(ATTIRER, 1)  ACTION\_SEC(EXPLORER, 1)  ACTION\_PRIM(EXPRESSION, 1)  ACTION\_SEC(PARLER, 50)  ACTION\_SEC(CRIER, 50)  ACTION\_SEC(GEINDRE, 50)  ACTION\_SEC(RIRE, 50)  }  Bot::~Bot()  {  //dtor  }  int Bot::decisionAction()  {  int ai\_sec = AIPrim->decisionCalc();  AISec[ai\_sec]->decisionCalc();  } |

Le Bot.cpp permet de définir la classe Bot et ses méthodes.

Le constructeur de cette classe permet de mettre en place un objet de type Bot avec une liste d’action avec des poids prédéfinis par des calculs se rapprochant au mieux de la réalité. Les calculs sont prévus de manière à ajouter, soustraire, multiplier des valeurs statique, c’est-à-dire que la valeur traitée est un chiffre entre 0 et 9, variable, la valeur traitée est calculé avec une variable changeante selon les caractéristiques du Bot, ou encore récursive, la valeur traitée est une combinaison redondante d’ajout, soustraction et/ou multiplication de valeur statique, variable et/ou à nouveau redondante.



int Bot::decisionAction() est une méthode de la classe *Bot* qui permet de stocker une valeur numérique, résultat de la méthode decisionCalc() de l’objet AIPrim, dans la variable ai\_sec, puis utilise cette dernière pour exécuter la même méthode à partie du tableau vectoriel AISec à la cellule ai\_sec.

IA.h

|  |
| --- |
| #ifndef IA\_H  #define IA\_H  #include <iostream>  #include <vector>  #include <stdlib.h>  using namespace std;  #include "Action.h"  #include "Trace.h"  class IA  {  protected:  vector<Action\*> actions;  private:  public:  IA();  ~IA();  int decisionCalc();  int addAction( Action\* action );  };  #endif // IA\_H |

Le header IA.h permet de déclarer la classe de l’IA en elle-même. En protected, on déclare un tableau vectoriel qui va contenir la différente liste d’actions probables, sélectionnées par les calculs effectué dans le Bot.cpp.

On déclare dans cette classe deux méthodes :

decisionCalc() qui renvoie une valeur numérique et addAction() qui prend un pointeur sur une variable de type Action en paramètre et qui renvoie une valeur numérique.

IA.cpp

|  |
| --- |
| #include "IA.h"  // -- Constructeur --  IA::IA()  {  // -- Initialisation des Actions --  // -- Initialisation des Secteurs Decisionnels --  }  IA::~IA()  {  //dtor  }  int IA::decisionCalc()  {  TRACE("Activation AI --" << endl)  /\* -- Calculating of the weight's sum -- \*/  int sum = 0;  for(int i=0; i<actions.size(); i++)  {  sum += (int) actions[i]->weightCalc();  }  /\* -- Epic Random function -- \*/  int random = rand() % sum + 1;  /\* -- Action's Choice -- \*/  int incProba = 1, retour = -1;  for(int i=0; i<actions.size(); i++)  {  incProba += (int) actions[i]->Getweight();  if(incProba > random)  {  retour = i;  break;  }  }  TRACE("AI's choice : " << retour+1 << endl << endl)  return retour;  }  int IA::addAction( Action\* action )  {  actions.push\_back(action);  } |

IA.cpp permet de définir la classe IA et ses méthodes.

Cette classe possède deux méthodes :

decisionCalc() permet de calculer la somme des poids des options d’actions et de renvoyer le numéro de l’action la plus volumineuse.

int IA::addAction( Action\* action ) permet d’ajouter une action au tableau vectoriel *actions* pour garder une trace de l’action effectuée grâce à la méthode *push\_back*.

Action.h

|  |
| --- |
| #include "Trace.h"  #include "Constraint.h"  #include "StaticConstraint.h"  #include "VarConstraint.h"  #include "RecursiveConstraint.h"  class Action  {  public:  Action( int id, double weight );  ~Action();  int addConstraint( Constraint \*constraint );  double weightCalc();  double Getweight() { return m\_actualWeight; }  protected:  private:  int m\_id;  double m\_weight, m\_actualWeight;  vector<Constraint\*> constraintsList;  }; |

Le header Action.h montre la déclaration de la classe *Action* et de ses méthodes.

Une action est représentée par un identifiant, un poids, un poids actuel, ainsi qu’un tableau vectoriel des différentes contraintes applicables au poids actuel pour obtenir le poids définitif.

Cette classe comporte un constructeur surchargé qui prend deux paramètres : un identifiant et un poids ; et trois méthodes :

Une méthode *AddConstraint* qui prend un pointeur sur une variable de type *Constraint* en paramètre, ressort une valeur numérique.

double weightCalc(); est une méthode, renvoyant une valeur numérique décimal, qui calcule le poids d’une action.

La dernière méthode est un Getter qui permet de récupérer le poids actuel *m\_actualWeight*.

Action.cpp

|  |
| --- |
| #include "Action.h"  Action::Action( int id, double weight ) : m\_id(id), m\_weight(weight)  {  }  Action::~Action()  {  //dtor  }  int Action::addConstraint( Constraint\* constraint )  {  constraintsList.push\_back(constraint);  return 0;  }  double Action::weightCalc()  {  TRACE(" Action " << m\_id << " : " << m\_weight)  double weight = m\_weight;  for(int i = 0; i < constraintsList.size(); i++)  {  weight = constraintsList[i]->weightCalc(weight);  }  TRACE(" -> " << weight << endl)  m\_actualWeight = weight;  return weight;  } |

StaticConstraint.h

|  |
| --- |
| #ifndef STATICCONSTRAINT\_H  #define STATICCONSTRAINT\_H  #include "Constraint.h"  #include "Trace.h"  class StaticConstraint : public Constraint  {  public:  StaticConstraint(int type=1, double value=0) : Constraint(type), m\_value(value) {}  double Getvalue() { return m\_value; }  void Setvalue(double val) { m\_value = val; }  virtual double weightCalc(double weight);  protected:  private:  double m\_value;  };  #endif // STATICCONSTRAINT\_H |

Le header StaticConstraint.h contient la déclaration de la classe *StaticConstraint* et de ses méthodes.

Un attribut de type *int* permet de stocker une valeur numérique et quatre méthodes complètent la classe :

Un Getter *Getvalue()* permet de récupérer la valeur de m\_value à l’extérieur de la classe.

Un Setter *Setvalue()* permet de modifier la valeur de m\_value à l’intérieur de la classe.

Un constructeur surchargé prenant une variable *type* numérique et une autre *value* de type double en paramètre, permet de stocker les paramètres dans les attributs internes.

virtual double weightCalc(double weight); est une méthode virtuel qui prend un poids de type *double* en paramètre et qui renvoie une valeur numérique décimale.

StaticConstraint.cpp

|  |
| --- |
| #include "StaticConstraint.h"  double StaticConstraint::weightCalc(double weight)  {  // Choice of Constraint's action  switch(m\_type)  {  case 1 :  {  weight += m\_value;  TRACE("+" << m\_value)  } break;  case 2 :  {  weight \*= m\_value;  TRACE("\*" << m\_value)  } break;  case 3 :  {  if(weight > m\_value)  weight = m\_value;  TRACE(" |>" << m\_value << "| ")  } break;  case 4 :  {  if(weight < m\_value)  weight = m\_value;  TRACE(" |<" << m\_value << "| ")  } break;  case 5 :  {  weight = m\_value;  TRACE(" |=" << m\_value << "| ")  } break;  case 6 :  {  weight /= m\_value;  TRACE("/" << m\_value)  } break;  }  return weight;  } |

VarConstraint.h

|  |
| --- |
| #ifndef VARCONSTRAINT\_H  #define VARCONSTRAINT\_H  #include "Constraint.h"  #include "Trace.h"  class VarConstraint : public Constraint  {  public:  VarConstraint(int type = 5, double\* variable = 0) : Constraint(type), m\_variable(variable) {}  double\* Getvariable() { return m\_variable; }  void Setvariable(double\* val) { m\_variable = val; }  virtual double weightCalc(double weight);  protected:  private:  double\* m\_variable;  };  #endif // VARCONSTRAINT\_H |

VarConstraint.cpp

|  |
| --- |
| #include "VarConstraint.h"  double VarConstraint::weightCalc(double weight)  {  // Choice of Constraint's action  if(m\_variable != 0)  {  switch(m\_type)  {  case 1 :  {  weight += \*m\_variable;  TRACE("+" << \*m\_variable)  } break;  case 2 :  {  weight \*= \*m\_variable;  TRACE("\*" << \*m\_variable)  } break;  case 3 :  {  if(weight > \*m\_variable)  weight = \*m\_variable;  TRACE(" |<" << \*m\_variable << "| ")  } break;  case 4 :  {  if(weight < \*m\_variable)  weight = \*m\_variable;  TRACE(" |<" << \*m\_variable << "| ")  } break;  case 5:  {  weight = \*m\_variable;  TRACE(" |=" << \*m\_variable << "| ")  } break;  case 6:  {  weight /= m\_value;  TRACE("/" << m\_value)  } break;  }  }  return weight;  } |

RecursiveConstraint.h

|  |
| --- |
| #ifndef RECURSIVECONSTRAINT\_H  #define RECURSIVECONSTRAINT\_H  #include <vector>  #include "Constraint.h"  #include "Trace.h"  using namespace std;  class RecursiveConstraint : public Constraint  {  public:  RecursiveConstraint(int type = 1, double value = 0) : Constraint(type), m\_initialValue(value) {}  vector<Constraint\*> GetconstraintList() { return m\_constraintList; }  void SetconstraintList(vector<Constraint\*> val) { m\_constraintList = val; }  double GetinitialValue() { return m\_initialValue; }  void SetinitialValue(double initialValue) { m\_initialValue = initialValue; }  virtual double weightCalc(double weight);  double ConstraintCalc(double weight);  int addConstraint(Constraint \*constraint);  protected:  private:  vector<Constraint\*> m\_constraintList;  double m\_initialValue;  };  #endif // RECURSIVECONSTRAINT\_H |

RecursiveConstraint.cpp

|  |
| --- |
| #include "RecursiveConstraint.h"  double RecursiveConstraint::weightCalc(double weight)  {  // Choice of Constraint's action  double value = m\_initialValue;  switch(m\_type)  {  case 1 :  {  TRACE("+(" << value)  weight += ConstraintCalc(value);  } break;  case 2 :  {  TRACE("\*(" << value)  weight \*= ConstraintCalc(value);  } break;  case 3 :  {  TRACE(" |>(" << value)  if(weight > value)  weight = value;  TRACE("| ")  } break;  case 4 :  {  TRACE(" |<(" << value)  if(weight < value)  weight = value;  TRACE("| ")  } break;  case 5:  {  TRACE(" |=(" << value )  weight = value;  TRACE("| ")  } break;  case 6:  {  weight /= m\_value;  TRACE("/" << m\_value)  } break;  }  TRACE(")")  return weight;  }  double RecursiveConstraint::ConstraintCalc(double weight)  {  // Calcul of weight  for(int i=0; i<m\_constraintList.size(); i++)  {  weight = m\_constraintList[i]->weightCalc(weight);  }  return weight;  }  int RecursiveConstraint::addConstraint( Constraint \*constraint )  {  m\_constraintList.push\_back(constraint);  } |