2014

Projet de fin d'études

Département de génie logiciel et des TI

Rapport final

GTI792 PROJET DE FIN D'ÉTUDES EN GÉNIE LOGICIEL

Tourelle de Paintball

Auteurs

RENÉ-ALEXANDRE GIROUX

GIRR19079003

Professeur superviseur

ALAIN APRIL

12 décembre 2014

TABLES DES MATIÈRES

Le génie pour l'industrie

Figures

Le génie pour l'industrie

Définitions

Terme	Définition
PFE	Projet de fin d'études
FSM	Finite State Machine (FSM)
TrackTargetState	État de l'agent lorsqu'il cherche une cible (RechercherCibleEtat)
AimTargetState	État de l'agent lorsqu'il vise vers une cible (ViserCibleEtat)
ShootState	État de l'agent lorsqu'il doit tirer une cible (TirerEtat)
FeedbackState	État de l'agent lorsqu'il doit savoir si la cible a été touchée (RetroactionEtat)
Agent	Système qui prend les décisions de la tourelle
Agent intelligent	Système qui prend les décisions de la tourelle
IA	Intelligence artificielle
Classe	Permets de définir des attributs et méthodes pour un objet
Objet	Classe instanciée dans le code

Introduction

Ce projet de fin d'études porte sur un travail commencé par un professeur à l'École de technologie supérieure, soit la tourelle de paintball. Ce projet fait par Alain April a débuté en hiver 2011. La particularité de ce projet est qu'il est multidisciplinaire. Les concentrations de Génie logiciel, Génie électrique ainsi que Génie mécanique peuvent y participer pleinement dû au fait qu'il comporte une partie informatique, une partie électrique ainsi que des pièces mécaniques.

La partie logicielle de la tourelle de paintball a été écrite avec le langage c++. L'objectif de ce travail est de concevoir un agent intelligent qui permet de prendre des décisions selon certaines conditions.

Le présent document va montrer et expliquer les différentes phases qui ont permis l'élaboration de l'agent intelligent. Tout d'abord, une mise en contexte sera présentée ainsi que les objectifs de la solution. Après, il sera question de conception soit les décisions prises pour répondre aux différents critères du projet ainsi que le fonctionnement entre les différents modules du cerveau de la tourelle.

Aussi, l'explication des différentes parties de l'agent intelligent et comment l'information voyage dans celui-ci. Une partie sur les recommandations et suggestions pour augmenter les fonctionnalités sera montrée.

Contexte

Le paintball est une activité sportive où s'affrontent deux équipes. Il y a plusieurs types de jeu, soit du paintball récréatif, simulation militaire ou compétitive. L'objectif de la partie est déterminé par l'administrateur du lieu ou par l'organisateur de l'évènement.

Un exemple d'objectif serait la capture d'un drapeau dans la zone adverse. Lorsqu'un tel évènement a lieu, chaque personne apporte son équipement c'est-à-dire son fusil, son casque, son armure de protection, etc.

La particularité du paintball, c'est qu'il est possible à chacun d'apporté ou de louer sur place des équipements spéciaux tels que des grenades, bazooka. Le professeur Alain April a eu l'idée de mettre en place une tourelle de paintball, qui peut repérer des cibles et les viser pour ensuite les tirer.

Le problème associé à ce projet est que la tourelle de paintball n'avait pas une assez bonne intelligence artificielle. Donc, le mandat de ce projet de fin d'études et de créer un agent intelligent capable d'envoyer les bonnes commandes selon ce que la vision de la tourelle repère. De plus, il faut trouver un moyen pour que la tourelle puisse savoir si la cible repérée a bel et bien été touchée.

Objectif

L'objectif du PFE est de concevoir un agent intelligent dans la tourelle de paintball qui respecte certains critères. Ces critères sont la performance, la modularité et la fiabilité. De plus, le système intelligent doit être conçu de façon à respecter le patron « état ».

Pour ce qui a trait à la performance, puisque l'application à plusieurs « thread », dont un qui s'occupe de l'algorithme de vision, il faut que le logiciel puisse être le plus rapide possible. Il ne faut pas que l'application commence à être lente, car dans un jeu de paintball, les joueurs sont toujours en mouvement rapide donc le système de vision doit reconnaître les cibles rapidement. Bref, il faut le moins de latence possible entre la vision et le système d'intelligence.

La modularité est le principe par lequel il est possible de segmenter les différentes parties de la solution en petit module interchangeable. Ces modules sont capables d'effectuer des tâches par eux-mêmes sans avoir besoins des autres modules. Cependant, lorsqu'ils sont tous mis ensemble, ils sont capables d'offrir une solution complète à un problème. Donc, pour ce système, il faut que la solution soit modulable ainsi, s'il y a des modifications à faire dans l'agent intelligent, il sera beaucoup plus simple de les effectuer. Aussi, ce critère aide à la maintenabilité, car s'il y a un défaut dans l'application, c'est beaucoup plus simple d'isoler le problème dans son module et de le corriger.

Pour la fiabilité, c'est un critère qui dit que le logiciel doit donner des résultats corrects et être tolérant aux pannes. Dans le cas où le système de vision arrêterait de fonctionner, l'agent intelligent va

continuer son traitement cependant, il ne pourra pas donner les instructions pour une nouvelle cible, car c'est la vision qui s'occupe de ce point. Si le système intelligent tombe en panne, le reste de l'application peut continuer de fonctionner cependant, la tourelle ne peut pas tirer sur les cibles que la vision trouve, car c'est l'intelligence qui s'occupe de faire ses appels.

Conception

Cette partie va décrire tout ce qui a été fait pour mener à bien ce projet.

Besoins du projet

Les besoins de ce projet sont très clairs, il faut un nouvel agent intelligent capable de prendre des décisions selon ce que l'algorithme de vision va nous donner. Cet agent, doit être de type FSM soit « Finite State Machine » ou machine à état fini. De plus, il faut répondre aux 3 critères logiciels discutés dans la section avant soit, la performance, modularité et la fiabilité. Aussi, il faut une recommandation pour évaluer la performance de l'agent pour savoir si la cible a bien été atteinte.

Donc voici les besoins de ce projet

- Performance
- Fiabilité
- Modularité
- Nouvel agent intelligent (FSM)
- Recommandation pour savoir si une cible est atteinte

Architecture

La tourelle de paintball est composée de plusieurs parties. Il y a une partie mécanique comme le montre la *figure 1* ci-dessous. Elle contient un ordinateur portable qui lui a le logiciel. Deux microcontrôleurs, ceux-ci servent à contrôler les deux moteurs. Les moteurs servent à bouger la tourelle et à activer la gâchette pour tirer sur une cible.

Figure 1 - Vue architecturale de la tourelle

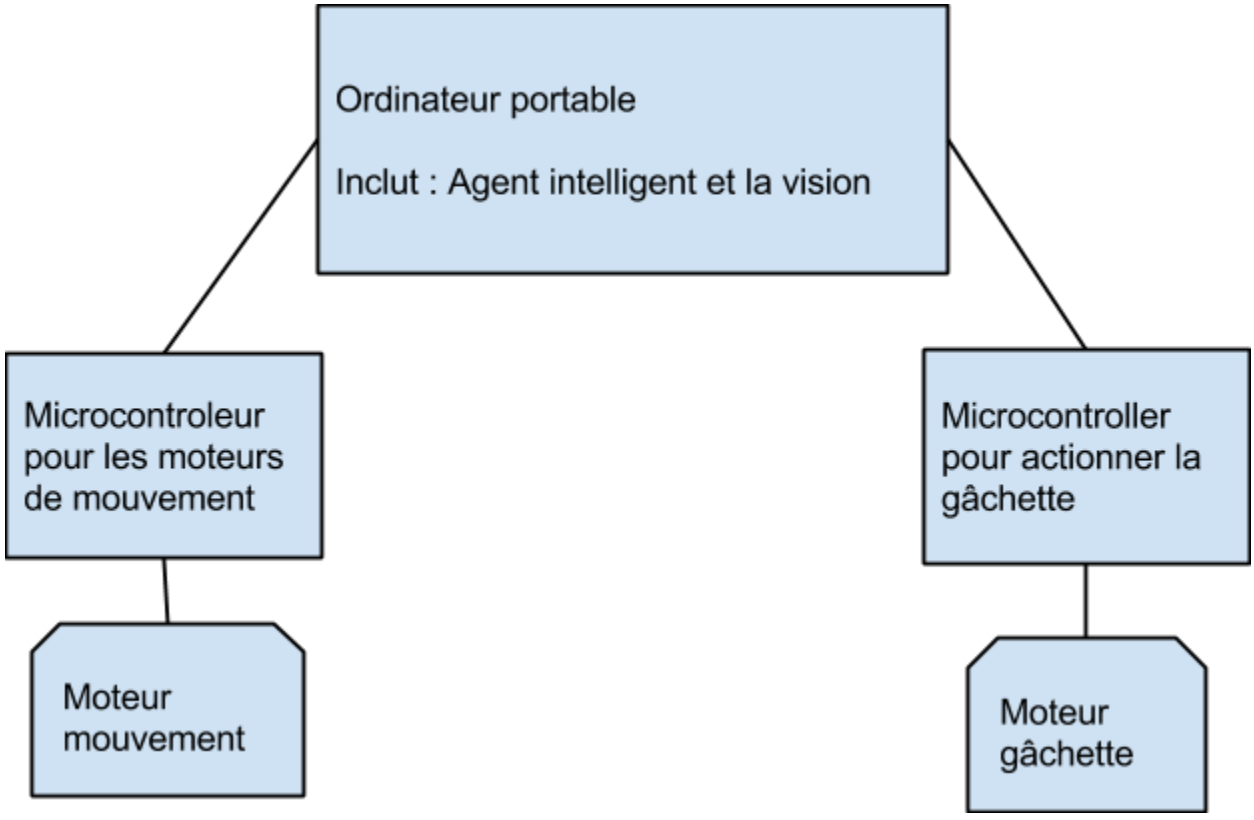
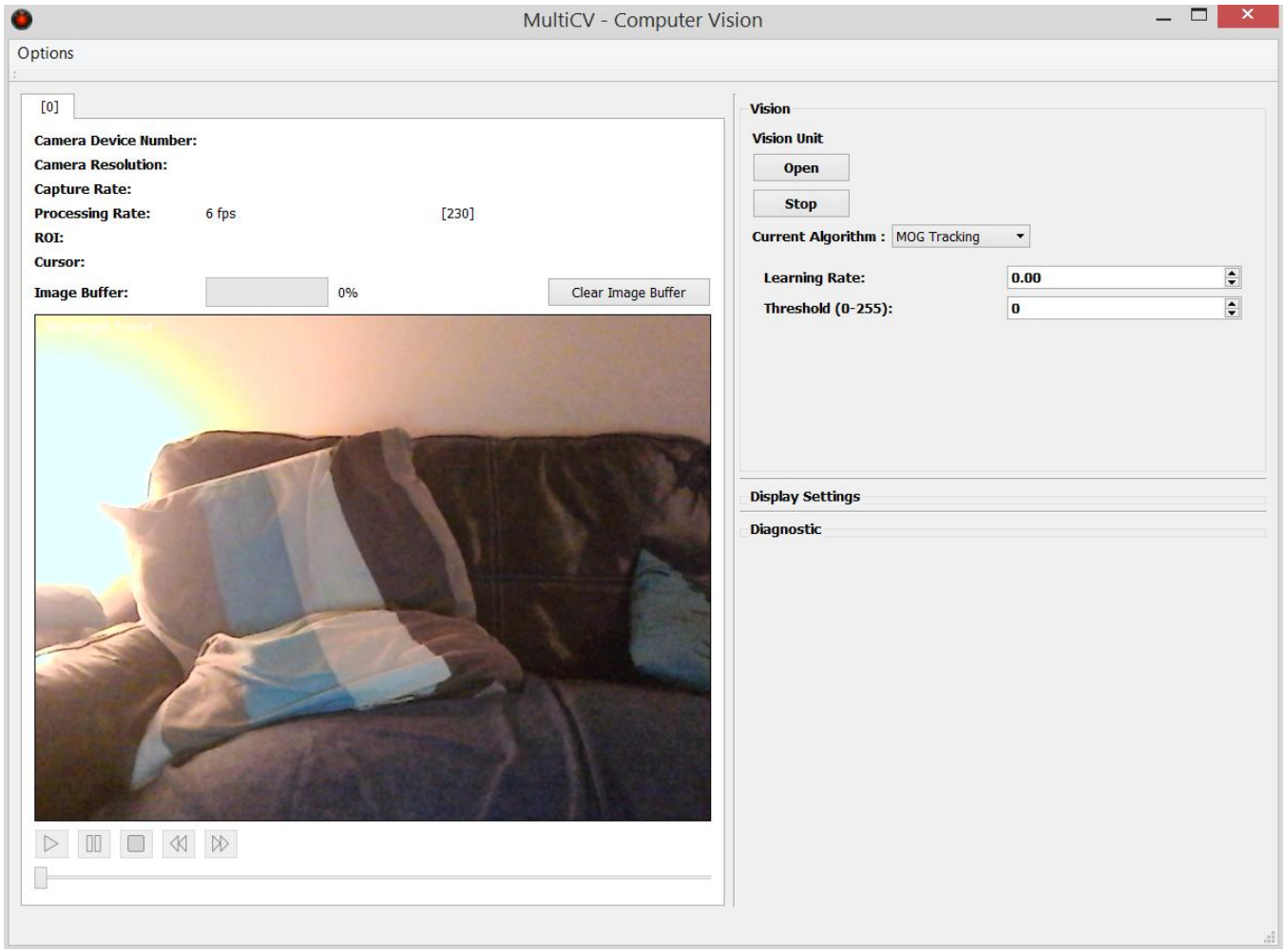


Figure 2 - Vue de l'application logicielle

Voici à quoi ressemble l'application logicielle qui réside sur l'ordinateur portable.



Agent intelligent

Analyse de l'agent

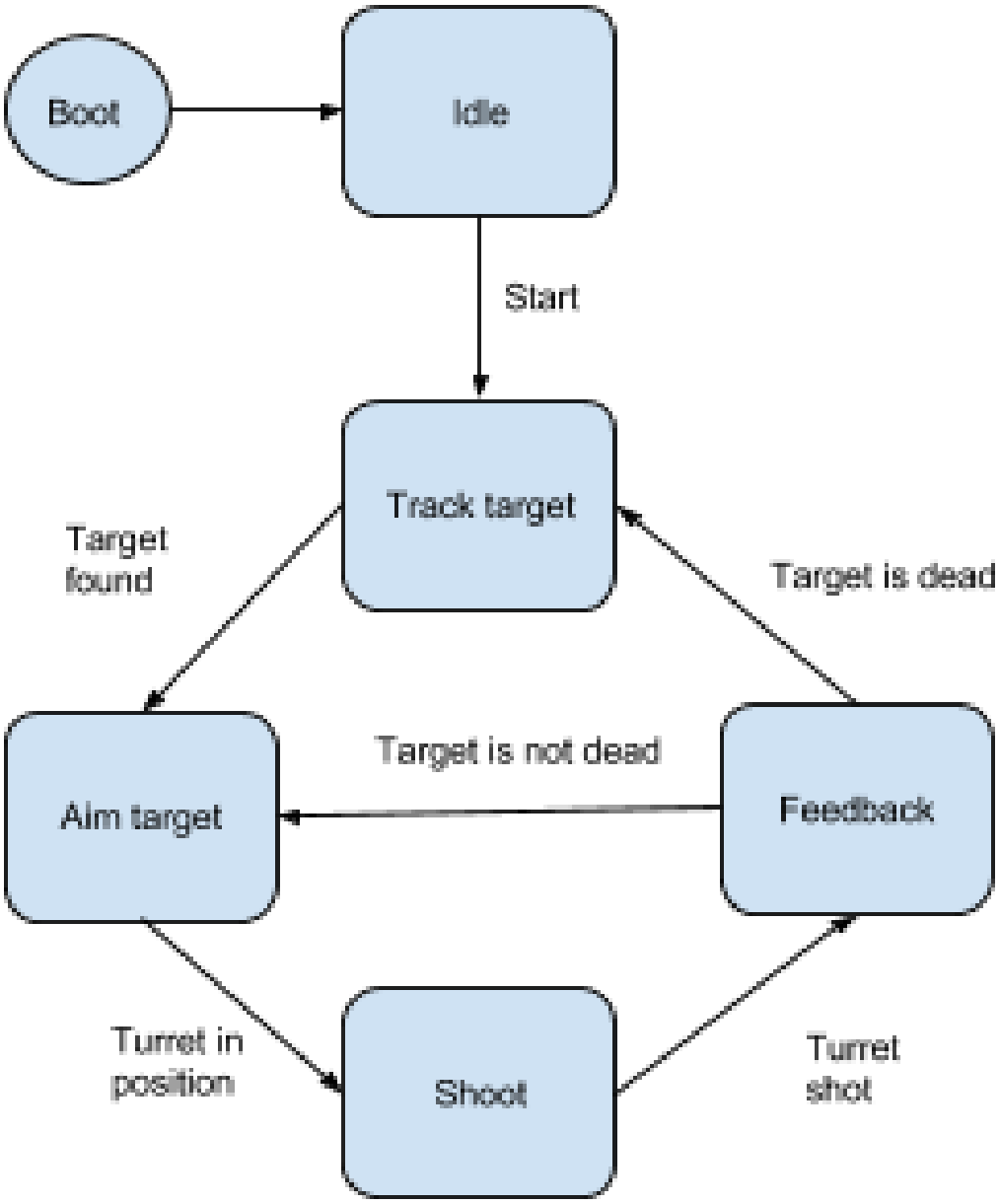
L'agent intelligent doit être une machine à état fini. Ce qui veut dire que l'agent à un nombre fini d'états qui sont prédéfinis et que pendant l'exécution de celui-ci, il ne peut y avoir de nouvel état inconnu qui s'ajoute. Pour répondre à ce besoin, le patron de conception « State » ou état s'y prête très bien. Ce patron est fait pour changer le comportement d'un objet sans avoir à « réinstancier » ledit objet.

C'est un design logiciel qui est utilisé dans l'industrie des jeux vidéos. Pour faire fonctionner ce patron, il faut identifier des états. Dans notre cas, cette intelligence artificielle à état comprend quatre états.

Voici les états de l'agent:

- TrackTargetState
- AimTargetState
- ShootState
- FeedbackState

Figure 3 - Schéma des états



Descriptions des états de l'agent intelligent

Track Target State

L'état « Track target » est l'état initial, lorsque le logiciel va démarrer, l'agent intelligent va tout de suite être paramétré pour être dans cet état. Cet état sert uniquement à recevoir l'appel de la vision pour lui transmettre des données. Ces données sont la position de la cible trouvée. La position comprend l'axe X et l'axe Y. Quand les données ont été envoyées par la vision, l'état garde les données dans une variable interne. Ensuite, un nouvel état est envoyé à l'agent intelligent. Cet état est « Aim target State ».

Aim Target State

Cet état est exécuté lorsque l'état « Track target state » a bel et bien reçu des données de la vision. Comme montré précédemment dans le diagramme d'architecture, il y a une connexion établie entre les moteurs de la tourelle et l'agent intelligent qui se trouve sur l'ordinateur portable. Donc, lorsque cet état est exécuté, les coordonnées X et Y trouvés par la vision sont envoyées au microcontrôleur et celui-ci prend en charge de faire les bons appels aux différents I/O pour faire bouger les moteurs dans à la position indiquée. Quand la tourelle de paintball est en position, il faut tirer sur la cible, pour ce faire un nouvel état est envoyé à l'agent intelligent soit « Shoot ».

Shoot State

Lorsque l'agent intelligent est dans cet état, il envoie des données au microcontrôleur qui s'occupe du moteur de la gâchette de la tourelle de paintball. Il est possible d'envoyer une commande pour dire de tirer qu'une seule fois ou il est possible de faire des tirs en rafales. Bien que cette option n'est pas incluse dans le développement, il est simple de l'ajouter dans une prochaine phase. Quand la tourelle a tiré, un nouvel état est lancé, celui du « Feedback state ».

Feedback State

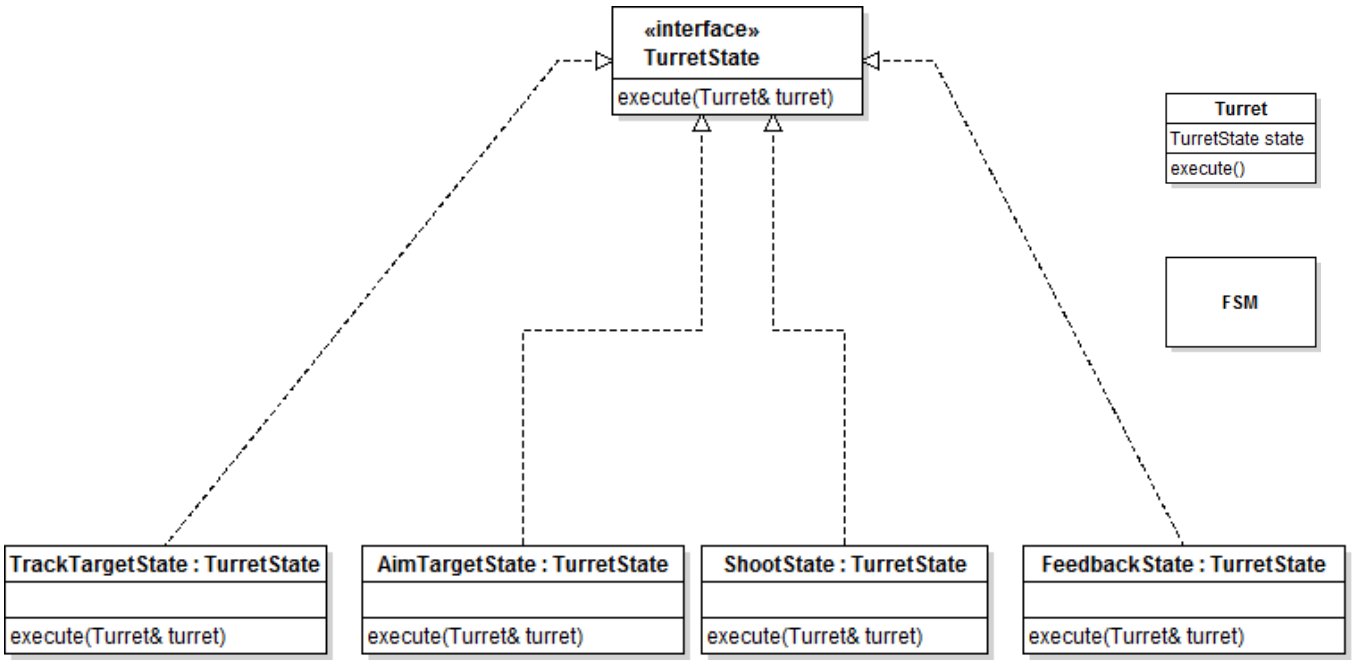
Cet état est le dernier, mais un des plus critiques. Cet état fait partie des recommandations et suggestion qui va être discuté un peu plus loin dans ce rapport. Le « Feedback state », sert à déterminer si la cible a bien été touchée. Si c'est le cas, alors un nouvel état est envoyé à l'IA pour lui signaler que la vision doit maintenant trouver une nouvelle cible. Si la cible n'a pas été touchée, alors il faut retrouver les coordonnées de la cible, viser de nouveau et ordonner un tir.

Implémentation de l'agent intelligent

Patron État

Pour implémenter le patron de conception « état », un diagramme de classe était nécessaire pour être sûr de ne pas oublier de partie et pour comprendre son fonctionnement général.

Figure 4 - Patron État



Comme le montre la figure 4, nous retrouvons les quatre états de la tourelle. Le fonctionnement du patron commence avec la classe abstraite « TurretState ». Cette classe permet le polymorphisme avec les états concrets. Les états concrets sont les états définis dans la section « Agent intelligent ». Chaque état à une méthode « execute » qui a un paramètre soit une instance qui pointe vers la classe « Turret ».

La classe « Turret » contient une instance de la classe abstraite « TurretState ». De plus, il y a une énumération des différents états disponibles dans l'objet « Turret ». C'est ainsi que lorsque l'état change, il faut appeler la méthode « setState(Enum state) » et c'est là que le nouvel état est associé à la variable « TurretState ». Ensuite, il suffit d'appeler la méthode « execute » de « Turret ». Cette méthode va faire un appel à « execute » du « TurretState » et c'est là que le paramètre « Turret » va être envoyé.

Voici un exemple d'utilisation :

```
//change the state to Aim
this->turret->setState(Turret::Aim);
//Execute the new state
this->turret->execute();
```

Détail de la méthode « execute » de la « turret » :

```
//execute the state with the turret parameter
this->state->execute(this);
```

Détail de la méthode « setState » de la « Turret »:

```
switch ( state )
{
        case Track:
                this->state = this->trackTargetState;
                break;
        case Aim:
                this->aimTargetState = new AimTargetState(this->trackTargetState->getTargetPosition());
                this->state = this->aimTargetState;
                break;
        case Shoot:
                this->state = this->shootState;
                break;
        case Feedback:
                this->state = this->feedbackState;
                break;
}
```

Problématique

Cependant, il y a un problème. L'algorithme de vision s'exécute en continu dans un « thread » séparé. Il fallait donc trouver une solution qui permet à l'état « TrackTarget » de savoir lorsque la vision repère une cible. Plusieurs solutions ont été essayées par contre, aucune n'était concluante et correspondait aux critères de performance, modularité et de fiabilité.

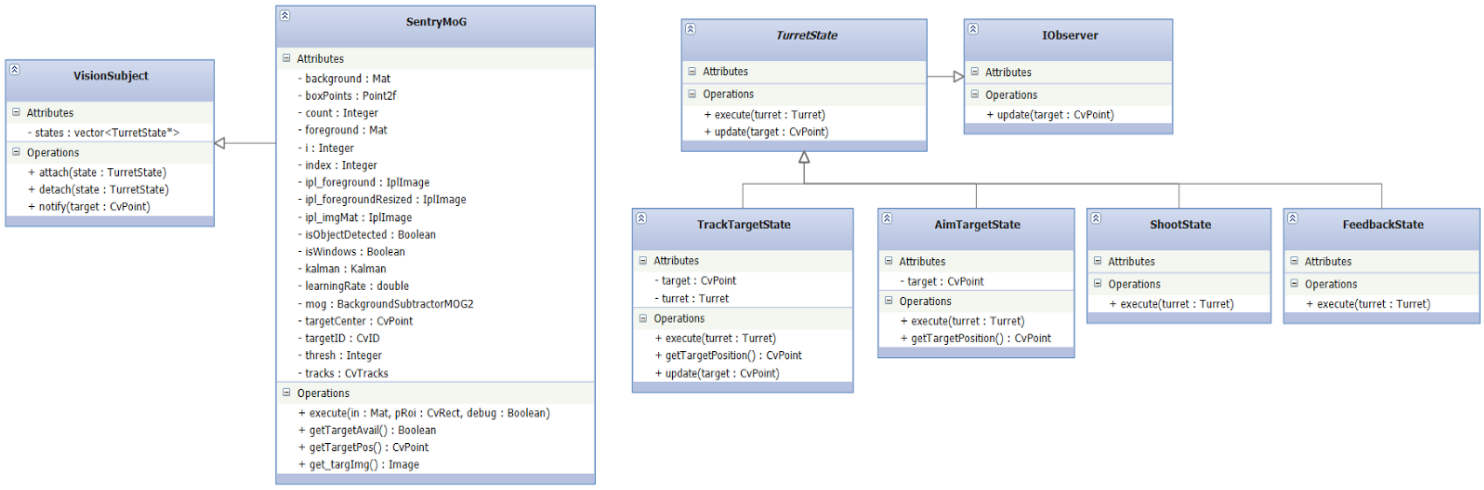
Finalement, l'approche utilisée a été d'incorporer un nouveau patron de conception. Soit le patron observateur.

Solution patron observateur

Celui-ci permet que lorsqu'il y a un évènement qui survient dans le code de notifier tous les observateurs de ce code. Dans le cas de notre agent intelligent de tourelle de paintball, lorsque la vision détecte une cible et que les coordonnées ont été traitées et sont prêtes à être envoyées, la vision notifie tout le monde qui l'observe et envoie les coordonnées.

Les classes qui peuvent observer la vision sont les états. Ce qui fait en sorte que l'état est notifié directement lorsqu'une cible a été repérée. Donc, on gagne en rapidité d'exécution, car il n'y a pas d'appel qui sont fait entre la notification de la vision jusqu'à l'état. C'est un chemin direct entre les deux.

Figure 5 - Patron observateur



Donc, nous pouvons voir que la classe « SentryMoG » a un héritage avec « VisionSubject ». C'est ce qui permet de notifier les états. La classe « TurretState » hérite maintenant de l'interface « IObserver ». C'est grâce à cette interface qu'il est possible de recevoir les notifications envoyées par le sujet.

Voici un exemple d'utilisation du patron observer dans le cadre du projet.

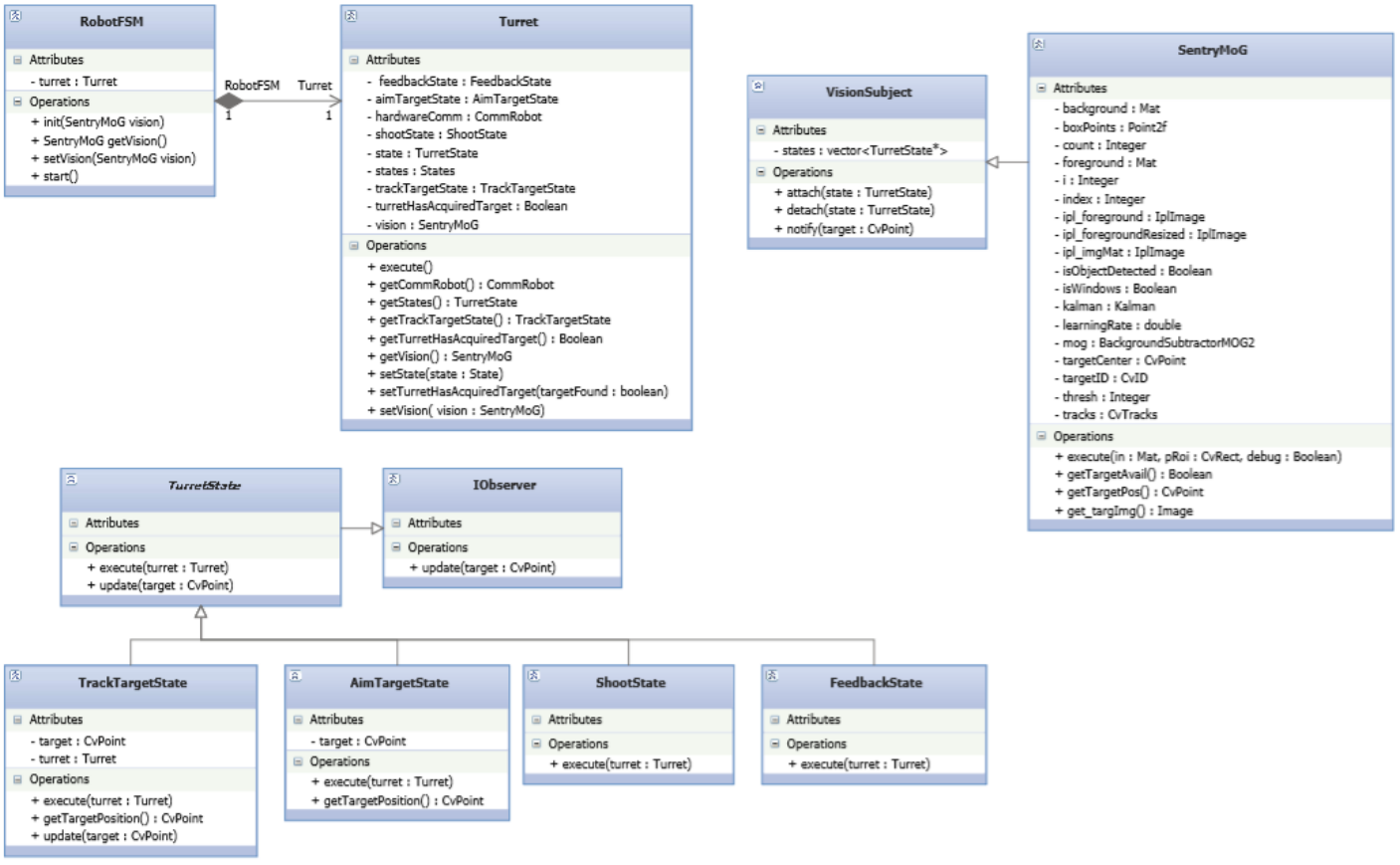
```
//attach an observer to the vision
vision->attach(this->turret->getTrackTargetState());

//here we can notify all the observer
notify(targetCenter);
```

Solution finale

Bref, c'est grâce à c'est deux patrons de conception qui travail de concert qu'il est possible de changer l'état de la tourelle de paintball. De plus, cette solution offre une façon facile de recevoir les messages de la vision qui est située dans un autre « thread ».

Voici maintenant le diagramme de classe de la solution globale

Figure 6 - Diagramme de classe de la solution finale



Grâce à toutes ces classes et aux deux patrons de conception inclus dans la solution, il est possible de contrôler la tourelle de paintball, et ce, de façon automatique. Pour y arriver, plusieurs interactions entre les différents modules sont nécessaires.

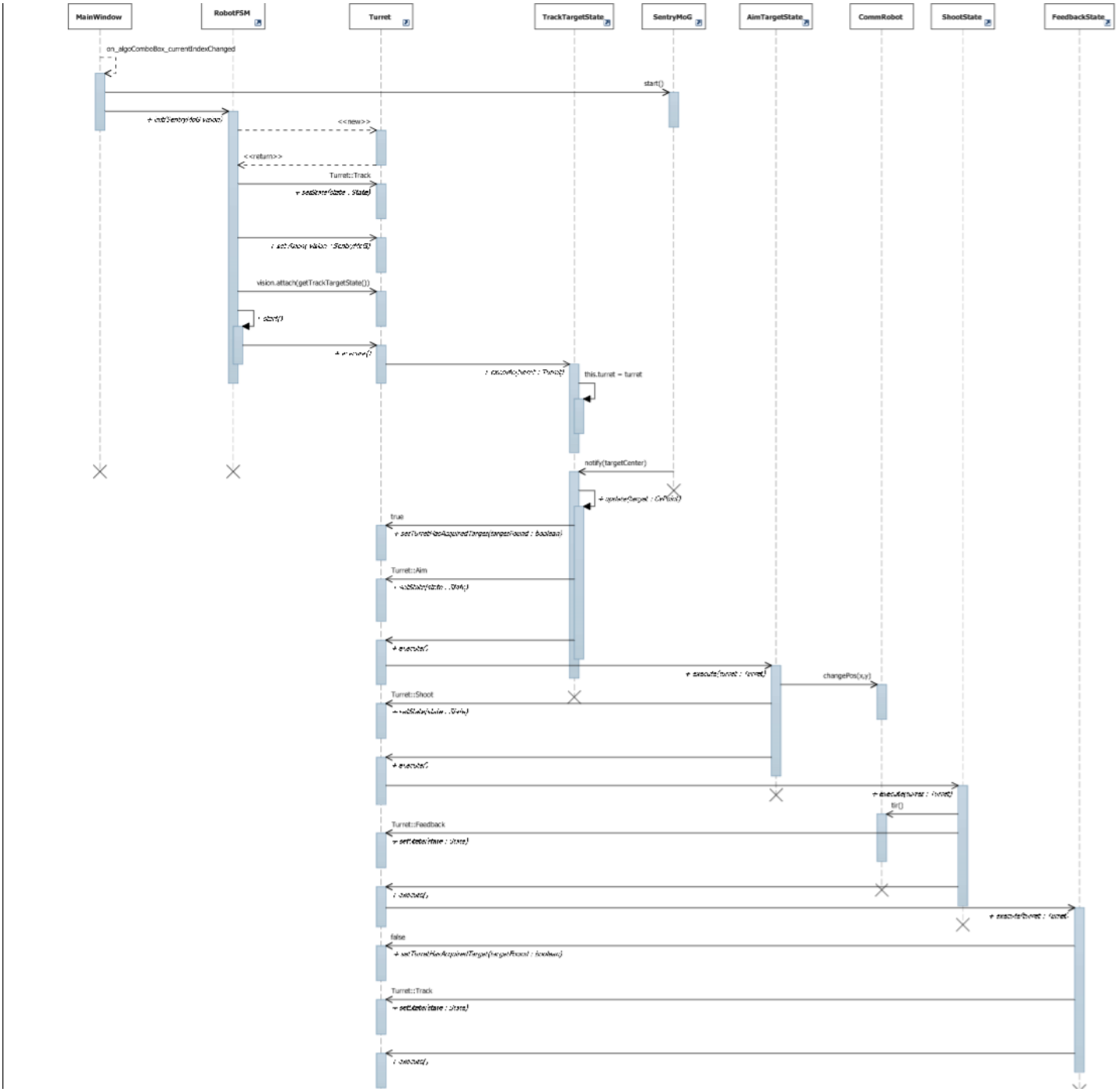
La classe « RobotFSM » permet d'englober la « Turret » et les quatre « State » pour être facilement utilisable dans l'ensemble du programme. C'est cette classe qui va s'occuper d'interagir avec la fenêtre « windows » du programme. De cette façon, l'intelligence artificielle se retrouve à une seule place et nécessite que de manipuler que « RobotFSM » pour lancer le traitement.

La classe « Turret » contient toutes les informations nécessaires au bon fonctionnement de la tourelle de paintball. Elle possède des instances d'objets et des méthodes pour faire des appels aux microcontrôleurs. De plus, elle a un pointeur vers l'algorithme de vision. Elle a aussi, les états qui lui sont propres.

La classe « SentryMoG » s'occupe de la partie vision de la tourelle de paintball. C'est dans cette classe qu'il y a plusieurs algorithmes et opérations qui sont effectués pour trouver une cible.

Voici un diagramme de séquence qui va permettre une meilleure compréhension de toutes les interactions entre les différentes classes dans la solution.

Figure 7 - Diagramme de séquence



Explication du diagramme de séquence

La séquence commence lorsque l'algorithme de vision est choisi dans le menu de l'application. L'objet qui s'occupe de l'interface fait appel à l'objet « RobotFSM » qui comme mentionné précédemment englobe tout l'agent intelligent. Cet appel permet d'initialiser l'agent intelligent et d'envoyer les informations de la vision à l'IA. De plus, c'est à ce moment que l'état initial est configuré, c'est-à-dire le « TrackTargetState ».

La prochaine étape c'est le patron observer qui entre en jeux. Donc, la vision va « attacher » un « observer ». Cet observer est l'état « TrackTargetState ». À partir de là, ils ont un lien qui les unit. Donc, quand la vision va trouver une cible, celui-ci va notifier son « observer » pour lui transmettre l'information.

À partir de ce moment, l'IA est en attente de la vision. C'est-à-dire, que temps que l'algorithme de vision n'a pas repéré de cible, l'agent ne va rien faire, il est en attente.

Lorsque la vision va repérer une cible, celui-ci va tout de suite notifier ses « observer ». C'est ainsi que le « TrackTargetState » va recevoir le message de la vision qui contient les coordonnées de la cible. C'est lorsque ce message est reçu que le reste de l'agent intelligent se met en marche.

À la réception du message, l'agent intelligent va bloquer les nouvelles cibles reçues par la vision pour se concentrer que sur la

première cible trouvée. Il va stocker l'information reçue et initier un nouvel état soit celui de « AimTargetState » qui permet de viser une cible. Dans cet état, l'agent intelligent va envoyer les commandes pour actionner les moteurs qui vont permettre de faire bouger la tourelle.

Ensuite un nouvel état va être lancé soit « ShootState », dans cet état, l'agent intelligent appelle le microcontrôleur qui s'occupe de la garcette du fusil de paintball pour l'actionner.

Finalement, l'état « FeedbackState » est lancé et dans cet état, le logiciel doit vérifier si la cible a bien été touchée ou non.

Recommandations pour le projet : Tourelle de paintball

État Feedback

Pour ce qui a trait au dernier état soit l'état « Feedback », il y a quelques enjeux importants. Pour que la tourelle soit vraiment intelligente, il faut que cet état soit capable de prendre des décisions. Pour ce faire, il faudrait un nouvel algorithme de vision qui serait en mesure de prendre en paramètre la cible repérée. Ensuite, il faudrait établir une façon de savoir si la cible a bel et bien été touchée.

Une façon serait de trouver une nouvelle couleur de peinture qui apparaîtrait sur la cible. Il suffirait de déterminer une couleur prédéfinie à utiliser dans l'arme de paintball. Avec cette couleur il serait possible d'utiliser des algorithmes de vision qui permettre d'isoler une couleur en

particulier. Donc, si la cible repérée a une nouvelle couleur qui apparaît sur lui à la suite de l'état « shootstate », l'agent intelligent pourrait déduire que la cible a bel et bien été atteinte et on pourrait passer à une nouvelle cible.

Grâce au patron observateur implémenter dans la solution, il est facile d'ajouter un nouvel algorithme de vision et ensuite d'attaché un observer pour qu'il puisse être notifié. Bref, il suffirait de notifier l'état « FeedbackState » pour que celui puisse prendre sa décision.

Nouveau mode

De plus, le patron observateur offre la possibilité d'ajouter beaucoup d'« observer » qui peuvent tous être notifiés en même temps. Un nouveau mode pourrait être ajouté à la tourelle soit un mode « plusieurs cibles » où lorsqu'une cible est repérée, l'agent intelligent envoie les commandes et les états pour tirer sur la cible, sauf que si une nouvelle cible est trouvée entre temps, on ajoute cette cible dans une liste ou dans un état « TrackTarget » différent à chaque fois et l'agent intelligent décide de ne pas s'attarder sur l'état « feedback ».

Donc, cela ferait en sorte que la tourelle essayerait de tirer plusieurs cibles un après les autres et finirait avec un état « feedback », mais seulement après avoir essayé d'atteindre plusieurs cibles.

Voici du code qui permettrait cette nouvelle implémentation

Liste d'état qui peut recevoir une notification :

```
std::vector<TurretState*> states;
```

Méthode qui permet de notifier plusieurs « observer » :

```
void VisionSubject::notify(CvPoint target)
{
        for(std::vector<TurretState*>::const_iterator i = states.begin(); i
!= states.end(); ++i )
        {
                if ( *i != 0 )
                {
                        (*i)->update(target);
                }
        }
}
```

Conclusion

Bref, dans ce projet, il y a eu plusieurs phases, soit l'analyse du besoin, la conception de l'agent intelligent, des recommandations pour le dernier état soit l'état « Feedback ». Toutes ces étapes ont assuré la réalisation de ce PFE.

L'agent intelligent répond au besoin soit la performance, la modularité et la fiabilité, ce qui fait qu'il est possible pour un ou des futurs étudiants de continuer l'amélioration de ce projet sans avoir à tout changer ce qui est déjà en place.

Le projet de fin d'études m'a permis de mettre en pratique les cours vues pendant le baccalauréat. Le cours LOG410 a été très utile pour bien cerner la portée et la vision du projet. Pour les patrons de conceptions, le cours LOG121 a permis de bien comprendre comment les incorporer dans le projet et de bien les utiliser.

Le génie pour l'industrie

Bibliographie

http://gameprogrammingpatterns.com/state.html

http://gamedevelopment.tutsplus.com/tutorials/finite-state-machines-theory-and-implementation--gamedev-11867

http://www.ai-junkie.com/architecture/state_driven/tut_state1.html

http://ai-depot.com/FiniteStateMachines/

http://unitygems.com/fsm1/

http://research.ncl.ac.uk/game/mastersdegree/gametechnologies/aifinitestatemachines/AI%20Tutorial%201%20-%20FSM.pdf

http://www.burgzergarcade.com/tutorials/game-engines/unity3d/108-unity3d-tutorial-mob-ai-fsm-part-1

https://www.scirra.com/tutorials/1139/how-to-simplify-your-ai-code-with-finite-state-machines

http://fr.wikipedia.org/wiki/%C3%89tat_%28patron_de_conception%29

http://fr.wikipedia.org/wiki/Observateur_%28patron_de_conception%29

Annexes

Annexe 1 : Code source l'application

```
#pragma once
#include "AI\turretstate.h"
#include <opencv/cv.h>

class AimTargetState : public TurretState
{
public:
	AimTargetState(CvPoint);
	~AimTargetState(void);

	virtual void execute (Turret* turret);
	CvPoint getTargetPosition();

private:
	CvPoint target;
};


#include "AimTargetState.h"
#include <iostream>
#include "Turret.h"
#include "ShootState.h"
using namespace std;


AimTargetState::AimTargetState(CvPoint target)
{
	this->target = target;
}


AimTargetState::~AimTargetState(void)
{
}

void AimTargetState::execute(Turret* turret)
{
	cout << "AimTargetState::execute()\n";
```

```
	cout << "Turret is now aimed at target...\n";

	turret->getCommRobot()->changePos(this->getTargetPosition().x, this->getTargetPosition().y);

	//attendre que les moteurs aient finis de bouger pour tirer
	//while (!turret->getCommrobot()->isInPosition());

	//change state to shoot
	turret->setState(Turret::Shoot);

	turret->execute();
}

CvPoint AimTargetState::getTargetPosition()
{
	return this->target;
}

#pragma once
#include "turretstate.h"
class FeedbackState :
	public TurretState
{
public:
	FeedbackState(void);
	~FeedbackState(void);

	virtual void execute(Turret* turret);
};


#include "FeedbackState.h"
#include "Turret.h"
#include <iostream>
using namespace std;


FeedbackState::FeedbackState(void)
{
}


FeedbackState::~FeedbackState(void)
{
}
```

```
void FeedbackState::execute(Turret* turret)
{
	cout << "FeedbackState::execute()\n";

	//reset turret target
	turret->setTurretHasAcquiredTarget(false);
	turret->setState(Turret::Track);
	turret->execute();


}

#pragma once
#include "turretstate.h"
class ShootState :
	public TurretState
{
public:
	ShootState(void);
	~ShootState(void);

	virtual void execute(Turret* turret);
};


#include "ShootState.h"
#include "Turret.h"
#include <iostream>
#include "FeedbackState.h"
using namespace std;


ShootState::ShootState(void)
{
}


ShootState::~ShootState(void)
{
}

void ShootState::execute(Turret* turret)
{
	cout << "ShootState::execute()\n";
	cout << "Shooting at target!\n";
```

```
		turret->getCommRobot()->tir();

		//change state to feedback
		//turret->setState(new FeedbackState());
		turret->setState(Turret::Feedback);
		turret->execute();
}

#pragma once
#include "AI\TurretState.h"
#include "AI\IObserver.h"
#include <opencv/cv.h>

class TrackTargetState : public TurretState
{

public:
		TrackTargetState(void);
		~TrackTargetState(void);

		virtual void execute(Turret* turret);

		//override from IObserver
		virtual void update(CvPoint);

		CvPoint getTargetPosition();

private:
		Turret* turret;
		CvPoint target;

};


#include "TrackTargetState.h"
#include <iostream>
#include <pthread.h>
#include "Turret.h"
#include "AimTargetState.h"
#include "Vision/SentryMoG.h"
using namespace std;


TrackTargetState::TrackTargetState(void)
{
}
```

```
TrackTargetState::~TrackTargetState(void)
{
}


void TrackTargetState::execute(Turret* turret)
{

	cout << "TrackTargetState::execute()\n";
	cout << "Waiting on vision for a target\n";
	this->turret = turret;
}

void TrackTargetState::update(CvPoint target)
{

	if ( !this->turret->getTurretHasAcquiredTarget() )
	{
		cout << "Target acquired\n";
		cout << "Changing state to Aiming\n";

		//target has been acquired
		this->target = target;

		//we can't find new target until this one has been shot
		this->turret->setTurretHasAcquiredTarget(true);

		//change state to Aim
		this->turret->setState(Turret::Aim);
		//Target is changing state to aim at the target based on the vision
recognition
		this->turret->execute();
	}
}

CvPoint TrackTargetState::getTargetPosition()
{
	return this->target;
}



#pragma once
#include <opencv/cv.h>
```

```
class IObserver
{
public:
	virtual void update(CvPoint){};
};

#pragma once
#include "AI\TurretState.h"
#include "AI\TrackTargetState.h"
#include "AI\AimTargetState.h"
#include "AI\ShootState.h"
#include "AI\FeedbackState.h"
#include <opencv/cv.h>
#include "Vision/SentryMoG.h"
#include "Controller/CommRobot.h"

class Turret
{
public:

	enum States {Track, Aim, Shoot, Feedback};

	Turret(void);
	~Turret(void);

	virtual void execute();

	virtual void setState(States state);

	virtual void setVision(SentryMoG* vision);

	virtual SentryMoG* getVision();

	virtual TurretState* getState();

	virtual CommRobot* getCommRobot();

	virtual TrackTargetState* getTrackTargetState();

	virtual void setTurretHasAcquiredTarget(bool);

	virtual bool getTurretHasAcquiredTarget();

private:
```

```
        class TurretState *state;
        SentryMoG* vision;
        CommRobot* hardwareComm;
        bool turretHasAcquiredTarget;

        AimTargetState* aimTargetState;
        ShootState* shootState;
        FeedbackState* feedbackState;
        TrackTargetState* trackTargetState;

};


#include "Turret.h"

Turret::Turret(void)
{
        this->trackTargetState = new TrackTargetState();
        this->shootState = new ShootState();
        this->feedbackState = new FeedbackState();
        this->hardwareComm = new CommRobot();
        this->setTurretHasAcquiredTarget(false);

        if ( this->hardwareComm->check() )
        {
                cout << "Turret is online and operating...\n";
        }
}


Turret::~Turret(void)
{
}

void Turret::execute()
{
        this->state->execute(this);
}

void Turret::setState(States state)
{

        switch ( state )
        {
                case Track:
                        this->state = this->trackTargetState;
                        break;
```

```
                case Aim:
                        this->aimTargetState = new AimTargetState(this-
>trackTargetState->getTargetPosition());
                        this->state = this->aimTargetState;
                        break;
                case Shoot:
                        this->state = this->shootState;
                        break;
                case Feedback:
                        this->state = this->feedbackState;
                        break;
        }

}

void Turret::setVision(SentryMoG* vision )
{
        this->vision = vision;
}

TurretState* Turret::getState()
{
        return this->state;
}

SentryMoG* Turret::getVision()
{
        return this->vision;
}

CommRobot* Turret::getCommRobot()
{
        return this->hardwareComm;
}

TrackTargetState* Turret::getTrackTargetState()
{
        return this->trackTargetState;
}

void Turret::setTurretHasAcquiredTarget(bool targetFound )
{
        this->turretHasAcquiredTarget = targetFound;
}

bool Turret::getTurretHasAcquiredTarget()
```

```
{
	return this->turretHasAcquiredTarget;
}

#pragma once
#include "AI\IObserver.h"

class TurretState : public IObserver
{
public:
	//constructor
	TurretState(void);
	~TurretState(void);

	//method
	virtual void execute(class Turret* turret){};
	virtual void update(CvPoint){};
};


#include "TurretState.h"


TurretState::TurretState(void)
{
}


TurretState::~TurretState(void)
{
}



#pragma once
#include <opencv/cv.h>
#include "Vision/SentryMoG.h"
#include "AI\Turret.h"

class RobotFSM
{
public:
	RobotFSM(void);
	~RobotFSM(void);

	virtual void init(SentryMoG* vision);
```

```
        virtual void start();

private:
                Turret* turret;
};

#include "RobotFSM.h"


RobotFSM::RobotFSM(void)
{
}


RobotFSM::~RobotFSM(void)
{

}

void RobotFSM::init(SentryMoG* vision)
{

        this->turret = new Turret();

        //observer pattern
        vision->attach(this->turret->getTrackTargetState());

        //first state
        this->turret->setState(Turret::Track);

        this->turret->setVision(vision);

        this->start();
}

void RobotFSM::start()
{

        //start turret intelligence
        this->turret->execute();

}
```

Département de génie logiciel et des TI

Document de vision

GTI792 PROJET DE FIN D'ÉTUDES EN GÉNIE DES TI

Tourelle de Paintball

Auteurs
RENÉ-ALEXANDRE GIROUX
GIRR19079003

Professeur superviseur
ALAIN APRIL

Date
1er novembre 2014

1. Introduction

1.1 Objectif

L'objectif de ce document est de bien comprendre l'ensemble du projet « Paintball Turret » pour identifier ce qu'il faut faire pour rendre la tourelle plus performante et fiable lors de son utilisation. La principale composante touchée va être l'intelligence artificielle. Soit le coeur de la tourelle, car, c'est cette partie qui va traiter les intrants de la vision et envoyer les commandes aux différents moteurs.

1.2 Portée

Le document de vision est pour toute personne qui va travailler sur le projet ou qui va utiliser la tourelle de paintball. Les améliorations vont être effectuées dans le cadre d'un projet de fin d'études à l'École de technologie supérieure (ETS).

1.3 Définition

- ETS - École de technologie supérieure
- IA - Intelligence artificielle
- AI - Artificial intelligence (traduction en anglais)
- BAC - Baccalauréat
- Génie LOG - Génie logiciel
- Génie - Ingénieur ou Ingéniérie
- PFE - Projet fin Étude
- Agent intelligent - Le centre de décision de la tourelle

2. Positionnement

2.1 Opportunité d'affaires

Puisque c'est un projet qui intéresse beaucoup de personne, il est possible faire travailler les étudiants qui sont en fin de BAC en génie logiciel pour apporter des améliorations et/ou modifications sur la tourelle au niveau informatique. De plus, puisque la tourelle comporte plusieurs parties soit mécaniques, électrique et informatique, il peut y avoir plus d'un type de PFE en même temps sur ce projet. Cependant pour ce document de visions nous allons nous arrêter à la partie logicielle.

2.1 Problème

Le problème de	la tourelle de paintball est que l'agent intelligent peut être mieux implanté et doit avoir certain critère tel que la performance, réutilisabilité et la fiabilité. L'agent intelligent
affecte	la façon que la tourelle prend ses décisions pour tirer sur ses cibles
l'impact est	qu'il faut que la tourelle puisse être rapide dans ses décisions et atteindre une cible
une bonne solution serait	de faire un agent intelligent sur le patron de conception « State » qui est utilisé dans l'industrie des jeux vidéo.

2.2 Position du produit

La tourelle de paintball est pour les personnes qui jouent au paintball et qui veulent avoir un niveau de compétition élevé pour éviter d'être repérer par la tourelle.

3. Partie prenante

3.1 Marché démographique

La tourelle de paintball est pour les joueurs de paintball qui veulent se battre contre un engin logiciel et mécanique

3.2 Sommaire des parties prenantes

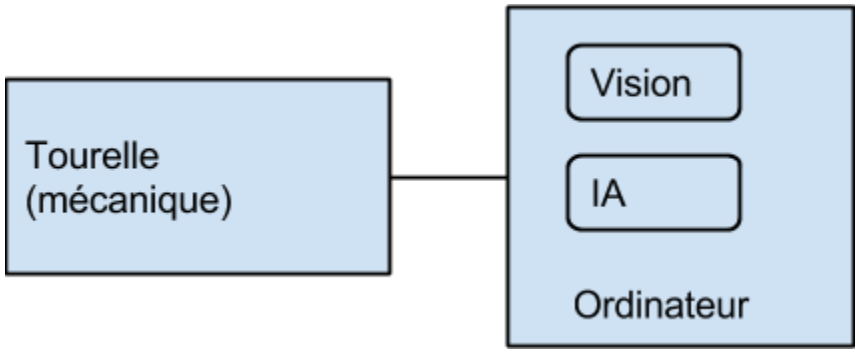
Nom	Descriptions	Responsabilité
Alain April	Professeur à l'ETS	Chef du projet : Tourelle de paintball
Stéphane Franiatte	Étudiant à la maîtrise à l'ETS	Concepteur et développeur du système informatique de la tourelle de paintball

3.3 Utilisateur

Nom	Description	Responsabilité
Joueur	joueur de paintball	joueur au paintball et se battre contre la tourelle de paintball

4. Vue globale du produit

4.1 Perspective du produit



4.2 Assumptions et dépendance

- L'ordinateur fonctionne avec une batterie longue durée
- Les algorithmes de vision permettre d'aller chercher une cible
- Le microcontrôleur est en place et permet des appels aux pièces mécaniques

5. Fonctionnalités du produit

5.1 Fonctionnalités

- Nouveau design de l'agent intelligent (patron de conception : État)
- Critère de performance
- Critère de réutilisabilité
- Critère de fiabilité

6. Contraintes

- Puisque c'est une application qui doit fonctionner sur un ordinateur portable sur une batterie longue durée, il faut que l'application ne demande pas trop d'énergie puisque les algorithmes de visions en prennent beaucoup, car, ils roulent en continu pour chercher des cibles.
- Le logiciel doit être développé en c++
- Visual Studio 2010 doit être utilisé
- Utilisation du microcontrôleur : RB-PHI67 et RB-PHI79

7. Autres requis du produit

7.1 Standard applicable

Le logiciel doit suivre les normes de programmation établie par l'ETS

7.2 Requis système

La solution doit pouvoir fonctionner sur l'ordinateur portable de la tourelle

7.3 Requis de performance

La solution doit être rapide dans son exécution pour ne pas manquer de cible

7.4 Requis de réutilisabilité

La solution doit être réutilisable facilement par un autre développeur dans le cadre d'un projet futur, et ce, sans avoir à refaire du développement inutile.

7.5 Requis de fiabilité

Puisqu'il peut y avoir des risques si le système d'intelligence artificielle tombe en erreur, la solution doit être fiable pour éviter à tout prix des problèmes.

Guide installation

Projet tourelle paintball

version 1.0

2014-11-28

Prérequis

- Visual Studio 2010
  - http://msdn.microsoft.com/en-us/library/dd831853%28v=vs.100%29.aspx
- Qt project 4.8.6
  - http://download.qt-project.org/official_releases/qt/4.8/4.8.6/qt-opensource-windows-x86-vs2010-4.8.6.exe.mirrorlist
- OpenCV 2.4.9
  - http://opencv.org/downloads.html

Installation

1. Installer Visual Studio 2010
2. Installer Qt project
3. Prendre la solution MultiCV sur le repository
4. Ouvrir la solution MultiCV
5. Clique droit sur la solution -> Qt Project settings
   - Changer la version pour 4.8.6
6. Installer OpenCv à l'aide de ce lien
   - http://stackoverflow.com/questions/10901905/installing-opencv-2-4-3-in-visual-c-2010-express

Annexe 4 : Explication du finite state machine

Turret AI

L'intelligence artificielle de la tourelle de paintball fonctionne sous le patron de conception « State Pattern ». Ce patron communément utilisé dans l'IA des jeux vidéo a été adapté pour le système de la tourelle. Le système comporte plusieurs états et transitions.

Voici la liste des états :

- TrackTargetState
- AimTargetState
- ShootState
- FeedbackState

Voici les transitions :

- Target found
- Turret in position
- Turret shot
- Target is not dead
- Target is dead

Lorsque le système démarre, l'agent intelligent se met dans un état « idle ». Quand l'algorithme de vision MOG Tracking va être choisi, l'agent intelligent va se mettre dans le mode « TrackTargetState ». Dans cet état, le logiciel va crée un nouveau « thread » qui va permettre d'aller chercher les informations sur la cible. Lorsque la cible va être trouvée, une transition « Target Found » va être lancée pour mettre l'agent dans l'état « Aim Target ». Cet état permet d'envoyer les commandes au microcontrôleur pour bouger les moteurs de la tourelle. Lorsque la tourelle va être en place pour tirer, une nouvelle transition va être lancé soit « Turret in position » cette transition va lancer l'état « Shoot ». Dans cet état l'agent envoie une commande au microcontrôleur qui contrôle la gâchette du fusil pour donner l'ordre de tirer. Finalement, l'état « Feedback » va être appelé après avoir tiré pour essayer de déterminer si la cible a été atteinte ou non. Si la cible n'a pas été atteinte, on retourne à l'état « Aim Target » pour bouger les moteurs pour viser la cible à nouveau. Si la cible a été atteinte, on retourne au premier état pour aller chercher une nouvelle cible.