Le Mans Université

Implémentation d'une interface action/événement pour un logiciel Windows de supervision

Guihaume PROST

Maître de stage : Michaël AZEVEDO

Tuteur : Jean-Hugh THOMAS

Rapport de 5A en vue de l'obtention
du diplôme d'ingénieur en Informatique

dans la

Filière ASTRE, ENSIM

28 août 2024

Résumé

Ce rapport décrit le travail effectué dans le cadre de mon stage chez Prynel, où j'ai été chargé de moderniser certaines fonctionnalités du logiciel Prynvision®. Le projet principal a consisté en l'introduction d'une nouvelle logique d'action/évènement, permettant une gestion centralisée et personnalisable de déclencheurs et d'actions au sein du logiciel. Pour cela, j'ai conçu et implémenté une série de classes métier en C++, intégrées ensuite dans une interface utilisateur développée avec MFC.

Un autre aspect important de ce stage a été l'amélioration de la gestion des cartes au sein du logiciel. En remplaçant les anciens plans gérés via un moteur de rendu 3D par l'affichage d'un set de tuiles permettant une implémentation plus légère. Cela étant réalisé en adaptant un système existant pour fonctionner localement sans dépendre de serveurs externes. J'ai alors pu améliorer l'expérience utilisateur en leur offrant une meilleure réactivité et flexibilité du logiciel.

Abstract

This report describes the work carried out as part of my internship at Prynel, where I was tasked with modernizing certain functionalities of the Prynvision® software. The main project involved the introduction of a new action/event logic, enabling centralized and customizable management of triggers and actions within the software. To achieve this, I designed and implemented a series of business classes in C++, which were then integrated into a user interface developed with MFC.

Another important aspect of this internship was the improvement of card management within the software. By replacing the old maps managed via a 3D rendering engine with the display of a set of tiles, enabling a lighter implementation. This was achieved by adapting an existing system to run locally without relying on external servers. I was then able to improve the user experience by making the software more responsive and flexible.

Table des matières

Remerciements

Je tiens à exprimer ma profonde gratitude envers la société TEB pour m'avoir offert l'opportunité d'effectuer ce stage au sein de leurs équipes. Cette expérience a été extrêmement enrichissante, tant sur les plans technique et professionnel que sur le plan personnel, et m'a permis de développer des compétences essentielles dans le domaine du développement logiciel et de la gestion de projets complexes.

Je souhaite également remercier tout particulièrement mon maître de stage, Michaël Azevedo, pour son soutien, sa patience et ses conseils avisés tout au long de cette période. Son expertise et son approche pédagogique ont été déterminants dans la réussite de ce projet, et il a su me guider à travers les défis techniques avec une grande disponibilité et bienveillance. Grâce à lui, j'ai pu acquérir une expérience précieuse et développer des solutions concrètes pour l'amélioration du logiciel Prynvision®.

Enfin, je remercie l'ensemble des collaborateurs de TEB pour leur accueil chaleureux et leur esprit d'équipe. Leur soutien et leur collaboration ont contribué à faire de ce stage une expérience mémorable et formatrice.

Chapitre 1

Introduction

Le logiciel Prynvision®, utilisé pour la gestion de parc de caméras, enregistreurs, alarmes, et autres dispositifs à travers plusieurs sites, est un outil robuste mais vieillissant. Conçu initialement sous une technologie ancienne, il est devenu lourd à maintenir car chaque nouvelle fonctionnalités doit lui être implémentée avec un code dédié, contrairement aux nouveaux logiciels du groupe. Dans ce contexte, mon stage au sein de Prynel avait pour objectif d'améliorer les fonctionnalités existantes du logiciel Prynvision® en introduisant une nouvelle logique d'action/évènement, et d'intégrer cette logique de manière cohérente et efficace en s'adptant à linterface utilisateur déjà mise en œuvre dans le reste du logiciel.

Ce rapport de stage détaille le processus d'implémentation de cette logique, depuis la conception des classes métier jusqu'à l'intégration de celles-ci dans l'interface graphique développée avec MFC (Microsoft Foundation Class). Il aborde également les défis techniques rencontrés, notamment l'implémentation de threads pour assurer la réactivité du logiciel, ainsi que l'ajout de fonctionnalités de cartographie permettant une meilleure gestion de certains événements grâce à une interface visuelle intuitive.

Chapitre 2

Présentation de l'entreprise

2.1 Création

La société TEB est fondée en 1978 par Louis Bidault. Sa création, le Tub Caméra®, lui permit de se faire remarquer au salon de l'agriculture par le président d'Auchan. Bien que son invention avait pour objectif de surveiller le bétail en utilisant une seule caméra se déplaçant le long d'un tube. Auchan lui donna une autre utilisation en l'implantant dans ses entrepôts et surfaces de vente. Cela leur permit de simplifier leur infrastructure de caméra ainsi que le suivi de personne par les opérateurs de surveillance. Ainsi la société TEB venait de trouver son premier client important. Cela permit de développer la société en continuant de proposer de nouveaux produits. Aujourd'hui Auchan est toujours un partenaire de la société tandis que le Tub Caméra® reste un incontournable du catalogue de produits TEB.

TEB est une PME française, familiale et indépendante. En 2023, elle enregistre un chiffre d'affaire d'environ 28 millions d'euros. Elle reverse 10% de celui-ci en recherche et développement, fonction réalisée par le bureau d'études, Prynel, rattaché au groupe. Prynel crée des produits sous la marque Pryntec filiale de TEB. C'est au total, environ 200 collaborateurs qui travaillent afin de réaliser et distribuer à l'échelle européenne différentes solutions hardware et software permettant la vidéosurveillance.

2.2 Marché actuel

Les différents clients du groupe sont présents dans plusieurs secteurs d'activité. Le marché du retail représente 50% du CA. Parmi les clients les plus importants de ce secteur on peut retrouver Auchan, Carrefour, Leclerc, Fnac-Darty, Leroy-Merlin... Ces clients utilisent les solutions de TEB pour surveiller leurs entrepôts de stockage ainsi que leurs points de vente.

Le deuxième secteur d'activité se trouve être le secteur bancaire. En effet, Prynel a développé des solutions de gestion de caméras et d'enregistrement. Mais ils ont aussi développé une analyse vidéo permettant de renforcer la sécurité ainsi en identifiant l'unicité d'une personne dans une pièce via traitement d'image ou alors des comportements suspects via IA.

Durant les dernières années, l'IA a pris une part importante dans les conceptions Prynel leur permettant d'acquérir de nouveaux marchés tels que le secteur autoroutier en créant des

solutions de comptage de personnes à bord des véhicules.

2.3 La gamme de produits proposée par la société

Le produit emblématique et toujours mis en avant par la société TEB est le Tub Caméra®. De part sa longévité, celui-ci a suivi un processus d'innovation incrémentale devenant aujourd'hui un produit abouti permettant le suivi de personnes. Cette tâche est donc réalisée par un seul produit et aide l'opérateur qui manipule une seule caméra se déplaçant dans un tube. Sinon, il devrait commuter entre plusieurs caméras et rendrait le suivi beaucoup plus complexe. De plus, le Tub Caméra® possède un mode "Ronde" i.e. lorsqu'il n'est pas piloté par un opérateur, le module fonctionne de manière autonome. Il se déplace le long du tube et filme de manière aléatoire certaines zones habituellement surveillées.

Ensuite viennent les solutions de gestion de parc de caméras car en général le Tub Caméra® est un produit qui ne fonctionne pas seul. Les industriels possèdent sur leurs sites tout un tas de différentes caméras. TEB leur offre la solution pour gérer ce parc de caméras en proposant le Digipryn®. C'est une solution d'enregistrement etde centralisation qui offre plusieurs fonctionnalités telles que le tracking, la détection d'intrusion, le franchissement de zone, etc... Le client peut alors centraliser l'enregistrement de toutes ses caméras quelque soit sa marque, son modèle ou sa technologie. Les Digipryn® sont installés localement et fonctionnent généralement seuls après installation.

Mais Prynel ne développe pas seulement des solutions hardwares. Ils développent notamment le logiciel Prynvision®, un client lourd installé sur une machine Windows permettant l'accès à un ou plusieurs Digipryn® ne se trouvant pas forcément sur le même réseau. Il est tout de même capable de se connecter en direct à des caméras individuelles sans passer par la solution de l'enregistreur vidéo. Cette solution permet à un opérateur de gérer plusieurs sites à distance en se connectant à des Digipryn® et à des caméras en direct. Bien qu'étant déjà plutôt complet, la gestion d'action/évènement présente sur d'autres logiciels du groupe manquait au Prynvision®.

Chapitre 3

Contexte du stage

3.1 Présentation de Prynvision®

Prynvision® est un client lourd destiné aux machines sous Windows. Elle permet une gestion centralisée d'un large éventail de caméras, enregistreurs, alarmes et disques durs répartis sur divers sites à travers le monde.

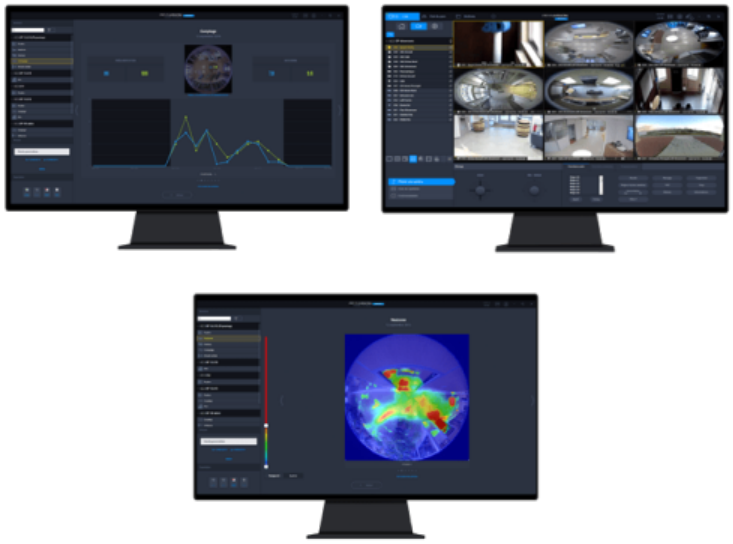

Figure 3.1 : *Interface Prynvision®*

Conçu pour être l'un des outils les plus complets et intuitifs de la gamme, Prynvision® souffre cependant de sa conception sur une technologie aujourd'hui dépassée. Sa complexité due à l'accumulation de fonctionnalités au fil des années, rend la migration vers une nouvelle technologie très coûteuse en termes de ressources. Chaque nouvelle fonctionnalité développée par Prynel doit alors être entièrement réintégrée dans Prynvision®, avec un code dédié.

En 2015, Prynel a décidé de lancer une nouvelle gamme de logiciels basés sur une architecture moderne et fonctionnant sur plusieurs systèmes d'exploitation. Cette nouvelle plateforme, appe-

lée Santenay, offre notamment une logique d'action/événement entièrement paramétrable via un serveur web, permettant à chaque client de personnaliser le comportement de son logiciel.

Toutefois, Prynvision®, étant antérieur à la création de la plateforme Santenay, ne bénéficie pas de cette base commune. La fonctionnalité d'action/événement, présente dans les nouveaux logiciels, manque cruellement au Prynvision®, rendant son utilisation moins intuitive. Pour pallier à ce manque et assurer une cohérence avec les autres solutions du groupe, il a été nécessaire d'implémenter cette logique en C++ en utilisant le framework MFC.

3.2 L'implémentation d'action/évènement

Une logique d'action/évènement dans un logiciel est un mécanisme qui réagit à des évènements spécifiques pour déclencher des actions prédéfinies. Ces évènements peuvent être de différentes natures comme un évènement externe tel qu'un problème sur une caméra ou une alerte générée par un Digipryn® vis-à-vis d'un enregistrement. Mais ces évènements peuvent aussi être internes au logiciel, telle qu'une connexion de l'utilisateur ou même des actions volontaires de celui-ci. Cette logique est un mécanisme simple à comprendre et qui peut répondre à beaucoup de types de besoin.

Actuellement, ce type de comportement n'est pas centralisé dans le logiciel, ce qui entraîne une implémentation parfois légèrement différente selon les objets nécessitant pourtant une logique similaire et une configuration moins intuitive pour l'utilisateur. D'autres objets, bien que potentiellement bénéficiaires, ne disposent pas de cette fonctionnalité, qui pourrait pourtant leur apporter de la valeur ajoutée.

L'implémentation d'une telle logique permettrait d'établir un lien entre les différents objets du logiciel. En créant une page dédiée à la gestion de ces actions, l'utilisateur pourrait s'approprier plus facilement ces fonctionnalités, rendant leur utilisation plus intuitive et efficace.

L'objectif est donc de centraliser et simplifier ce mécanisme pour garantir une expérience utilisateur plus homogène et cohérente.

3.3 Présentation de la technologie MFC

La technologie MFC (Microsoft Foundation Classes) est un ensemble de classes en C++ développées par Microsoft pour simplifier le développement d'applications Windows.

Figure 3.2 : *Logo MFC de 1992*

Introduites au début des années 1990, les MFC avaient pour objectif de faciliter le développement en encapsulant les API Windows (WinAPI) complexes dans des classes C++ plus accessibles. MFC reste une technologie intéressante pour les applications Windows en C++, bien qu'elle soit maintenant souvent remplacée par des technologies plus modernes comme .NET et les bibliothèques basées sur des technologies web dans les nouveaux projets. Il existe aussi d'autres frameworks récents tels que Qt permettant la réalisation d'interface en C++. Qt à l'avantage d'être multiplateforme tandis que le MFC se limite à Windows. Cependant, MFC continue d'être utilisé pour des applications existantes ayant commencé leur développement dans la fin des années 90 et nécessitant l'ajout de fonctionnalités de nouvelles fonctionnalités.

Chapitre 4

Développement de la logique d'action/évènement en C++

4.1 Développement de la classe action

4.1.1 Code métier

La création de la classe Action représentait la première étape du développement de cette fonctionnalité. Comprendre le fonctionnement de cette Classe est nécessaire pour mieux appréhender sa relation avec le code métier puis, dans un second temps, avec l'interface utilisateur : c'est pourquoi je vais détailler le fonctionnement de celle-ci. Cette logique sera ensuite appliquée à l'ensemble des développements réalisés pour Prynvision®.

Une Action est en réalité le code qui s'exécute en réponse à un événement spécifique. Dans la majorité des exemples, j'utiliserai l'événement "Journalier", qui déclenche l'exécution d'une action à une heure précise chaque jour. Toutefois, il peut arriver qu'une action doive accomplir plusieurs tâches simultanément. Par exemple, à une heure donnée, le Prynvision® pourrait devoir non seulement envoyer une notification sur le PC sur lequel il est installé, mais également envoyer un SMS à un numéro spécifique. C'est pourquoi, lors de la création d'une action, l'utilisateur aura la possibilité d'y intégrer plusieurs éléments, des actionnables. Chaque Action sera munie d'une liste d'actionnables possédant un type propre.

Ici on peut bien se rendre compte de l'intérêt de la programmation orientée objet qu'est le C++. Tous les actionnables héritent de la classe mère "Actionnable" et possèdent des attributs propres. Mais la classe Action n'a besoin de connaître que la classe Actionnable et les prototypes des méthodes *startActionnable*() et *stopActionnable*() pour fonctionner. L'avantage d'utiliser des classes qui dérivent d'Actionnable permet d'ajouter des actionnables assez facilement dans le projet, sans avoir pour autant à modifier le code d'Action.

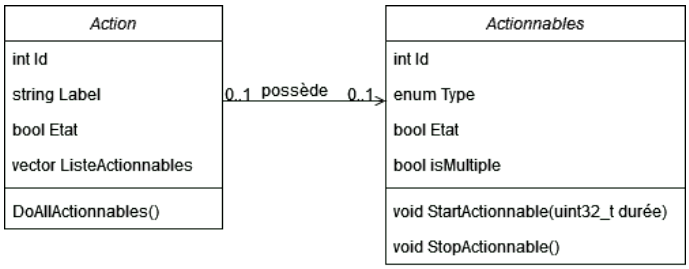

Figure 4.1 : *Diagramme de classe des actions*

Un évènement est associé à des actions, lorsqu'il se produit, il les déclenche en réalité via la méthode DoAllActionnables(). C'est cette méthode qui a pour objectif d'exécuter chaque élément de la liste des actionnables en appelant sa méthode StartActionnable(). Chacun est caractérisé par un type, un état actif ou non, et un booléen qui détermine s'il peut être exécuté plusieurs fois. En effet, certains actionnables, comme les notifications, peuvent être déclenchés plusieurs fois au cours d'une même action, tandis que pour d'autres, cela n'est pas souhaité. Le caractère multiple ou unique d'un actionnable est donc défini en fonction de son type.

4.1.2 Projet de test MFC

Pour commencer à exploiter le code métier de manière simple, il était nécessaire d'implémenter une interface en MFC. J'ai donc créé une page de configuration des actions, dont le but est de permettre à l'utilisateur de créer toutes les actions nécessaires à ses besoins, tout en leur attribuant une liste d'actionnables prédéfinis.

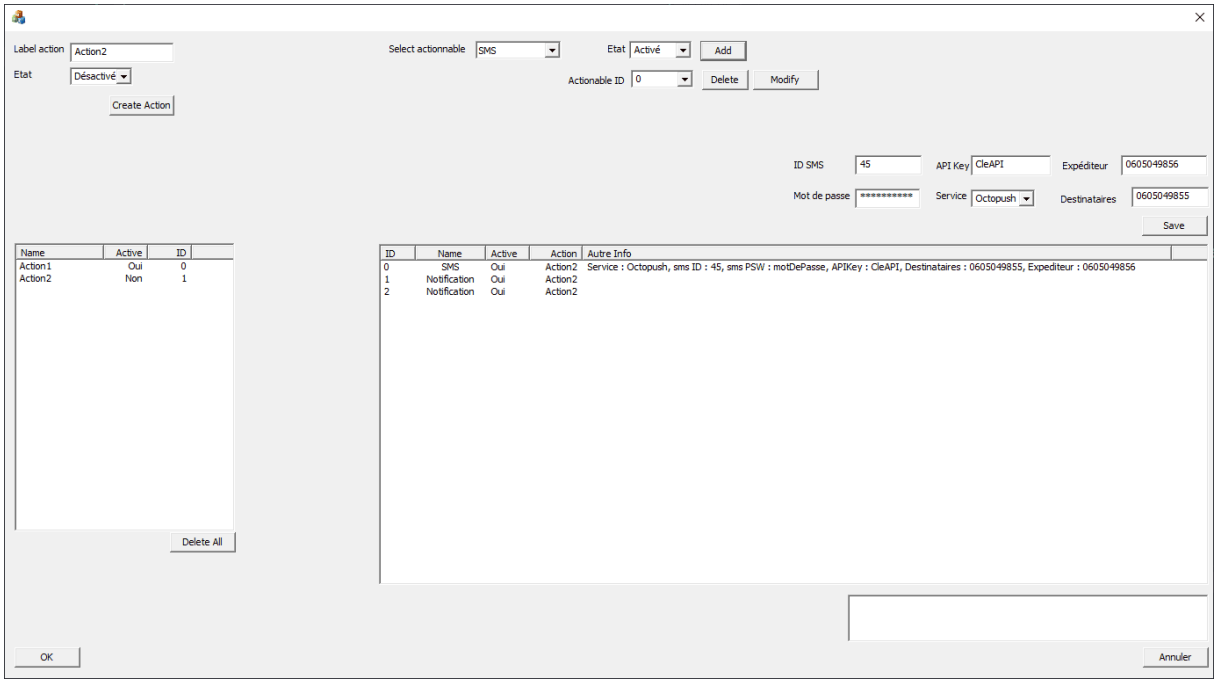

Figure 4.2 : *Première version de l'interface en pur MFC*

Sur la partie de gauche on retrouve la partie création de l'action, donc son label et son état

actif ou non définissable à l'aide d'une boîte combo (ComboBox). Une fois créée elle se retrouve dans un CListCtrl (composant MFC permettant de créer des Listes visuelles).

L'action est alors séléctionnable dans cette liste et on accède à sa liste d'actionnables paramétrable sur la droite de la page. L'interface permet d'ajouter chaque type d'actionnable et ensuite de lui affecter les paramètres spécifiques à son type. Par exemple, ici on retrouve le type SMS, qui a besoin du numéro de téléphone de destination, de séléctionner le service voulu pour l'envoyer, etc... L'interface développée est simple et peu esthétique, mais permet de manipuler tous les paramètres et méthodes des actions et donc de vérifier que l'on obtient un comportement similaire aux actions/évènements des autres logiciels Pryntec. De plus, cela permet de faire une première version de l'interface utilisateur et se rendre compte de l'agencement optimal des composants pour l'utilisation.

4.2 Développement de la classe évènement

Une fois la classe Action finalisée, il était nécessaire de passer à la partie permettant de déclencher ces actions : les événements. Un événement est constitué d'une liste de déclencheurs et d'une liste d'actions. Lorsque tous les déclencheurs sont validés, l'action est alors déclenchée. Pour configurer son événement, l'utilisateur a accès à une liste prédéfinie de déclencheurs, auxquels il devra attribuer les paramètres requis. Par exemple, pour un déclencheur basé sur l'heure, l'utilisateur devra spécifier l'horaire auquel il souhaite que la liste d'actions soit déclenchée. Parmi la liste des déclencheurs, on trouve des événements liés à l'état du disque dur de la machine, la connexion d'un utilisateur spécifique, etc.

Un événement peut accepter plusieurs actions, et une action peut être associée à plusieurs événements. De plus, chaque action doit pouvoir être exécutée sur une durée définie par l'utilisateur et liée à l'événement. Cela se traduit par le diagramme de classes suivant :

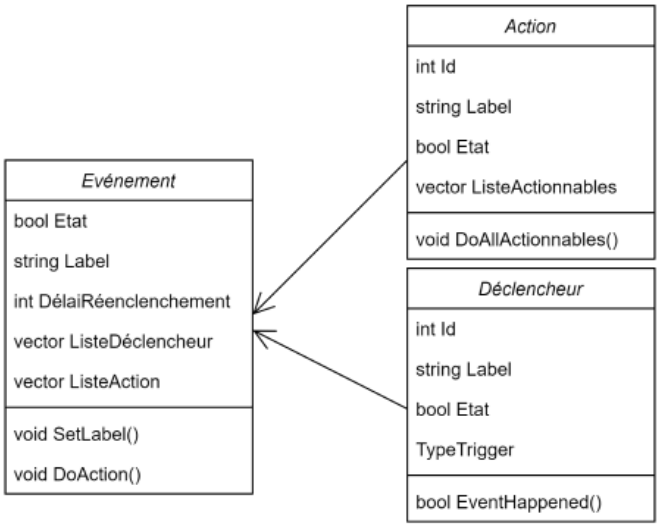

Figure 4.3 : *Diagramme de classe des évènements*

Le développement de cette classe fût assez rapide car après la création de la classe Action, ici on réutilisait peu ou prou les mêmes logiques. Les concepts orientés objets utilisés pour les

actionnables étaient donc réutilisables pour les déclencheurs. Ces concepts permettront, une fois le projet intégré à Prynvision®, d'ajouter facilement des déclencheurs sans avoir à modifier la gestion des actions/évènements. Je suis donc très rapidement passé sur le développement de l'interface utilisateur en y apportant de nouvelles contraintes.

4.2.1 Exemple d'implémentation d'un composant standard en MFC

L'utilisateur doit pouvoir choisir d'activer ou de désactiver certains événements temporairement. Alors, pour améliorer l'affordance de cette fonctionnalité dans l'interface, j'ai décidé d'utiliser un "toggle switch" (traduit par "Interrupteur à glissière"). Dans la plupart des interfaces ce bouton simple permet de définir l'état d'un booléen. Il donc que j'ajoute ce bouton à chaque ligne de la liste, et par la suite l'utiliser sur toutes les listes de mon interface permettant à l'utilisateur de définir un état d'un objet. Cependant, en MFC, les listes sont affichées à l'aide de ListCtrls qui ne supportent pas l'affichage d'autres éléments d'interface comme des boutons. Mais ils sont tout de même capables d'afficher des images.

J'ai donc créé deux images (bouton actif / bouton inactif) à insérer sur chaque ligne. Cependant, les ListCtrl n'acceptent par défaut que des images bitmap et uniquement dans la première colonne. Pour contourner cette limitation, j'ai d'abord converti mes images PNG en bitmap, puis j'ai créé une liste de bitmaps à associer au ListCtrl.

Après avoir effectué cette transformation, j'ai modifié le comportement standard du composant afin qu'il accepte des images dans une colonne autre que la première. Tandis que pour la première colonne, j'ai spécifié au ListCtrl d'afficher une image à un index situé en dehors de la liste d'images associées au composant permettant d'afficher une image vide. Bien que cela laisse un espace devant le texte de la première colonne, le résultat reste satisfaisant, car il permet à l'utilisateur de modifier rapidement l'état d'une action ou d'un événement.

En effet, grâce à la possibilité de récupérer l'index de la colonne cliquée ainsi que la ligne associée, il devient possible de changer l'image en cliquant directement dessus. Cela crée ainsi un comportement similaire à celui d'un "toggle switch". Étant donné la complexité liée à la modification du comportement des classes de base de MFC, je suis satisfait du résultat obtenu. Cependant, cet exemple nous montre les limitations liées à un framework vieillissant comme MFC. Dans la plupart des frameworks récents comme Qt, il est plus simple d'ajouter d'autres éléments d'interface à des listes.

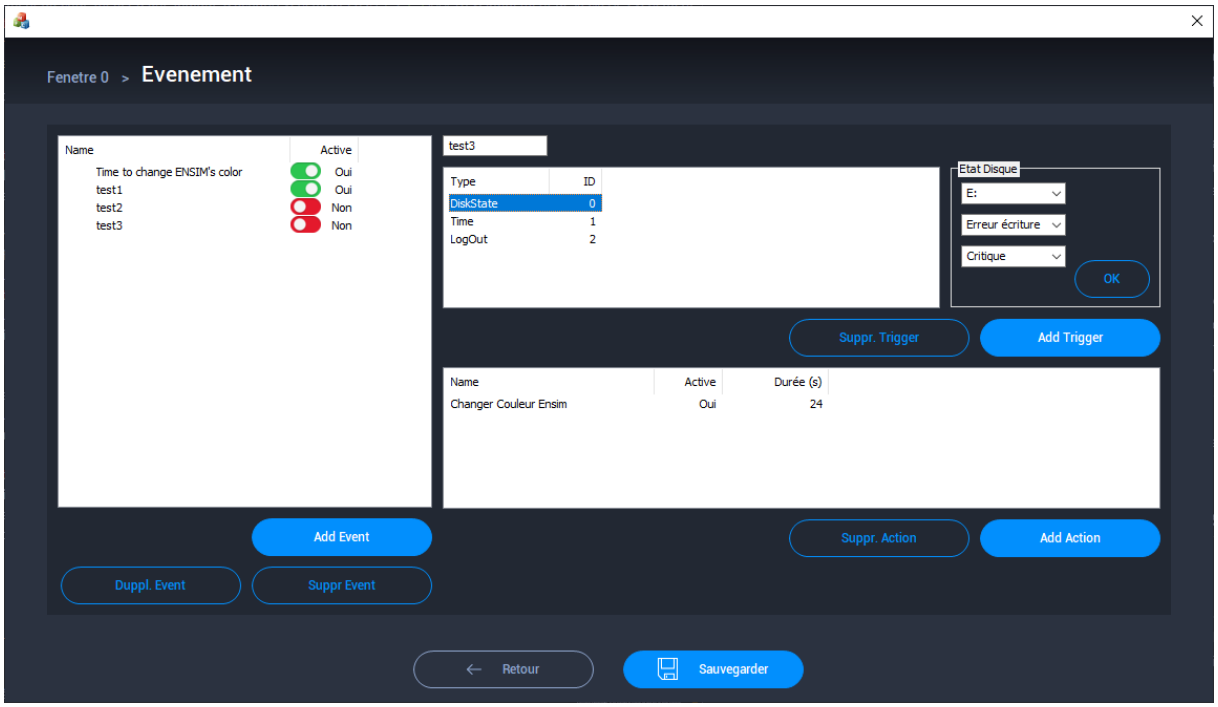

Figure 4.4 : *Interface actuelle avec "toggle switch"*

Chapitre 5

Programme de test en MFC

5.1 Présentation de CDialog

CDialog est une classe MFC qui représente une boîte de dialogue dans une application Windows. Ces boîtes de dialogue sont des fenêtres permettant l'interaction de l'utilisateur avec l'application. Elles sont couramment utilisées pour recueillir des informations, afficher des messages ou permettre la configuration de paramètres.

Voici à quoi ressemble la création d'une boîte de dialogue CDialog :

1. Définition de la boîte de dialogue dans une ressource : La première étape consiste à créer une ressource de boîte de dialogue dans un fichier de ressources (.rc). En général, cela se fait à l'aide de l'éditeur de ressources intégré à Visual Studio, qui permet de glisser-déposer des éléments d'interface dans une fenêtre. Cette approche facilite l'édition du fichier de ressources sans nécessiter de code manuel.

```
IDD_MYDIALOG DIALOGEX 0, 0, 320, 200
STYLE DS_SETFONT | WS_POPUP | WS_VISIBLE | WS_CAPTION | WS_SYSMENU
CAPTION "My Dialog"
FONT 8, "MS Sans Serif"
BEGIN
    DEFPUSHBUTTON   "OK",IDOK,209,7,50,14
    PUSHBUTTON      "Cancel",IDCANCEL,209,24,50,14
    EDITTEXT        IDC_MYEDIT,7,7,100,14,ES_AUTOHSCROLL
    LTEXT           "My Static Text",IDC_STATIC,7,30,100,8
END
```

2. Déclaration de la classe de boîte de dialogue. Il faut alors créer une classe dérivant de CDialog :

```
class CMyDialog : public CDialog
{
public:
    CMyDialog(CWnd* pParent = nullptr); // constructeur standard

    // Dialog Data
    enum { IDD = IDD_MYDIALOG };

```

```
protected:
    virtual void DoDataExchange(CDataExchange* pDX); // DDX/DDV support

    // Implementation
protected:
    DECLARE_MESSAGE_MAP()
};
```

C'est de cette manière que toutes les pages du logiciel Prynvision sont créées, car elles dérivent toutes de CDialog. En pratique, elles dérivent de CDialogFenetre, une classe développée par Prynel, qui étend CDialog et permet de créer une page standard pour le logiciel, incluant par exemple une surcouche pour ajouter un fond de fenêtre. L'avantage d'avoir une fenêtre de base à partir de laquelle toutes les autres dérivent est que cela assure une cohérence visuelle et fonctionnelle au sein du logiciel, avec un thème et une interface uniformes, ainsi que des méthodes pratiques (comme l'ouverture d'une fenêtre fille ou le retour vers la fenêtre parente, l'ajout de boutons, etc.).

Ainsi, il suffit de modifier une seule page pour introduire des nouveautés dans l'interface ou même appliquer un nouveau thème de couleur.
Exemple :

```
class CPrynvisionActionFenetre : public CDialogFenetre
{
    DECLARE_DYNAMIC(CPrynvisionActionFenetre)

public:
    CPrynvisionActionFenetre(CWnd* pParent = NULL); // constructeur
        standard
    virtual ~CPrynvisionActionFenetre();
```

En réalité, mon maître de stage m'a fourni un code de démonstration qui m'a servi de base pour comprendre comment les fenêtres sont conçues dans Prynvision®. Grâce à cet exemple, j'ai pu rapidement créer une fenêtre similaire à celles du logiciel dans sa version actuelle. Toutes les fenêtres que j'ai développées par la suite sont dérivées de cette fenêtre d'exemple, devenant ainsi des fenêtres filles de celle-ci. Concrètement, on accède à mes fenêtres via un bouton sur la page d'exemple, et un retour depuis l'une de mes fenêtres renvoie l'utilisateur à la fenêtre d'origine.

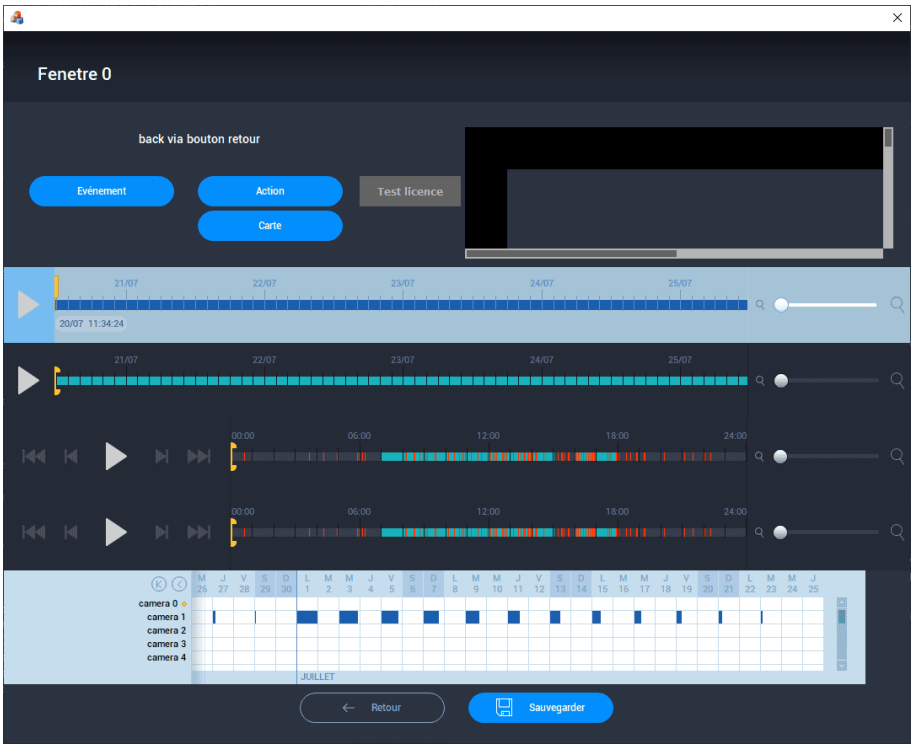

FIGURE 5.1 : *Fenêtre de démonstration pour utiliser l'UI officielle du Prynvision®*

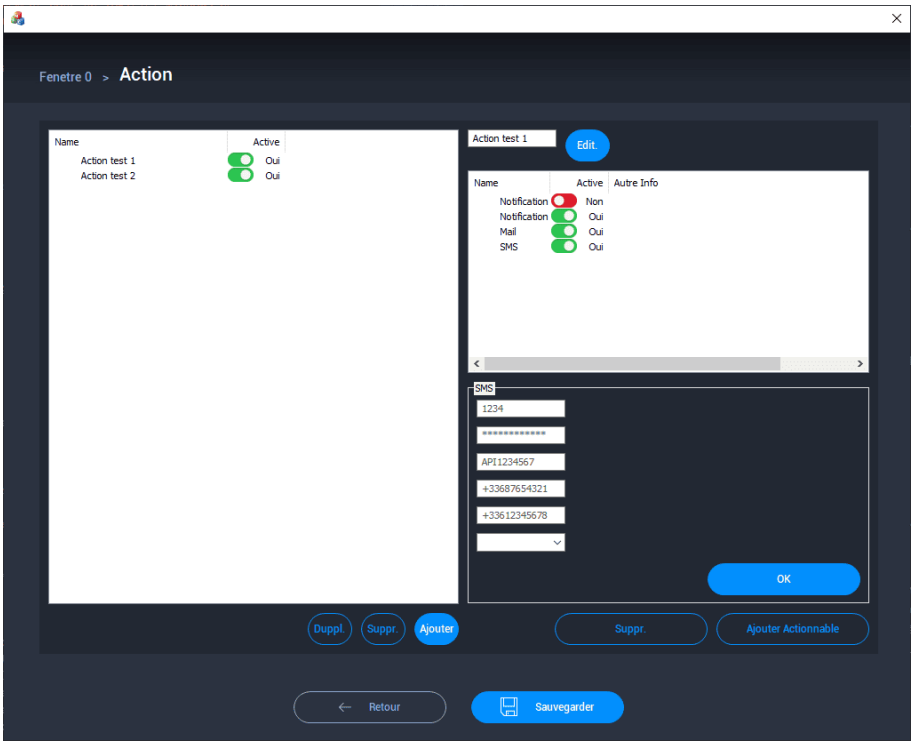

FIGURE 5.2 : *Page action, fille de la fenêtre de démonstration*

5.2 Utilisation des threads

Étant donné que j'ai développé un système d'action/événement, il était nécessaire de mettre en place un mécanisme pour vérifier l'état de chaque événement. Ce vérificateur a pour objectif de s'assurer que les conditions de déclenchement de chaque événement sont bien remplies. Si elles le sont, l'événement est considéré comme déclenché, et il devait alors exécuter les actions associées. Cependant, certaines de ces actions, comme l'envoi d'un SMS ou d'un e-mail, peuvent prendre du temps. De plus, le changement temporaire d'une icône, que nous aborderons plus tard, peut également ajouter des problèmes. Par exemple, si l'on souhaite changer une icône pendant 5 secondes, le code associé impliquera nécessairement un délai de 5 secondes, ce qui pourrait bloquer l'application pendant cette période.

Pour éviter ce type de blocage, il est nécessaire d'utiliser des threads. Un thread est une séquence d'instructions qui peut être exécutée indépendamment des autres, permettant de réaliser plusieurs tâches simultanément au sein d'un même programme. L'utilisation de threads améliore l'efficacité et la réactivité des logiciels, en permettant par exemple de gérer l'interface utilisateur tout en traitant des données en arrière-plan, ce que nous allons faire.

J'ai donc implémenté ces threads de manière à ce que le gestionnaire d'événements et les actions qu'il déclenche n'interfèrent pas avec la navigation dans l'application. Pour ce faire, j'ai créé la classe EventManager, qui est instanciée au démarrage de l'application. Cette classe lance un thread unique qui vérifie les déclencheurs de chaque événement, exécute les actions nécessaires, puis les arrête après la durée définie. Ce processus n'est pas continu ; en réalité, le thread reste inactif la plupart du temps et ne se réveille que toutes les 500 ms pour effectuer ses vérifications.

Sans cette pause, le thread effectuerait ses vérifications en boucle, ce qui pourrait entraîner une utilisation excessive de l'un des cœurs du processeur, le sollicitant inutilement à 100%. Comme nous n'avons pas besoin d'un niveau de réactivité extrêmement élevé, une fréquence de vérification toutes les 500 ms est largement suffisante pour détecter si un événement a eu lieu. Cela permet d'éviter une surcharge inutile du processeur, préservant ainsi les ressources de la machine.

5.3 Présentation des patterns utilisés

5.3.1 Les Singletons

Nous avons vu précédemment que l'EventManager était une classe lançant un thread au démarrage du programme. Ce thread teste toutes les configurations d'évènements qui eux-mêmes sont liés aux actions et déclencheurs. Mais le logiciel permet à l'utilisateur de modifier ces trois types d'objet. Il faut alors trouver une méthode permettant d'accéder à l'instance de l'EventManager instanciée au début du programme pour l'avertir de la modifiaction d'un des objets précédents. Cette méthode est de faire de cette classe un Singleton. Le Singleton garantie l'instanciation unique, ce qui permettra que sur chaque page du logiciel, lors de la sauvegarde de chaque configuration d'action, d'évènement ou de déclencheur, on puisse avertir l'EventManager qu'il doit mettre à jour ses listes d'objets.

Au moment de la sauvegarde, on obtient alors l'instance unique de l'EventManager de la manière suivante. Ensuite on appelle la méthode $UpdateLists()$ qui permet à l'EventManager

de se mettre à jour.

```
EventManager& eventManager = EventManager::getInstance();
eventManager.UpdateLists();
```

Le Singleton nous permet alors d'accéder n'importe où dans le programme à son instance tout en s'assurant qu'elle est unique.

5.3.2 Les Publisher/Subscribers

Dans la suite du développement nous utiliserons une carte qui sera intégrée comme un composant de notre application. Cette carte possède des méthodes et des variables propres. L'EventManager sera amené à modifier celle-ci avec des actions permettant de modifier les icônes présentes par exemple. Sauf que le manager ne peut pas accéder directement à l'instance de la carte. Donc il ne peut pas modifier directement les icônes. De plus, on ne va pas non plus créer une méthode de la classe de carte permettant d'aller vérifier à intervalle régulier si les icônes doivent être modifiées ou non, car cela n'arrive pas souvent et cela prendrait beaucoup de ressources pour rien. C'est pour cela que l'on fait appel au pattern Publisher/Subscribers.

Le Publisher/Subscriber est un modèle de conception dans lequel des objets, appelés "publishers" envoient des messages sans connaître les récepteurs. Les objets "subscribers" s'inscrivent pour recevoir les messages.

Dans ce cas, la carte peut être informée qu'elle doit mettre à jour ses données lorsqu'elle reçoit un message envoyé par l'EventManager. De plus, le plan n'est pas toujours instancié tout au long de la durée de vie du programme. Lorsqu'on accède à la fenêtre du plan, celui-ci s'abonne alors à l'EventManager, en l'obtenant via le Singleton. Lorsque l'utilisateur quitte la fenêtre, le plan se désabonne soit manuellement, soit automatiquement. Dans notre programme, le désabonnement se fait grâce à une utilisation judicieuse des pointeurs intelligents en C++.

5.3.3 Le Delegate Pattern

Le modèle Delegate Pattern a une approche similaire au Publisher/Subscribers dans la mesure ou la classe délégante ne connaît pas les détails de l'implémentation des tâches qu'elle délègue. Elle se contente d'appeler des méthodes sur la classe déléguée.

Pour implémenter le pattern de délégation en code, on crée généralement deux classes. Une classe déléguée qui effectue réellement le travail et une classe délégante qui délègue le travail à la classe déléguée. La notion d'interface n'existe pas en C++, on utilise donc une classe virtuelle pure dont on dérive pour faire un travail similaire.

L'intégration de la carte dans la fenêtre de notre application permet à l'utilisateur d'interagir avec celle-ci. L'utilisateur peut naviguer sur la carte, zoomer, déplacer des marqueurs, etc. Cependant, ces interactions ne sont initialement liées qu'à la classe de la carte affichée dans la fenêtre. Or, il peut être nécessaire d'accéder à ces informations pour ajouter des éléments à la page, comme l'affichage du niveau de zoom actuel, par exemple.

La classe *MapDelegate* est créée par la classe *CPrynvisionMapFenetre*, et elle doit lui transmettre certaines informations. Bien que l'on puisse inclure le fichier header de CPrynvision-MapFenetre dans le code de *MapDelegate*, cela créerait une dépendance directe entre les deux

objets. Pour éviter cette liaison, *MapDelegate* transmet ses informations à un délégué qui n'a besoin d'implémenter que certaines fonctions spécifiques, sans connaître les détails de la classe. Dans ce cas, ce délégué est la classe qui a créé *MapDelegate*, à savoir *CPrynvisionMapFenetre*.

Voici la classe délégante avec une de ses méthodes permettant de renvoyer le niveau de zoom :

```
class MapDelegate {
public :
    virtual void SendCurrentZoomLevel(int ZoomLevel) = 0;
    ...
```

Ensuite on dérive de MapDelegate la fenêtre contenant l'affichage de la carte. Alors quand la carte réalise la méthode MapDelegate : :SendCurrentZoomLevel(), la fenêtre exécute sa surcharge de cette méthode et permet d'afficher le niveau de zoom actuel n'importe où sur son interface :

```
class CPrynvisionMapFenetre : public CDialogFenetre, public MapDelegate{

    virtual void SendCurrentZoomLevel(int iZoomLevel) {
        CString cStaticZoomLevel = " Zoom Level : " + iZoomLevel; //
            String permettant d'afficher dans l'interface le niveau de
            zoom actuel
    };
}
```

Ce mécanisme permet alors à n'importe qu'elle classe dérivant de MapDelegate de recevoir les informations du gestionnaire de carte au moment où celui-ci décide de les envoyer.

5.4 La sérialisation

Le code que j'ai réalisé permet à l'utilisateur de créer des configurations pour des actions, des événements et par la suite des configurations de plan. Mais toutes ces configurations ne doivent pas être perdues à chaque fois que le logiciel est fermé. Il faut alors pouvoir sauvegarder ces données dans des fichiers.

Pour ce faire, Pryntec® utilise toujours le même processus. La sérialisation de ces données dans un buffer d'entiers non signés sur 8 bits. Ce buffer est ensuite écrit dans un fichier binaire. Pour en extraire les données, on réalise le processus inverse et récupère les données du fichiers dans un buffer qu'on désérialise. Donc il existe une classe fournie par Prynel : *Base_Serialisable_Object* possédant trois méthodes à surcharger pour chaque objet que l'on veut sauvegarder.

La manipulation de ces méthodes est un peu rude au départ, car il faut faire attention à bien faire concorder le décalage (offset) pour la lecture et l'écriture. Mais après plusieurs implémentations cela devient plutôt simple. L'avantage de cette méthode est que l'on gagne de l'espace disque comparé à un JSON qui ne stocke que du texte. Ce même JSON demanderait d'implémenter un parser ce qui alourdirait encore la gestion des sauvegardes. De plus le fait d'avoir créé des classes de base telles qu'*Actionnable* dont les actionnables dérivent, c'est que lors de la sérialisation, on appelle simplement la sérialisation pour chaque actionnable et que celle-ci est gérée automatiquement en focntion de son type.

Chapitre 6

Ajout d'un contrôleur de carte

6.1 Recherches sur les tuiles vectorielles

L'ajout d'un nouveau système de cartographie était requis pour Prynvision, l'actuel étant limité par son architecture. J'ai donc entrepris des recherches pour déterminer la meilleure solution afin de gérer les cartes de manière simple et locale sur la machine. Durant cette analyse, j'ai découvert des méthodes modernes de gestion de cartes, notamment l'utilisation des cartes vectorielles. Contrairement aux cartes traditionnelles, qui sont stockées sous forme de mosaïques d'images, les cartes vectorielles sont plus légères. Elles représentent les données géographiques non pas par des images brutes, mais par des formes géométriques (points, lignes, polygones) définies par des coordonnées mathématiques, offrant donc une plus grande flexibilité et une réduction de la taille des fichiers.

Cependant, les cartes vectorielles nécessitent une manipulation complexe pour être utilisées, notamment la génération de fichiers PNG en temps réel pour l'affichage avec MFC. En comparaison, les cartes utilisant des tuiles brutes, qui consistent en une division de la carte en plusieurs petites images carrées assemblées comme une mosaïque, sont plus simples à gérer pour notre application. Bien que ces images brutes prennent plus d'espace sur le disque, elles vont se révéler plus efficaces pour notre usage.

6.2 Fonctionnement du Système de Cartographie

6.2.1 Structure des Tuiles

Pour illustrer le fonctionnement d'un système de cartographie avec des tuiles brutes, prenons le niveau de zoom minimum (niveau 5) utilisé pour notre représentation de la France. À tous les niveaux, les tuiles sont identifiées par la structure hiérarchique suivante :

```
Z (zoom) / X (abscisse) / Y (ordonnee)
```

Ainsi, au niveau 5, les 4 tuiles adjacentes sont alors organisées comme suit :

```
5/15/10 - 5/16/10
   |          |
5/15/11 - 5/16/11
```

6.2.2 Augmentation du niveau de zoom

Pour réaliser la fonctionnalité de zoom il faut que lorsqu'on zoom sur une tuile, celle-ci augmente en niveau de détail. Sauf que toutes les tuiles sont en 256×256 *px* Donc ce qui est réalisé c'est que chaque tuile se divise en quatre nouvelles tuiles de 256×256, offrant une échelle plus grande et une précision accrue.

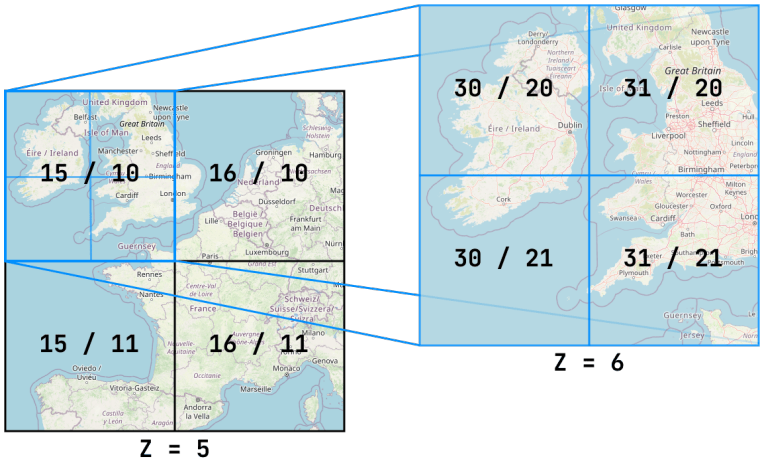

Figure 6.1 : *Hiérarchie des tuiles de carte*

Ainsi, une tuile du niveau Z se subdivise en quatre tuiles au niveau Z+1, selon les coordonnées suivantes (X et Y les coordonnées de la tuile au niveau Z) :

```
Z+1 / X*2 / Y*2
Z+1 / X*2 / Y*2 + 1
Z+1 / X*2 + 1 / Y*2
Z+1 / X*2 + 1 / Y*2 + 1
```

Avec la logique vue précédemment, on peut rapidement se rendre compte que toutes les coordonnées des tuiles se trouveront, pour chaque niveau de zoom, entre les valeurs (X_{min}, X_{max}) et (Y_{min}, Y_{max}) selon les formules suivantes :

$$(X_{min}, X_{max}) = \left(X_{init} \cdot 2^{Z-Z_{init}}, (2 + X_{init}) \cdot 2^{Z-Z_{init}} - 1\right) \tag{6.1}$$

$$(Y_{min}, Y_{max}) = \left(Y_{init} \cdot 2^{Z-Z_{init}}, (2 + Y_{init}) \cdot 2^{Z-Z_{init}} - 1\right) \tag{6.2}$$

6.3 Découverte et Implémentation d'OSMCtrl

6.3.1 Contexte et Choix de la Solution

Après plusieurs recherches infructueuses, il m'avait été recommandé d'explorer une solution déjà envisagée par un ancien stagiaire. Bien que son stage n'ait pas abouti à l'intégration dans PrynVision®, il avait identifié un gestionnaire de carte appelé OSMCtrl. Développée par PJ Naughter pour les applications MFC (Microsoft Foundation Class), cette bibliothèque permet d'intégrer des cartes avec des tuiles OpenStreetMap dans des applications Windows.

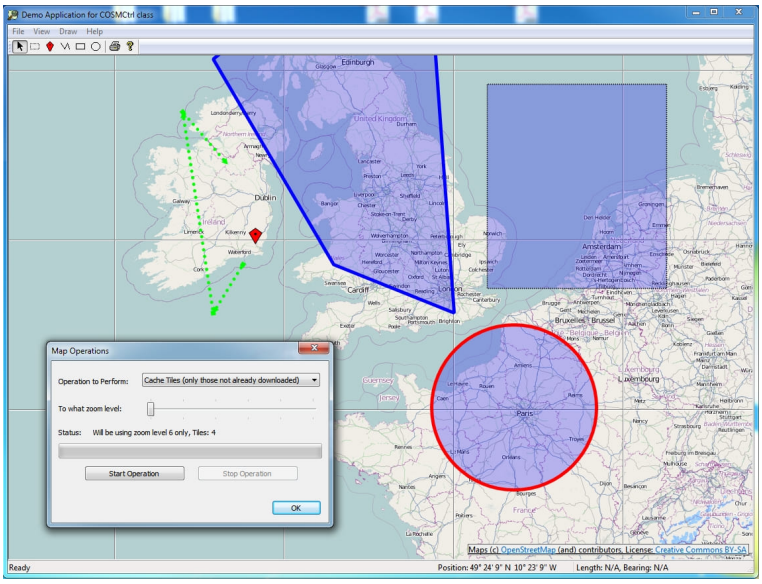

Figure 6.2 : *Programme d'exemple d'OSMCtrl*

6.3.2 Limitations Initiales

Cependant, OSMCtrl présente une limitation majeure : il ne permet pas de gérer les cartes localement. Lors de chaque utilisation, il envoie des requêtes au serveur OpenStreetMap pour récupérer les tuiles nécessaires, en spécifiant le niveau de zoom et les coordonnées X et Y de la tuile. Cela pose deux problèmes principaux :

1. L'utilisation régulière du serveur OpenStreetMap de cette manière n'est pas conforme aux politiques d'utilisation d'OpenStreetMap. Il est préférable de récupérer une seule fois les tuiles nécessaires.

2. Nous avons besoin d'une solution fonctionnant sans connexion internet, avec les cartes disponibles localement.

6.3.3 Solution Adoptée

Pour surmonter ces problèmes, j'ai utilisé un script Python pour extraire manuellement toutes les tuiles nécessaires. Le script envoyait les mêmes requêtes qu'OSMCtrl, mais uniquement pour la zone géographique de la France, et pour les niveaux de zoom de 5 à 11. Avec les coordonnées *min* et *max* déterminée dans la partie précédente, via les formules *voir* équation 6.1, équation 6.2 on peut alors déterminer l'algorithme *Python* permettant de télécharger la zone définie au niveau du zoom 5 sur tous les niveaux de zoom suivants.

Code *Python* qui permet d'extraire les tuiles sur les serveurs d'OpenStreetMap :

```
Zinit = 5
Zmax = 12
Xinit = 15
Yinit = 10

for z in range(Zinit, Zmax):
    for x in range(2**(z-Zinit)*Xinit, 2**(z-Zinit) (2 + Xinit)):
        for y in range(2**(z-Zinit)*Yinit, 2**(z-Zinit) (2 + Yinit)):
            download_tile(x, y, z, "Tiles_Carte_France") #Fonction qui
               permet d'enregistrer un fichier JPG dans les bons
               dossiers selon l'arborescence voulue
```

6.3.4 Calcul du nombre de tuiles et de l'espace disque nécessaire

En partant des 4 tuiles au niveau de zoom 5, on peut calculer le nombre de tuiles nécessaires pour chaque niveau de zoom supérieur et ainsi se rendre compte de l'espace qu'il occupe :

Niveau zoom	Nombre tuiles	Espace disque (Mo)
5	4	0.136
6	16	0.524
7	64	1.87
8	256	7.03
9	1 024	24.6
10	4 096	~ 80
11	16 384	248
12	65 536	~ 1 000

Tableau 6.1 : *Tableau de l'espace occupé par les tuiles à chaque niveau de zoom*

On remarque que l'ajout du niveau de zoom 12 serait très gourmand en espace disque (~1 *Go*), ce qui serait trop contraignant pour une application locale. De plus, le niveau 11 offrait suffisamment de détails pour notre utilisation. Pour un peu plus optimiser l'espace occupé par l'ensemble de la carte, j'ai décidé de supprimer le niveau 10, sachant qu'un saut de zoom (9 vers 11) lors de l'utilisation de l'application n'est pas trop dérangeant visuellement. Car OsmCtrl réalise un zoom numérique du niveau 9 (pour compenser le niveau 10 manquant) avant d'arriver au niveau 11, on n'a alors pas d'interruption même avec le niveau manquant.

6.4 Utilisation des tuiles locales

Après avoir extrait toutes les tuiles nécessaires pour notre carte locale, il était nécessaire d'ajuster le fonctionnement de l'OSMCtrl afin qu'il ne fasse plus de requêtes vers les serveurs d'OpenStreetMap.

Par chance, le code d'origine était conçu de manière à ce que les tuiles téléchargées soit automatiquement stockées dans un dossier de cache. Si le logiciel avait besoin de l'afficher à nouveau, il la chargeait directement depuis ce dossier.

La logique d'affichage d'une image brute étant déjà en place par le fait d'afficher les images du dossier de cache, j'ai simplement modifié le code pour supprimer tout ce qui concernait le

téléchargement des icônes. Au lieu de charger les tuiles depuis le cache, l'OSMCtrl les charge maintenant depuis un dossier spécifié. De plus, comme nous prévoyons d'implémenter plusieurs cartes et plans, j'ai ajouté la possibilité de changer dynamiquement le chemin du dossier où l'OSMCtrl va chercher ses icônes, permettant ainsi l'affichage de diverses cartes choisies dans l'interface par l'utilisateur.

Par la suite, j'ai également intégré la prise en charge des tuiles JPG, car par défaut, l'OSMCtrl ne supportait que les fichiers PNG. Cette amélioration a rendu l'utilisation du plan plus flexible et a facilité l'ajout de nouvelles cartes.

6.5 Implémentation des marqueurs

L'interêt principal de cette carte, une fois intégrée dans le produit final, sera d'afficher des informations en rapport avec des produits d'une société qui seraient répartis un peu partout sur le territoire français. Pour identifier ces produits, il est important qu'au premier coup d'oeil, l'opérateur les reconnaisse. Il faut alors pouvoir placer des marqueurs sur la carte, leur affecter une certaine configuration et les sauvegarder. L'OSMCtrl possède déjà une liste d'icônes compilées, mais qui ne correspond pas du tout à ce dont nous avons besoin et manque de fonctionnalités, comme la rotation d'une icône ou la sauvegarde par exemple. De plus, comme les ressources étaient compilées il fallait changer le comportement de l'OSMCtrl pour pouvoir charger des images depuis le disque et donc prendre en compte des images utilisateur.

J'ai donc créé une configuration externe à l'OSMCtrl nommée PrynMarker dans laquelle j'ai pus stocker les données dont j'avais besoin et ai pu gérer la sauvegarde. Cette configuration est alors chargée dans l'OSMCtrl en réutilise ses classes d'Icônes et de Marqueur.

6.6 Ajout des plans utilisateur

Dans la version actuellement distribuée du Prynvision, il est possible d'afficher des plans simples, comme une image statique sur laquelle on peut apposer des marqueurs. En revanche, la gestion de cartes complètes, comme une carte de la France, n'est pas supportée. Avec l'intégration de l'OSMCtrl, nous perdons cette fonctionnalité de base qui permet l'affichage de plans simples.

Utiliser la méthode actuelle pour afficher des plans en 3D tout en intégrant la carte de la France avec un autre outil serait dommage. En effet, on parvient déjà à manipuler une carte complexe avec l'OSMCtrl. De plus, cela reviendrait à implémenter deux technologies différentes pour réaliser des tâches similaires.

C'est pourquoi j'ai fait en sorte de pouvoir convertir une image JPG en un plan pour l'OSMCtrl et qu'il soit ainsi manipulable de la même façon que la carte de la France.

Pour ce faire, je récupère une image JPG, donnée par l'utilisateur, puis j'utilise des fonctions interne à Pryntec® pour l'extraire dans un buffer RGB ($24bits$). Ainsi à partir de ce buffer, je peux recréer des images aux dimensions voulues. C'est-à-dire que je prends le buffer RGB et le décompose en un set de tuiles contôlables comme la carte de la France. Cette opération peut être complexe à réaliser car souvent, les dimensions de l'image ne sont pas multiples de 256×256 (dimensions d'une tuile) et alors l'image ne se retrouve pas centrée sur le plan afficher. Pour

obtenir quelque chose de plus esthétique j'ai calculé les positions pour obtenir une image centrée peu importe le nombre de tuiles qui la découpent et affiché des pixels blancs sur les bords du plans non remplis par des pixels de l'image.

Le pseudo-code suivant démontre comment est réalisé le découpage :

Algorithme 1 : Conversion d'une image JPG en plan pour OSMCtrl

```
Input : cheminJPG, OSMCtrl

image ← decodeImageJPEG(cheminJPG);

largeur, hauteur ← obtenirDimensions(image);
(nbTuilesL, nbTuilesH) ← ajusterNbTuiles(largeur, hauteur, OSMCtrl);
offsets ← calculerOffsets(largeur, hauteur, nbTuilesL, nbTuilesH, OSMCtrl);

bufferImage ← obtenirBuffer(image);
encodeur ← nouveauEncodeurJPEG();
bufferTuile ← allouerMemoire(tailleTuile);

for x ← 0 to nbTuilesL do
    for y ← 0 to nbTuilesH do
        initialiserBufferTuile(bufferTuile);
        copierImageDansTuile(bufferImage, bufferTuile, largeur, hauteur, offsets);
        encodeur.Load(nouvelleImage(bufferTuile, 256, 256));
        bufferJPEG ← encodeur.Encode();
        sauvegarderTuile(bufferJPEG, creerCheminTuile(LabelCarte, x, y));

chemin ← creerCheminZoom(LabelCarte);
niveauZoom ← 12;

while creerNiveauZoomSuivant(chemin, niveauZoom, OSMCtrl) do
    niveauZoom = niveauZoom - 1;
```

On remarque une méthode dans le code précédent : $creerNiveauZoomSuivant()$. En fait pour les petites images le code fonctionnait très bien. Mais lorsqu'on avait une grande image, par exemple 10 000 × 10 000 *pixels*, au moment d'afficher le plan on se retrouvait alors avec un niveau de zoom très important, on était très proche de l'image. Car les tuiles de 256×256 *pixels* sont toujours affichées de cette taille. Donc pour les grandes images il était nécessaire de créer des niveaux de zoom permettant de s'éloigner du premier niveau. J'ai alors réalisé une fonction qui transforme, pour l'entièreté du niveau de zoom, quatre tuiles adjacentes en une seule de 256×256.

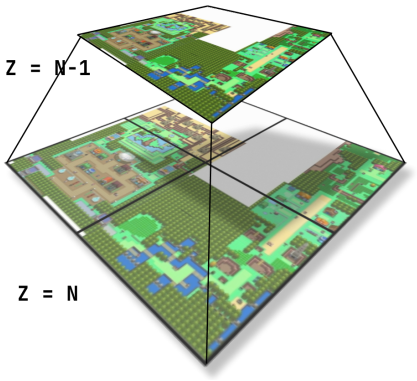

Figure 6.3 : *Transformation de 4 tuiles en 1*

Pour réaliser ceci, je récupère le buffer de chaque tuile et pour chacun, je prends 4 pixels adjacents, réalise la moyenne, pour n'en faire qu'un seul dans la nouvelle tuile. Ainsi en faisant ça de manière récursive jusqu'à atteindre un minimum de 2 tuiles de large ou de haut. On transforme un fichier JPG en mosaïque de tuiles.

Pour tester cette fonction j'ai décidé d'utiliser une carte d'un jeu très connu (Pokémon). L'image de base fait 14 879 × 9602 *px* donc nous permettra d'exécuter un cas complexe et d'identifier les limites de notre algorithme. Voici ce qu'on obtient après avoir transformé l'image en plan :

Figure 6.4 : *JPG de la carte de Pokémon*

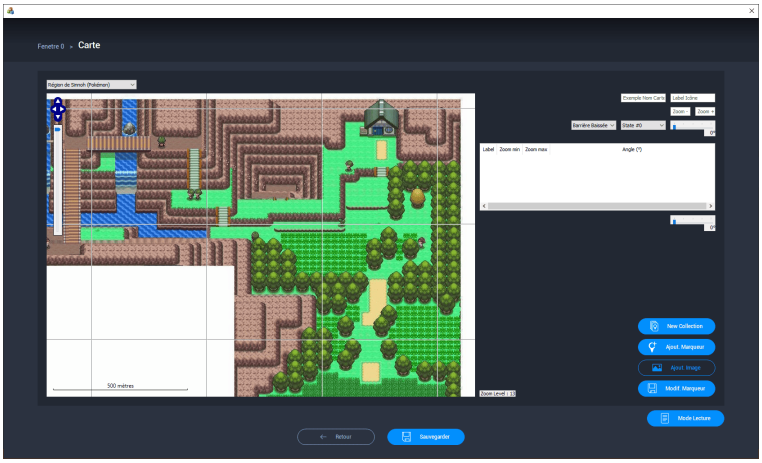

Figure 6.5 : *Tuiles du plan nouvellement créé*

On remarque alors qu'il est maintenant possible de créer un plan et l'utiliser dans notre application à partir de n'importe qu'elle image. En réalité, cela permettra surtout aux utilisateurs de situer leurs caméras sur un plan représentant leur bâtiment, magasin, entrepôt, etc. Donc peu de chances d'utiliser une image de cette taille mais cela nous permet de nous rendre compte que cela devient possible. En effet, à l'heure actuelle la solution dans le Prynvision® ne permettait pas de charger un plan de cette dimension.

6.7 Actions/évènements liés à la carte

Maintenant que nous avons développé l'utilisation de plans dans Prynvision®, nous pouvons utiliser la logique d'action/évènement vue en première partie afin d'apporter des modifications à la carte. Pour démontrer comment cela serait réalisé, je vais placer une icône sur l'ENSIM (*Ecole Nationale Superieure des Ingenieurs du Mans*) et réaliser un changement de couleur de celle-ci à une heure précise et pendant une durée déterminée.

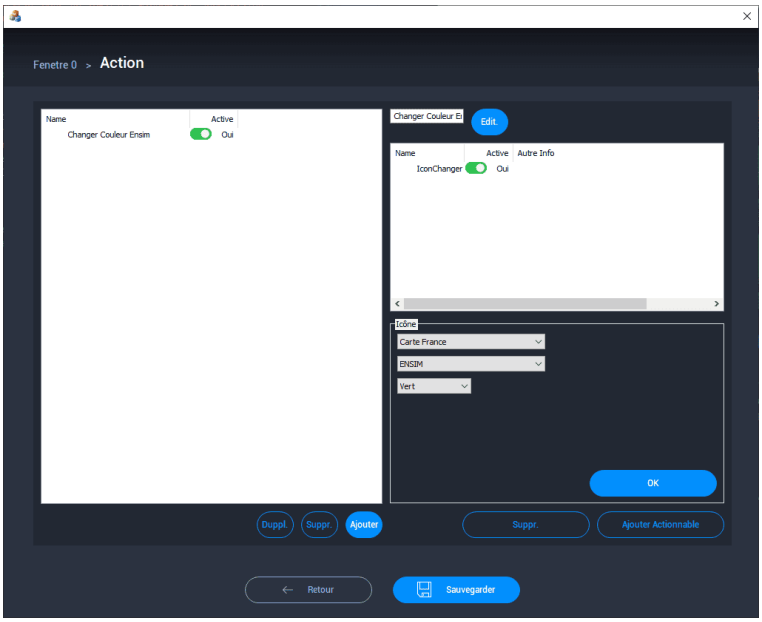

Figure 6.6 : *Création action changement icône*

J'ai alors créé dans l'interface des actions, une action appellée "Changer couleur ENSIM" prenant en actionnable un $IconChanger$ s'appliquant sur l'icône $ENSIM$ de la carte $CarteFrance$ et passant le l'icône du marqueur en vert. Ensuite, on crée l'évènement avec un déclencheur $Journalier$ pour qu'il réalise cette action à 15h pour une durée de 5 *sec*.

Ainsi on obtient le résultat suivant :

Figure 6.7 : *Changement de couleur du marqueur "ENSIM"*

L'utilisateur peut alors réaliser le même type d'évènement pour chaque icône du plan. Associer un plan à cette logique d'action/évènement est un gros avantage pour le Prynvision®. En effet, remonter des informations à l'aide d'alertes visuelles sur un plan permet à l'opérateur d'être beaucoup plus efficace. Cela lui permet de voir tout de suite où se situe le produit qui a un problème ou au moins qui possède un intérêt particulier nécessitant son action dans l'immédiat. Dans le plan actuellement utilisé sur Prynvision®, on peut se connecter à un produit en cliquant sur une icone du plan. Il n'est pas juste uniquement utilisé pour l'affichage, mais permet d'accéder rapidement aux sites qui remotnent des alertes.

L'utilisateur peut alors appliquer la même logique d'événement à chaque icône du plan. Associer un plan à cette logique d'action/événement représente un avantage majeur pour Prynvision®. En effet, le fait de pouvoir recevoir des alertes visuelles directement sur un plan permet à l'opérateur d'agir avec une plus grande efficacité. Il est ainsi capable de localiser immédiatement

le produit qui présente un problème, ou du moins celui qui nécessite une intervention immédiate en raison d'un intérêt particulier.

Dans le plan actuellement utilisé par Prynvision®, il est possible de se connecter à un produit en cliquant simplement sur une icône. Le plan n'est donc pas seulement un outil d'affichage, il permet également un accès rapide aux sites qui émettent des alertes. Cette fonction n'est pas encore implémentée dans ma version du plan mais est prévue pour la suite du développement

6.8 Configuration marqueurs utilisateur

Le Prynvision® possède à l'heure actuelle une large gamme d'icônes permettant d'identifier visuellement beaucoup de produits ou du moins une gamme de produits. Mais le prynvision est compatible avec be ucoup de matériels différents. Donc si l'utilisateur veut des icônes permettant d'identifier son matériel spécifique, il faut lui laisser la possibilité d'en rajouter. Par exemple, le client pour lequel était développé ce nouveau plan, voulait pouvoir l'utiliser pour remonter des informations en rapport avec des barrières. L'icône barrière n'est pas présente à l'heure actuelle dans le logiciel donc il serait satisfait de pouvoir l'ajouter au logiciel afin d'afficher des alertes visuelles plus pertinentes. J'ai alors réalisé une page qui permet à l'utilisateur de configurer ses icônes. Cette configuration est en fait une liste de Collections d'icônes. Ces icônes étant elles-même composées de listes d'états.

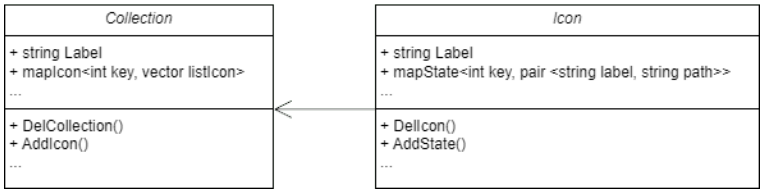

Figure 6.8 : *Diagramme classe collection d'icônes*

Maintenant que nous avons visualisé les classes métier, il faut réaliser la page qui permettra à l'utilisateur de voir les configurations d'icônes déjà présentes, mais aussi d'ajouter les composants lui permettant d'ajouter facilement ses icônes. Ainsi sur la page on retrouve deux listes principales. L'une permettant de manipuler les différentes collections. L'autre premettant de manipuler les icônes ainsi que leurs différents états. Lorsqu'on veut rajouter une icône, un bouton nous permet d'ouvrir l'explorateur de fichier et de séléctionner les PNG correspondant à nos différents états. Une fois séléctionnés, on est invité à séléctionner les coordonnées du point d'ancrage. Ce point permet de définir les coordonnées de l'image que le marqueur pointe. Plus précisément, cela permet de définir l'endroit de l'icône qui correspond aux coordonnées géographiques pointées par le marqueur.

Maintenant que nous avons visualisé les classes métier, il est temps de réaliser la page permettant à l'utilisateur de gérer les configurations d'icônes existantes, ainsi que d'ajouter de nouvelles icônes de manière efficace. Cette page est composée de deux listes principales. La première liste permet de manipuler les différentes collections d'icônes, tandis que la seconde se concentre sur la gestion des icônes liées aux premières, ainsi que de leurs différents états.

Pour ajouter une nouvelle icône, l'utilisateur peut cliquer sur un bouton qui ouvre l'explorateur de fichiers, permettant ainsi de sélectionner les fichiers PNG correspondant aux différents états de l'icône qu'il veut ajouter. Une fois les fichiers sélectionnés, l'utilisateur est invité à ajouter

les coordonnées du point d'ancrage. Ce point d'ancrage est crucial, car il détermine la position exacte de l'image sur la carte. Il spécifie quelle partie de l'icône correspond aux coordonnées géographiques pointées par le marqueur.

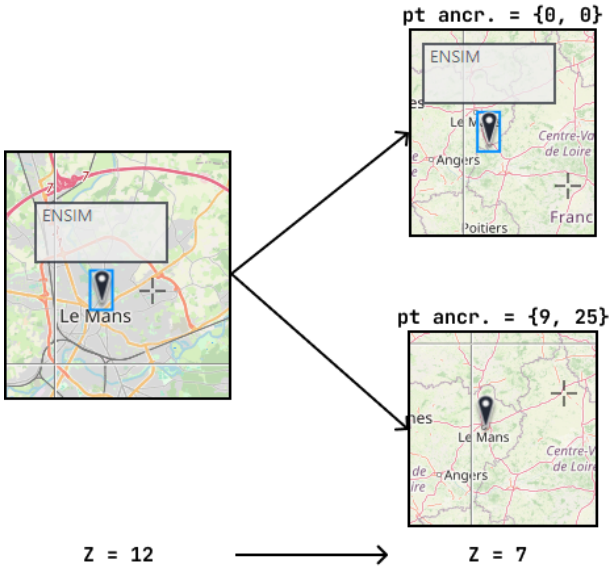

Figure 6.9 : *Différence utilisation point d'ancrage ou non*

On remarque avec l'exemple précédent l'utilité du point d'ancrage. Lorsqu'on place une icône, il faut lui associer un point d'ancrage. Par défaut on pourrait utiliser un coin de l'icône ou même le centre, mais on observerai toujours un comportement anormal lorsqu'on diminue le zoom. L'icône ne pointerait plus les bonnes coordonnées suivant le niveau de zoom utilisé. Ce point d'ancrage sert aussi en cas de rotation de l'icône, permettant de pointer toujours le même point malgrés celle-ci. Et comme chaque icône possède un design et des dimensions propres, elles possèdent toutes un point d'ancrage différent. Il faut donc laisser la possibilité à l'utilisateur de renseigner le point d'ancrage de son icône. Dans un premier temps, cela sera seulement 2 champs de texte permettant de remplir ces deux coordonnées. Mais plus tard, l'icône sera dessinée au moment de l'ajout, et des sliders permettront d'ajuster le point d'ancrage par rapport au dessin de l'icône.

6.9 Résultats du système de cartographie

L'implémentation d'un système de manipulation de carte est au final très intéressante car il y a beaucoup d'aspects mathématiques à prendre en compte lors de la création et la manipulation des tuiles. De plus, intégrer un projet existant dans son projet était une chose que je n'avais encore jamais faite. Il faut alors s'adapter à un code existant, se l'approprier pour pouvoir le modifier et en faire ce dont on a réellement besoin. Grâce à l'implémentation d'OSMCtrl, l'utilisateur est maintenant capable de repérer sur une carte possédant différents niveaux de zoom, l'ensemble de ses produits tout en retrouvant le comportement des plans existant déjà dans le Prynvision®. Cela manque encore de paufinage, pour éviter les quelques bugs ainsi que retrouver l'ensemble des fonctionnalités du plan actuel mais c'est une base solide pour la suite de ce projet.

Chapitre 7

Conclusions

Ce stage a été une expérience enrichissante, me permettant d'acquérir une compréhension approfondie des défis liés à la modernisation d'un logiciel complexe tel que Prynvision®. L'introduction de la logique d'action/évènement a non seulement amélioré la réactivité et l'efficacité du logiciel, mais elle a aussi permis une meilleure centralisation et gestion des fonctionnalités, rendant l'outil plus intuitif pour les utilisateurs finaux.

L'intégration réussie de cette nouvelle logique dans l'interface utilisateur, ainsi que l'amélioration des fonctionnalités de cartographie, constituent des avancées significatives pour le Prynvision®. Ces améliorations garantissent une plus grande cohérence avec les autres logiciels de Prynel, tout en offrant aux utilisateurs une interface plus moderne et fonctionnelle. Ce projet a donc permis de renforcer l'attrait de Prynvision® tout en posant les bases pour de futures évolutions.