

HAUTE ÉCOLE D'INGÉNIERIE ET DE GESTION
DU CANTON DE VAUD

PROJET DE GROUPE (PDG)

RAPPORT DE PROJET

LibreDraw

Auteurs :

Sacha BRON
Yassin KAMMOUN
Paul NTAWURUHUNGA
Marc PELLET
David VILLA

Superviseur :

Pr René RENTSCH

4 janvier 2016



HAUTE ÉCOLE
D'INGÉNIERIE ET DE GESTION
DU CANTON DE VAUD
www.heig-vd.ch

Hes-SO
Haute Ecole Spécialisée
de Suisse occidentale

Table des matières

1	Introduction	11
1.1	Description du projet	11
1.2	Objectif du projet	13
1.3	Technologies utilisées	13
1.4	Équipe de projet	13
1.5	Cadre de réalisation	13
1.6	Choix du sujet	14
2	Conception	15
2.1	Système global	15
2.1.1	Architecture	15
2.1.2	Diagramme d'activités	19
2.2	LibreDraw System	20
2.2.1	Architecture	21
2.2.2	Drawing Subsystem	23
2.2.3	Tracking Subsystem	32
2.2.4	Camera Subsystem	35
2.3	Mouse Simulator System	36
2.3.1	Architecture	36
2.3.2	Network I/O Listener	36
2.3.3	Mouse Controller	36
2.4	Beamer	37
2.5	Camera	37
2.6	Stylus	37
2.7	Drawing Area	37
2.8	Stylet	37
2.8.1	Modèle à une LED	37
2.8.2	Modèle à deux LEDs	40
2.8.3	Pointeur lumineux	44
2.9	Environnement de dessin	46
2.9.1	Matériel requis	46

2.9.2	Mise en place du matériel	46
2.9.3	Contraintes	46
2.9.4	Exemple d'environnement	46
3	Description technique	49
3.1	Structure du projet	50
3.2	Patrons de conception	51
3.3	Librairies	51
3.3.1	Qt	51
3.3.2	OpenCV	51
3.4	Interface graphique utilisateur	52
3.4.1	Structure de l'interface utilisateur	52
3.4.2	Zone de dessin	54
3.4.3	Barre d'outils	55
3.4.4	Menu	56
3.4.5	Ressources	58
3.5	Tracking	60
3.5.1	Introduction à la vision par ordinateur	60
3.5.2	Système de détection	62
3.6	Dessin	64
3.6.1	Transition entre outils	65
3.6.2	Outils de dessin	66
3.6.3	Outils de formes	69
3.6.4	Outil de sélection d'épaisseur de trait	74
3.6.5	Outil de gestion de pile d'actions	75
3.7	Interfaçage tracking-dessin	79
3.7.1	Simulation et contrôle de la souris	79
3.8	Clavier virtuel	80
3.8.1	Ouverture de fichier	82
3.8.2	Sauvegarde de fichier	83
3.8.3	Interface d'utilisation	84
3.9	Impression	85
4	Tests & Validation	87
4.1	Stratégie de tests	87
4.1.1	Tracking Subsystem	87
4.1.2	Drawing Subsystem	88
4.2	Matériel	89
4.3	Procédures de tests	90
4.4	Résultats	90
5	Problèmes connus	91
6	Conclusion	93

6.1	Solution proposée	93
6.1.1	Fonctionnalités implémentées	93
6.1.2	Fonctionnalités manquantes	93
6.1.3	Propositions d'amélioration	94
6.2	Problèmes rencontrés	94
6.2.1	Problèmes techniques	94
6.2.2	Problèmes organisationnels	94
6.2.3	Problèmes de planification	95
6.3	Respect du planning	95
6.3.1	Planification initiale	96
6.3.2	Évolution	97
6.4	Déroulement du projet	98
6.4.1	Points positifs	98
6.4.2	Points négatifs	98
6.5	Synthèse	98
A	Cahier des charges	99
B	Journal de travail	111
C	Planification	114

Table des figures

1.1	Exemple d'environnement de dessin	12
2.1	Vue d'ensemble de l'architecture du système global	16
2.2	Vue détaillée de l'architecture du système global	18
2.3	Activité globale d'une esquisse utilisateur	19
2.4	Architecture du système LibreDraw System	21
2.5	Architecture du sous-système Drawing Subsystem	24
2.6	Architecture du composant Core	25
2.7	Structure de l'élément Window	26
2.8	Structure de l'élément Menu	26
2.9	Structure de l'élément Toolbar	27
2.10	Architecture du composant Virtual Keyboard	27
2.11	Agencement des touches du clavier virtuel selon le mode alphabétique	28
2.12	Agencement des touches du clavier virtuel selon le mode symbole	28
2.13	Elément Key mono-evalué	29
2.14	Elément Key multi-evalué	29
2.15	Architecture du composant Drawing	30
2.16	Structure logique d'un dessin	30
2.17	Architecture du sous-système Tracking Subsystem	32
2.18	Architecture du sous-système Tracking Subsystem	35
2.19	Architecture du système Mouse Simulator System	36
2.20	Prototype initial du stylet	38
2.21	Seconde version du prototype du stylet	41
2.22	Utilisation de notre prototype de stylet à deux LEDs	42
2.23	Exemple de problème d'alignement des LEDs avec la pointe	43
2.24	Exemple de pointeur lumineux à LED	44
2.25	Exemple d'environnement de dessin - vue en perspective	47
2.26	Exemple d'environnement de dessin - vue de gauche	48
3.1	Interface graphique utilisateur finale	52
3.2	Structure de l'interface graphique utilisateur finale	53
3.3	Zone de dessin	54

3.4	Barre d'outils	55
3.5	Menu	56
3.6	Interface - Demande de sauvegarde	56
3.7	Interface - Fenêtre de sauvegarde	57
3.8	Image source de la caméra	63
3.9	Filtre HSV	64
3.10	Rectangle approximés de l'écran	64
3.11	Sélection d'un outil de la barre d'outils	65
3.12	Clavier virtuel selon le mode alphabétique	80
3.13	Clavier virtuel selon le mode symbole	80
4.1	Test grandeur nature de l'application lors de son développement	88
4.2	Lancement d'une calibration en utilisant une tablette comme écran vert	89
4.3	Dessin en utilisant la tablette comme surface	89
6.1	Planification initiale des tâches	96
6.2	Planification initiale des tâches	97
A.1	Mockup de l'interface graphique utilisateur	100
A.2	Prototype initial du stylet	102
A.3	Vue en perspective d'un environnement idéal	105
A.4	Vue de côté d'un environnement idéal	106
A.5	Planification initiale des tâches	107
C.1	Planification initiale du projet	115

Liste des tableaux

1.1	Équipe de projet	13
3.1	Outils de la barre d'outils de l'interface graphique utilisateur	55

Chapitre I

Introduction

Ce chapitre est une introduction de ce rapport de travail. Il s'agit dans un premier temps de rappeler le sujet du projet par une description complète de celui-ci. Viennent ensuite l'énumération, le commentaire et l'explication des objectifs que cherche à atteindre ce projet. Les technologies utilisées pour la réalisation du système sont introduites par la suite. Les différents membres constituant l'équipe de projet sont présentés. Cette introduction décrit précisément les rôles de chacun des protagonistes. Finalement, un commentaire quant à la décision de choisir un tel sujet de projet est exposé. Cela consiste à partager les motivations et les raisons qui ont poussé le groupe à partir sur un tel projet.

1.1 Description du projet

Le projet consiste en un outil de dessin tout à fait standard. Celui-ci permet entre autres de dessiner sur un espace de travail. Pour ce faire, l'utilisateur dispose d'un éventail d'outils :

- Outils de dessin : les outils de dessin mettent à disposition un crayon et une gomme permettant de dessiner sans restriction n'importe quelle forme géométrique.
- Outils de formes : les outils de formes mettent à disposition un ensemble de formes géométriques prédéfinies pouvant être dessinées au sein d'un dessin et redimensionnées à la guise de l'utilisateur.
- Outils de couleurs : les outils de couleurs mettent à disposition une palette de couleurs laquelle permet de définir la couleur du trait aussi bien pour les outils de dessin que pour les outils de formes.
- Outils d'épaisseur : les outils d'épaisseur permettent de définir selon une liste prédefinie l'épaisseur du trait aussi bien pour les outils de dessin que pour les outils de formes.

Malgré ce côté simpliste du système, celui-ci se démarque des outils de dessin traditionnels par le fait que l'espace de travail du dessinateur est non pas l'écran de l'utilisateur mais un support physique tel un mur, un sol, une table ou n'importe quelle autre surface susceptible de jouer le rôle de support de dessin. L'idéal est bien évidemment une surface plane. Toutefois, le système n'est pas restreint par une telle propriété. Celle-ci pourrait tout aussi bien être théoriquement abrupte, instable et bicornue. L'utilisateur définit lui-même ce qu'il juge être un support de dessin propice pour travailler.

Le fait de dessiner sur un support physique plutôt que virtuel nécessite une substitution de la souris de l'ordinateur à un outil plus adéquat pour dessiner. Ceci est rendu possible par la mise à disposition d'un stylet (aussi appelé pointeur) au dessinateur. Ce stylet est conçu sur mesure par l'équipe de projet pour les besoins du système. Il reste toutefois un outil expérimental avec une casquette de prototype. En effet, celui-ci n'a pour but que de valider la conception et l'implémentation du système. Il n'en demeure pas moins que ce stylet reste relativement complexe pour accomplir une telle tâche.

Le stylet se présente sous la forme d'un stylo tout à fait usuel. Toutefois, de par l'objectif de son utilisation, il est caractérisé par les composants suivants :

- Une LED rouge.
- Une LED verte.
- Un interrupteur.
- Une résistance.
- Une batterie.
- Un boîtier.

Bien que l'action de dessiner soit réalisée sur un support physique, le dessin à proprement parlé n'est pas gravé sur ce support. Les faits et gestes du dessinateur avec le stylet sont capturés par une caméra. Un programme informatique reçoit en permanence en provenance de cette caméra des informations liées aux faits et gestes du stylet manipulé par le dessinateur. C'est là qu'intervient le mécanisme du tracking, c'est-à-dire la détection et le suivi du stylet. Toutes ces informations récupérées sont communiquées à un autre programme informatique. Ce dernier stocke, analyse, traite et reproduit ces mêmes faits et gestes de manière à reconstituer logiquement et graphiquement le dessin correspondant.

Afin que le dessinateur puisse disposer d'un retour instantané de son dessin, un projecteur projette la reproduction fidèle du dessin réalisée au sein du second programme informatique. Ainsi, le dessinateur a l'illusion de dessiner directement sur son support physique. Il dispose en plus de cela des fonctionnalités usuelles de sérialisation de dessin. Il peut en effet enregistrer son travail et le reprendre ultérieurement.

La figure suivante illustre un exemple d'environnement de dessin :

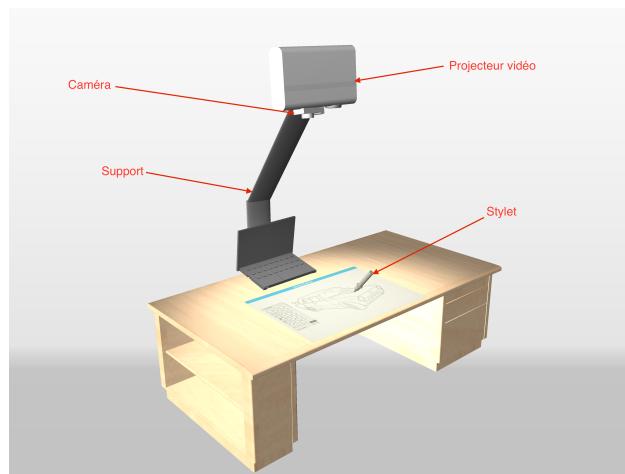


FIGURE 1.1 – Exemple d'environnement de dessin

1.2 Objectif du projet

L'objectif de ce projet est de concevoir un outil de dessin assisté par ordinateur permettant à l'utilisateur de réaliser ses dessins de la manière la plus naturelle possible. À terme, l'utilisateur dessinera directement sur sa table ou n'importe quelle autre surface plane à l'aide d'un stylet et son dessin sera projeté sur son plan de travail, donnant ainsi à l'utilisateur l'impression de dessiner avec un crayon et une feuille.

1.3 Technologies utilisées

Les technologies utilisées pour le développement de l'application et le suivi du stylet sont les suivantes :

- Qt 5.5 : Le framework Qt est une API orientée objet offrant des composants d'interface graphique, d'accès aux données, de connexions réseaux, de gestion des fils d'exécution, Dans le cadre du projet, elle est utilisée pour construire l'interface graphique utilisateur de l'application. Pour de plus amples informations, veuillez-vous référer au site officiel <http://www.Qt.io>.
- OpenCV 3.1 : Open Computer Vision (OpenCV) est une bibliothèque graphique libre spécialisée dans le traitement d'images en temps réel. La bibliothèque OpenCV met à disposition de nombreuses fonctionnalités pour le traitement d'images, le traitement vidéos, les calculs matriciels, Dans le cadre du projet, elle est notamment utilisée pour le tracking du stylet. Pour de plus amples informations, veuillez-vous référer au site officiel <http://OpenCV.org>.
- C++11 : Le C++ est utilisé en guise de langage de programmation de base permettant de manipuler les différentes librairies utilisées dans le cadre du développement du projet. La librairie OpenCV et le framework Qt utilisant par défaut le langage C++, celui-ci constitue donc un choix idéal pour le développement du système.

1.4 Équipe de projet

Le tableau suivant présente l'équipe de projet, la hiérarchie au sein du groupe ainsi que les rôles joués par les différents membres du groupe :

Nom, prénom	Hiérarchie	Rôles
Bron, Sacha	Chef de projet	Prototypes, gestion caméras, expérience utilisateur
Villa, David	Chef suppléant	Interface graphique, outils de dessin
Kammoun, Yassin	-	Interface graphique, documentation
Ntawuruhunga, Paul	-	Tracking, calibrage, tests
Pellet, Marc	-	Interface graphique, outils de dessin

TABLE 1.1 – Équipe de projet

1.5 Cadre de réalisation

Ce projet s'inscrit dans le cadre du cours de Projet de Groupe (PDG) au sein de la Haute Ecole d'ingénierie et de gestion du canton de Vaud (HEIG-VD) sis à Yverdon-les-Bains. Selon le plan d'études de l'école, il est dispensé aux étudiants IL du département des Technologies de l'Information et de la Communication (TIC) pour le compte de leur troisième année de formation Bachelor. Le but de ce cours

est d'effectuer un projet en passant par toutes les étapes de développement. Cela inclut le choix d'un sujet, la définition d'un cahier des charges, une phase de recherche et d'analyse suivie du développement de l'application et d'une phase de tests et de validation. En dernière instance, un rapport sur le déroulement du travail et une présentation du projet sont requis dans le but d'évaluer le travail effectué.

1.6 Choix du sujet

Le choix d'un tel sujet se justifie par le fait que ce projet exige un important travail de recherche, de découverte et d'apprentissage de nouvelles technologies comme cela peut notamment être le cas pour la librairie OpenCV. Bien évidemment, ce genre de projet présente des risques compte tenu du fait que la technologie n'est connue de personne, qu'elle peut être difficile à appréhender, à mettre en oeuvre et à maîtriser, que la faisabilité du projet n'est pas facilement définissable et que le temps d'apprentissage est difficilement estimable. Toutefois, toutes ces problématiques ne rendent le projet que plus motivant, attrayant et intéressant.

Ce projet s'inscrivant dans un cadre académique, il s'agit donc d'une opportunité idéale d'acquérir de nouvelles connaissances. Le sujet en lui-même est des plus intéressants. Il se démarque clairement de la monotonie des applications développées dans un tel contexte. Des projets similaires ont certainement déjà été réalisés que ce soit dans un cadre professionnel que dans un cadre académique mais à bien moindre mesure ce qui laisse énormément de place pour la créativité et l'innovation. Par ailleurs, la complexité d'un tel projet ne rend que plus grand le mérite une fois le travail terminé avec un système tout à fait fonctionnel.

D'un point de vue organisationnel, ce projet présente la particularité d'être subdivisé en deux sous-projets étant donné que deux programmes sont développés : l'outil de dessin et le système de tracking du stylet. Un troisième sous-projet pourrait encore ressortir de ces deux derniers puisque la conception et la fabrication du stylet est un travail à part entière. Ainsi, un défi clairement établi de ce projet est l'intégration des différentes composantes pour finalement former un tout.

En dernière instance, ce sujet de projet a été choisi en premier lieu par son originalité puisque il s'avère fort différent des travaux réalisés dans le cadre de la formation Bachelor. Par ailleurs, le fait qu'il présente une difficulté certaine et un risque potentiel d'échec rend le projet d'autant plus motivant à réaliser. Enfin, l'étude d'une nouvelle technologie telle que la librairie OpenCV jusque-là parfaitement inconnue à l'équipe est attrayante puisqu'elle permet à chaque membre du groupe d'enrichir son bagage technique.

Chapitre 2

Conception

Ce chapitre se veut être une description complète aussi bien de la conception du système que de la conception du stylet. Il débute donc par la présentation complète du système. Cette présentation introduit dans un premier temps une vision globale du système. Cette étape préliminaire est suivie par une description détaillée et progressive de chaque composant du système. Il s'agit concrètement de présenter en fond et en large l'architecture de l'ensemble du système, de chaque sous-système et de leurs composants respectifs. Des diagrammes de classes et des diagrammes d'activités accompagnent cette description du système. Les différents modèles de stylet pour leur part sont introduits en premier lieu par une description de leur principe d'utilisation. Une présentation des différents composants matériels qui les constitue est ensuite réalisée. Un exposé de leur prototype est effectué au moyen de schémas. Les étapes de fabrication de ces modèles sont énumérées et commentées en long et large par la suite. Enfin, le fonctionnement de chacun des modèles est introduit. En dernière instance, l'environnement de dessin idéal est schématisé à l'aide de modélisations accompagnées de descriptions.

2.1 Système global

Cette section présente l'ensemble du système d'un point de vue global. Le but est d'être aussi abstrait que possible de manière à ce que l'étude du système se fasse de manière progressive. Cette présentation débute par l'introduction de l'architecture du système.

Le système global correspond en la réunion de l'infrastructure matérielle et des composants logiciels compte tenu de la nature du projet. Cela se justifie notamment par le fait que des composants matériels interagissent avec des composants logiciels et inversement.

2.1.1 Architecture

L'architecture est présentée en deux variantes : une vue d'ensemble et une vue détaillée. Cette dernière n'est là qu'à titre indicatif de façon à ce que le lecteur dispose de la version complète de l'architecture ; elle ne fait pas l'objet d'une description, seule la version allégée le fait.

2.1.1.1 Vue d'ensemble

La figure suivante propose une vue d'ensemble de l'architecture de du système dans sa globalité :

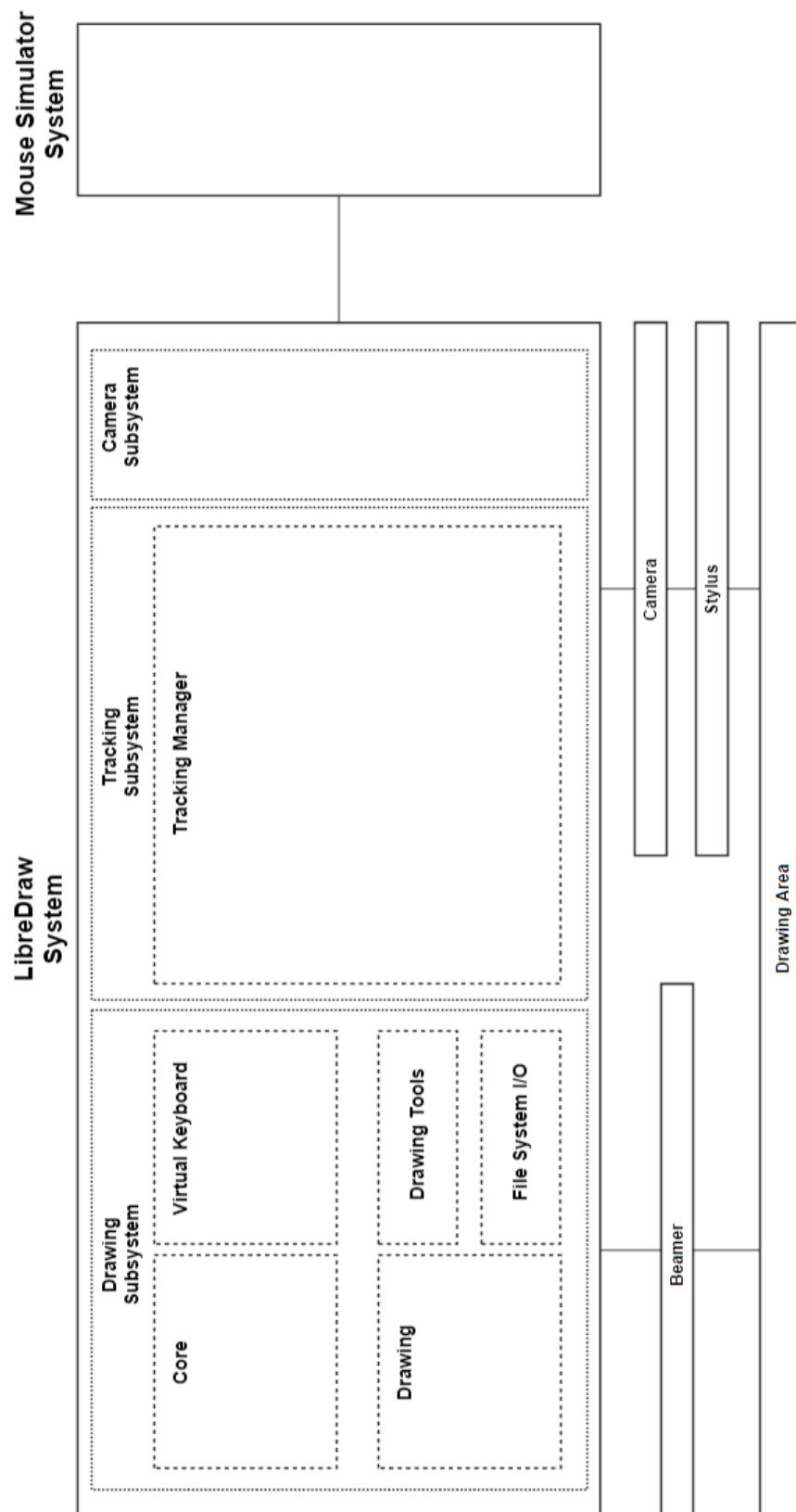


FIGURE 2.1 – Vue d'ensemble de l'architecture du système global

Le système dans sa globalité est en fait constitué de deux systèmes à part entière qui sont respectivement **LibreDraw System** et **Mouse Simulator System**. Par ailleurs, le système global est caractérisé par des éléments externes d'ordre matériel interagissant avec les deux systèmes mentionnés plus tôt ; il s'agit concrètement des éléments *Beamer*, *Camera*, *Stylus* et *Drawing Area*.

Le système **LibreDraw System** correspond à l'application de dessin à proprement parlé laquelle gère aussi bien la partie outil de dessin que la partie tracking du stylet. Ces tâches sont gérées respectivement par les sous-systèmes **Drawing Subsystem** et **Tracking Subsystem**. Par ailleurs, ce même système est caractérisé par un sous-système supplémentaire à savoir **Camera Subsystem**. Celui-ci a pour objectif de gérer l'initialisation et la mise à jour de la communication avec un dispositif d'enregistrement vidéo, une caméra donc, connecté au système hôte. Autrement dit, il s'agit de l'élément matériel *Camera*.

Le sous-système **Drawing Subsystem** contient entre autres le *Core* de l'application du point de vue expérience utilisateur, c'est-à-dire son interface graphique. Par ailleurs, compte tenu du fait que l'interaction de ce même utilisateur avec l'application s'effectue par le biais d'un stylet, un clavier virtuel est mis à disposition par le constituant *Virtual Keyboard* pour toute saisie textuelle d'information. L'esquisse du dessin de l'utilisateur avec le stylet est reproduite par le constituant *Drawing*. Cette reproduction est appliquée par des outils de dessin regroupés dans le constituant *Drawing Tools*. La sérialisation d'un dessin, sa sauvegarde et son importation au sein de l'application, est rendue possible par la mise à disposition de fonctionnalités fournies par le constituant *File System I/O*.

Le sous-système **Tracking Subsystem** est caractérisé par un constituant unique ; il s'agit du *Tracking Manager*, le gestionnaire du suivi du stylet dit autrement. En plus de gérer le stylet, ce constituant assure l'initialisation et la configuration de l'ensemble de l'infrastructure du système : écran, caméra et stylet.

Le système **Mouse Simulator System** est un composant à part entière. Celui-ci permet comme son nom l'indique de simuler une souris, c'est-à-dire de reproduire les faits et gestes de l'utilisateur lors de la manipulation du stylet comme s'il utilisait une souris en lieu et place de ce même stylet.

Le système dans son ensemble fonctionne globalement de la manière suivante :

1. Le lancement de l'application débute par l'initialisation et la configuration de l'infrastructure système. Ceci est réalisé par le constituant *Tracking Manager* du sous-système **Tracking Subsystem**.
2. Une fois la phase de configuration et d'initialisation terminée, les éléments matériels *Camera* et *Stylus* sont opérationnels et exploitables.
3. L'interaction de l'utilisateur par le stylet avec l'application est capturée par l'élément *Caméra* qui communique l'information au constituant *Tracking Subsystem*.
4. L'analyse des faits et gestes de l'utilisateur par le constituant *Tracking Manager* est communiquée au système **Mouse Simulator System**. Celui-ci simule l'action utilisateur et l'applique sur le sous-système **Drawing Subsystem** du système **LibreDraw**.
5. Le constituant *Drawing* du sous-système **Drawing Subsystem** veille à reproduire l'esquisse de l'utilisateur. Un élément adéquat du constituant *Drawing Tools* est utilisé pour ce faire.
6. L'élément matériel *Beamer* veillera à projeter la reproduction du dessin de l'utilisateur réalisé par l'élément *Stylus* sur le support de dessin physique représenté par l'élément *Drawing Area*.

2.1.1.2 Vue détaillée

La figure suivante propose une vue détaillée de l'architecture de du système dans sa globalité :

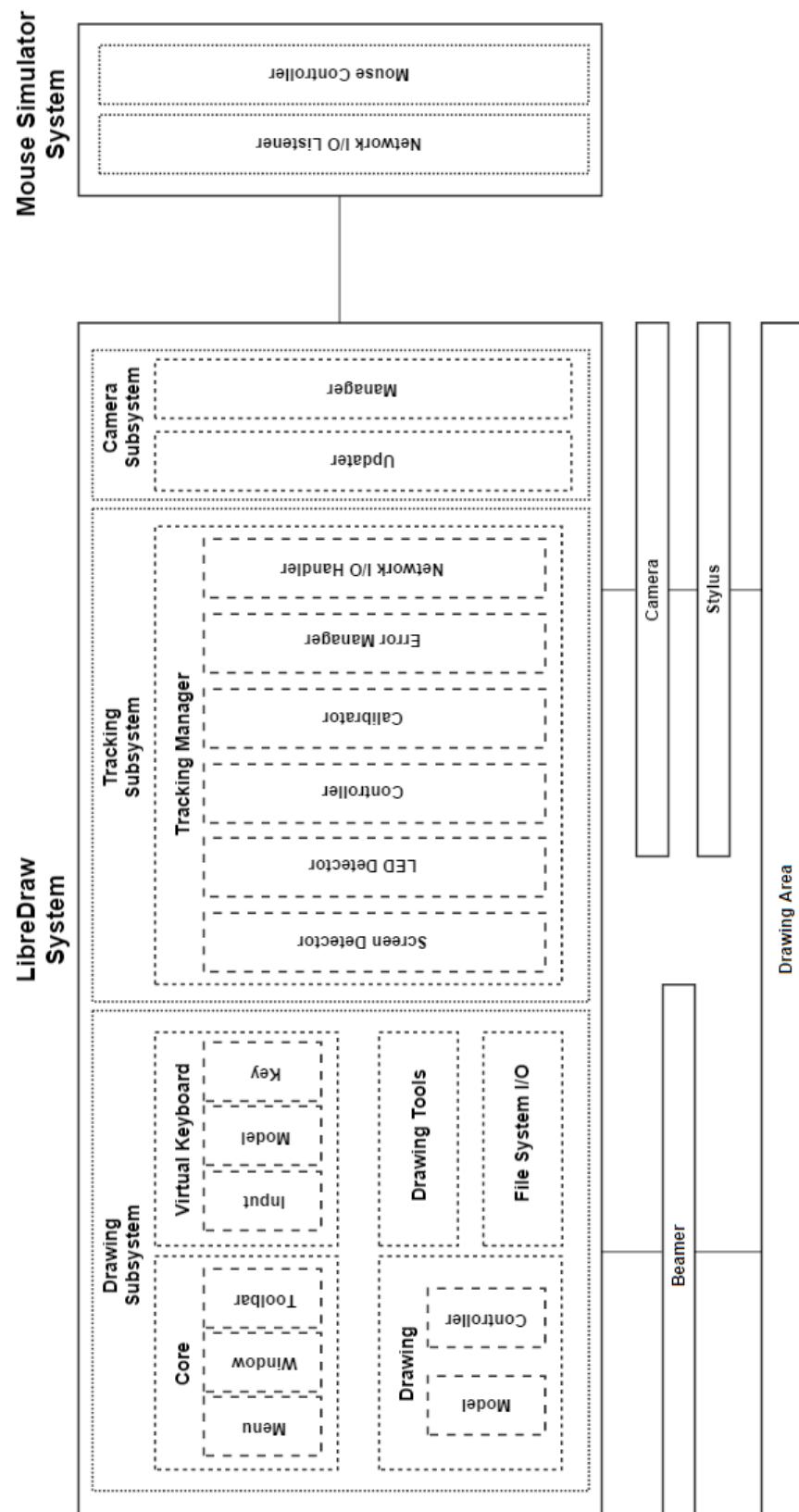


FIGURE 2.2 – Vue détaillée de l'architecture du système global

2.1.2 Diagramme d'activités

La figure suivante illustre l'activité globale liée à une esquisse utilisateur réalisée au moyen du stylet :

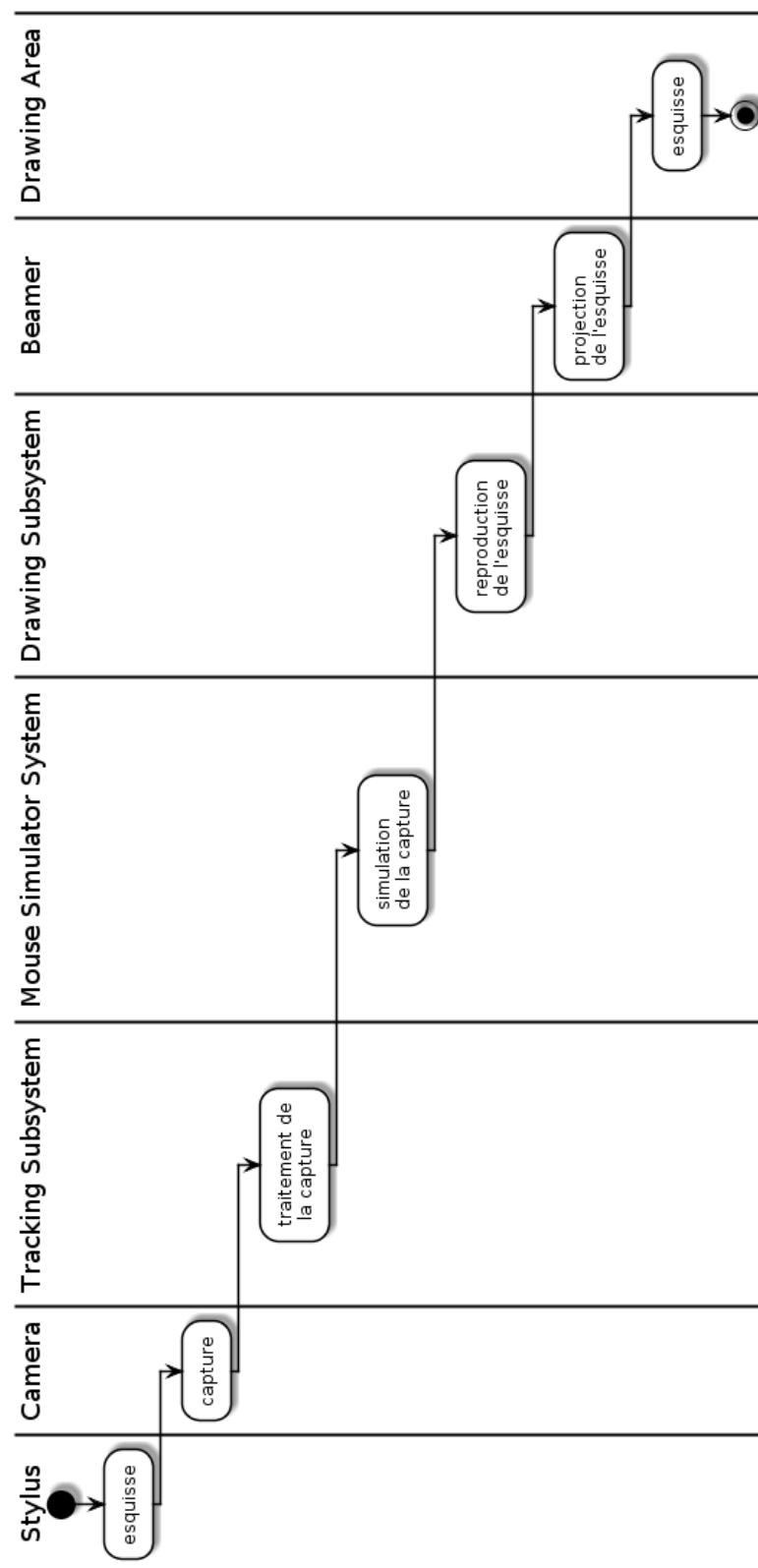


FIGURE 2.3 – Activité globale d'une esquisse utilisateur

2.2 LibreDraw System

Cette section présente le détail de la conception du système **LibreDraw System**. Cette présentation débute par l'introduction de l'architecture de ce système. Celle-ci est bien plus détaillée de manière à approfondir progressivement la description du système. Chaque constituant du système fait également l'objet d'une pareille décomposition et d'une pareille description. Ce système correspond à l'application de dessin à proprement parlé. Pour rappel, l'application se veut être un outil de dessin assisté par ordinateur permettant à l'utilisateur de réaliser ses dessins de la manière la plus naturelle possible. Il doit pouvoir dessiner directement sur sa table ou n'importe quelle autre surface plane à l'aide d'un stylet. Son dessin doit pour sa part être projeté sur son plan de travail, donnant à l'utilisateur l'impression de dessiner avec un crayon et une feuille. Une telle application nécessite bien évidemment la mise en place d'une infrastructure adéquate. Toutefois, cette infrastructure doit pouvoir être exploitée par l'application.

Le mécanisme de suivi du stylet étant relativement complexe, il ne peut en aucune manière être exécuté sans passer au préalable par une phase d'initialisation et de configuration. En effet, différents objectifs doivent être atteints initialement de manière à ce que le système de tracking soit non seulement en mesure d'interagir avec le matériel mais en plus, que son interaction s'effectue de la manière la plus optimale possible. Le système doit être capable de s'adapter à un certain nombre de contraintes. Celles-ci peuvent être d'ordre matériel. En effet, malgré la présence conseillée d'une caméra et d'un projecteur, il n'en demeure pas moins qu'elle n'est pas obligatoire. En conséquence, l'application doit être en mesure de s'adapter lors d'une telle situation. Par ailleurs, le fait de disposer d'une caméra connectée au système hôte n'est pas suffisant pour que le système de tracking soit en mesure d'interagir de manière performante avec l'ensemble de l'infrastructure. L'angle et la position de la caméra étant des informations tout à fait variables, elles doivent être prises en compte lors de l'initialisation du mécanisme de tracking.

Le suivi du stylet n'est également pas une tâche facile. Le mécanisme de tracking doit être en mesure de détecter à tout moment le stylet tant qu'il reste dans le champ de vision de la caméra et ce, indépendamment de la gestuelle de l'utilisateur. Le principe de détection de stylet défini lors de l'établissement du cahier des charges reposant sur la détection de LED, celui-ci découle à un certain nombre de problèmes potentiels et de questions devant être imaginées, supposées et répondues à l'avance. Une de ces questions est le comportement du mécanisme de tracking lorsque la LED du stylet présente la même couleur que le contenu du dessin de l'utilisateur. Ainsi, le dessin et la LED risqueraient d'être confondues. Telles sont les problématiques auxquelles le système de tracking doit faire face. L'ergonomie de l'application et l'expérience utilisateurs sont tout autant des points à soulever. L'interaction de l'utilisateur avec l'application s'effectuant avec un stylet, celle-ci peut être sujette à des problèmes de précision. Par ailleurs, les situations de saisies d'information par l'utilisateur doivent être rendues aussi facile que possible. En effet, forcer l'utilisateur à quitté son support de travail pour être en mesure de sauvegarder son dessin par le biais de son clavier physique est une contrainte qui est loin d'être appréciable. De ce fait, une solution permettant à l'utilisateur d'accomplir une telle tâche en se passant de son clavier doit être proposée. Le fait que les faits et gestes de l'utilisateur soient captés par une caméra vidéo et interprétés nécessite de reproduire l'esquisse de dessin déclenchée par l'utilisateur dans l'application de dessin. Le dessin esquissé par l'utilisateur étant projeté via un projecteur sur son support de travail, le dessin doit donc être reproduit logiquement et visuellement au sein de l'application.

2.2.1 Architecture

La figure suivante illustre l'architecture du système **LibreDraw System** :

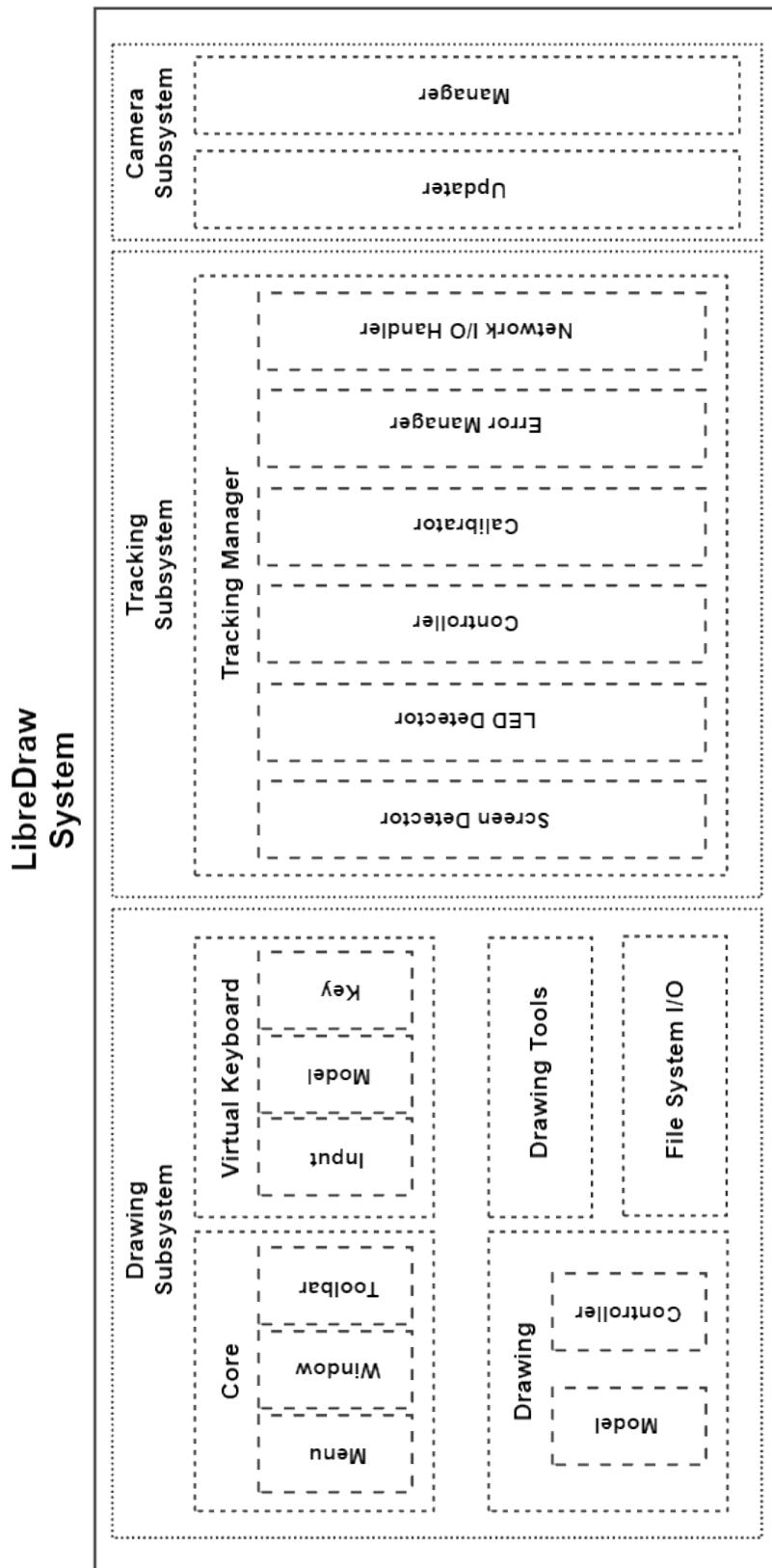


FIGURE 2.4 – Architecture du système LibreDraw System

Le système **LibreDraw System** est divisé en trois sous-systèmes distincts. Le premier sous-système est **Camera Subsystem**. Celui-ci a pour tâches de gérer l’initialisation, la configuration et sélection de la caméra utilisateur devant servir à capturer les mouvements du stylet. La partie initialisation et configuration est assurée par le constituant *Manager*. Celui-ci fait appel au constituant *Updater* pour proposer une liste de caméras disponibles susceptibles d’être utilisées en guises de dispositif d’enregistrement vidéo de l’application.

Le second sous-système du système **LibreDraw System** est **Tracking Manager**. Celui-ci est constitué d’un constituant principal qui est le gestionnaire du tracking de stylet. Il a pour tâche principale, comme son nom l’indique, de gérer le tracking du stylet, c’est-à-dire d’assurer en permanence son suivi au sein du champ de vision de la caméra.

Le dernier sous-système du système **LibreDraw System** est **Drawing Subsystem**. Celui-ci gère entre autres toute la partie interface graphique utilisateur de l’application de dessin. Il dispose en premier lieu d’un constituant nommé *Core* qui fait office de cœur de l’interface. Il met également à disposition un constituant *Virtual Keyboard* qui fournit comme son nom l’indique un clavier virtuel. Celui-ci est sollicité entre autres lors de chaque saisie utilisateur.

Le dessin de l’utilisateur est représenté pour sa part par le constituant *Drawing*. Ce dernier gère toute la partie reproduction de dessin aussi bien au niveau logique que graphique. Il délègue toutefois l’application de l’esquisse réalisée à l’aide d’un outil à des éléments du constituant *Drawing Tools*. Finalement, le dernier constituant composant ce sous-système est *File System I/O* qui met à disposition des mécanismes de sérialisation de dessin.

2.2.2 Drawing Subsystem

Cette section présente le détail de la conception du sous-système **Drawing Subsystem** du système **LibreDraw System**. Cette section présente le détail de la conception du sous-système **Drawing Subsystem**. Cette présentation débute par l'introduction de l'architecture de ce système. Celle-ci est bien plus détaillée de manière à approfondir progressivement la description du système. Chaque constituant du système fait également l'objet d'une pareille décomposition et d'une pareille description.

L'interface graphique utilisateur doit concrètement proposer une fenêtre principale dans laquelle il est possible de dessiner. Bien évidemment, il est attendu de l'utilisateur qu'il dessine à l'aide du stylet sur son support de travail. Toutefois, il ne lui est pas impossible de travailler directement sur l'application au moyen de la souris de son ordinateur. Ce n'est évidemment pas jouir de la fonctionnalité principale de l'application de dessin. Néanmoins, aucune contrainte de cet ordre n'est imposée à l'utilisateur quant à l'esquisse de son dessin.

L'esquisse d'un dessin doit pouvoir être effectuée dans une zone appropriée laquelle correspond en un espace ouvert où l'utilisateur peut esquisser ce qu'il désire et ce, à sa guise. Par ailleurs, il doit disposer d'une panoplie d'outils lui permettant d'être inventif et créatif vis-à-vis de son dessin tant sur sa forme que sur son contenu. Les points suivants récapitulent les outils devant être mis à disposition :

- Outils de dessin : les outils de dessin mettent à disposition un crayon et une gomme permettant de dessiner sans restriction n'importe quelle forme géométrique.
- Outils de formes : les outils de formes mettent à disposition un ensemble de formes géométriques prédéfinies pouvant être dessinées au sein d'un dessin et redimensionnées à la guise de l'utilisateur.
- Outils de couleurs : les outils de couleurs mettent à disposition une palette de couleurs laquelle permet de définir la couleur du trait aussi bien pour les outils de dessin que pour les outils de formes.
- Outils d'épaisseur : les outils d'épaisseur permettent de définir selon une liste prédéfinie l'épaisseur du trait aussi bien pour les outils de dessin que pour les outils de formes.

Le dessin esquissé par l'utilisateur doit pouvoir être sauvegardé de manière à pouvoir y retravailler dessus ultérieurement. De ce fait, des boîtes de dialogue adaptées à l'utilisation d'un stylet doivent être proposées de manière à pouvoir effectuer de telles opérations. De plus, ces boîtes de dialogue doivent permettre l'utilisateur d'y saisir aisément du texte. En conséquence, une solution de saisie doit être proposée et celle-ci doit être en adéquation avec l'utilisation d'un stylet lors de l'accomplissement d'une telle tâche.

En dernière instance, l'interface graphique utilisateur doit être ergonomique, facile d'utilisation et agréable à l'utilisateur. Le point essentiel est l'aisance avec laquelle il doit être en mesure d'interagir avec l'interface graphique de l'application en recourant au stylet. En conséquence, l'élaboration de toutes les fenêtres, boîtes de dialogues et autres doivent impérativement être imaginées de telle sorte que l'utilisation du stylet remplace définitivement celle de la souris de l'ordinateur.

2.2.2.1 Architecture

La figure suivante illustre l'architecture du sous-système **Drawing Subsystem** :

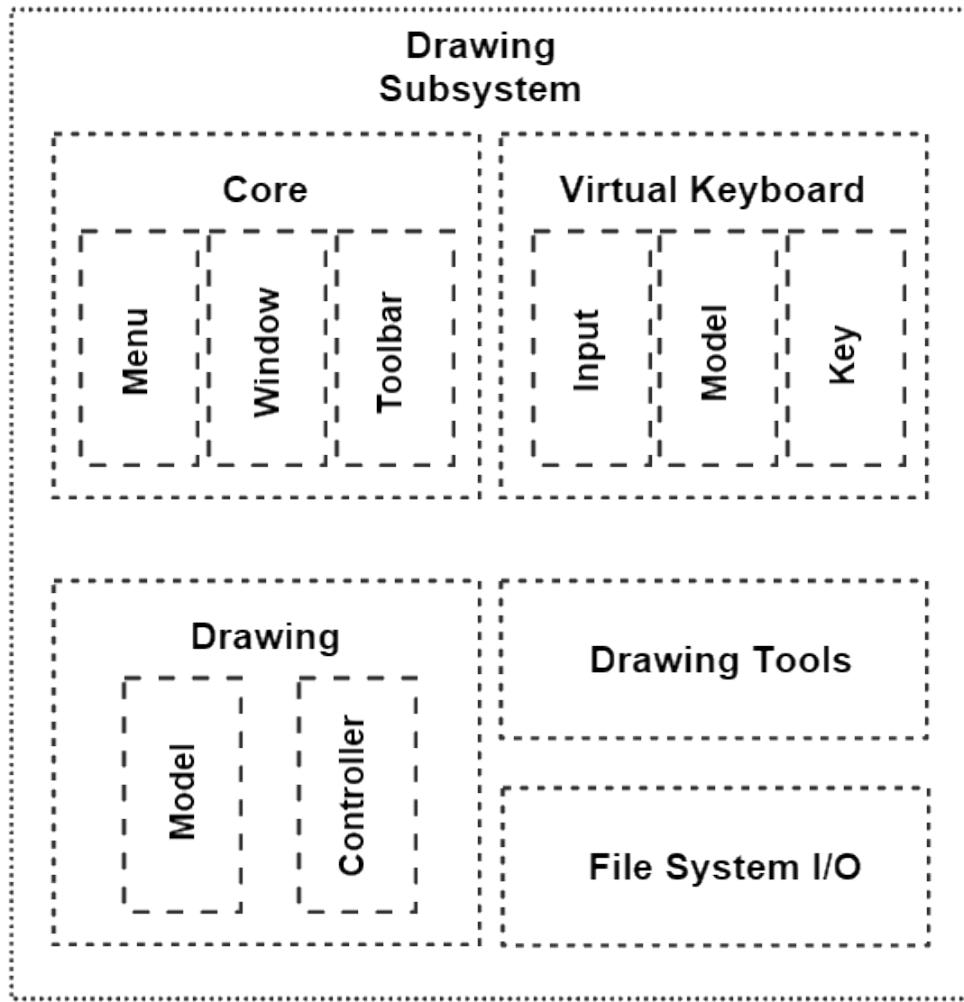


FIGURE 2.5 – Architecture du sous-système Drawing Subsystem

Le sous-système **Drawing Subsystem** gère entre autres toute la partie interface graphique utilisateur de l'application de dessin. Le composant *Core* correspond au cœur de ce sous-système et donc, à l'interface graphique de l'application. Celui-ci met effectivement à disposition tous les constituants graphiques nécessaires devant être combinés et regroupés ensemble pour former finalement l'interface utilisateur final. Il s'agit concrètement des constituants *Menu* qui fournit le menu de l'application de dessin, *Window* qui correspond à la fenêtre principale de l'application et enfin, *Toolbar* qui se révèle être la barre d'outils de cette même fenêtre.

La sérialisation d'un dessin est rendue possible par le constituant *File System I/O* qui fournit deux boîtes de dialogue permettant respectivement d'importer et de sauvegarder un dessin. La problématique liée à la saisie utilisateur par le biais du stylet est répondu grâce au constituant *Visual Keyboard*. Ce dernier fournit un clavier virtuel s'apparentant aux claviers usuels que l'on rencontre sur les applications des smartphones et des tablettes. Ainsi, les boîtes de dialogue mentionnées plus tôt intègrent ce clavier dans leur interface.

Les constituants *Drawing* et *Drawing Tools* collaborent ensemble en vue de produire le dessin esquissé par l'utilisateur. Pour ce faire, un élément *Controller* veille à appliquer l'esquisse sur le *Model* du dessin en prenant en compte l'outil courant proposé par le constituant *Drawing Tools*. À noter que, durant la totalité de son interaction avec l'interface graphique, l'utilisateur peut à tout moment recourir à la souris de son ordinateur pour réaliser les mêmes traitements qu'il était supposé réalisé initialement par le biais du stylet. Ainsi, la possibilité de s'abstenir d'utiliser le stylet demeure toujours possible pour l'utilisateur.

En définitive, le sous-système **Drawing Subsystem** fait collaborer l'ensemble de ses composants en vue de reproduire le dessin esquissé par l'utilisateur et de sérialiser ce dessin en recourant au clavier virtuel fournit par *Visual Keyboard*. Ce dernier se révèle être en parfaite adéquation avec une saisie utilisateur au moyen d'un stylet.

2.2.2.2 Core

Cette sous-section décrit le composant *Core*. Il s'agit de présenter son architecture, de la décomposer pour ensuite introduire individuellement ses constituants.

Le composant *Core* correspond en l'interface graphique de la fenêtre principale de l'application.

Architecture

La figure suivante illustre l'architecture du composant *Core* du sous-système **Drawing Subsystem** :

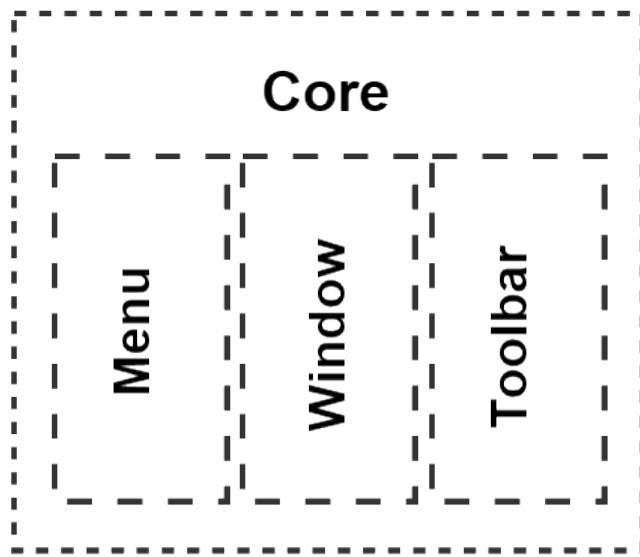


FIGURE 2.6 – Architecture du composant Core

Le composant *Core* a pour base l'élément *Window* qui consiste en la fenêtre principale de l'application. Dans cette fenêtre figure l'élément *Toolbar* faisant office de barre d'outils de la fenêtre. Cette barre d'outils permet de sélectionner un outil de dessin, de forme ou d'exécuter l'outil d'épaisseur, les outils d'historique des actions ou encore l'outil palette de couleurs. Par ailleurs, cette même barre permet d'accéder au menu de l'application représenté par l'élément *Menu*.

Window

La figure suivante illustre la structure de l'élément *Window* :

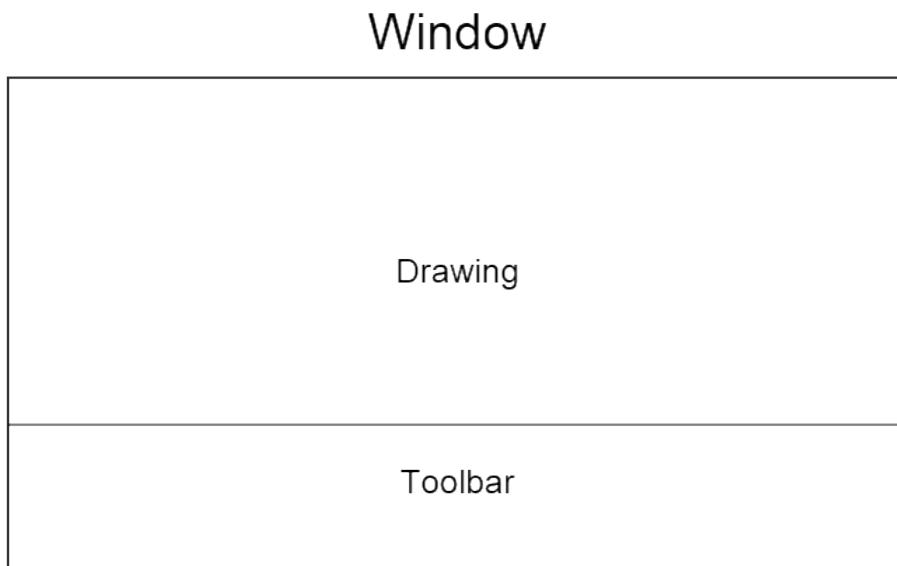


FIGURE 2.7 – Structure de l'élément Window

L'élément *Window* est composé d'une part par le composant *Drawing* censé contrôlé et stocké le dessin de l'utilisateur. D'autre part, il est caractérisé par l'élément *Toolbar* laquelle met à disposition un ensemble d'outils susceptibles de contribuer à l'esquisse du dessin, à modifier certaines propriétés de l'élément *Controller* du composant *Drawing* comme l'épaisseur du trait. Par ailleurs, ce même élément *Toolbar* permet d'accéder à l'élément *Menu*.

Menu

La figure suivante illustre la structure de l'élément *Menu* :

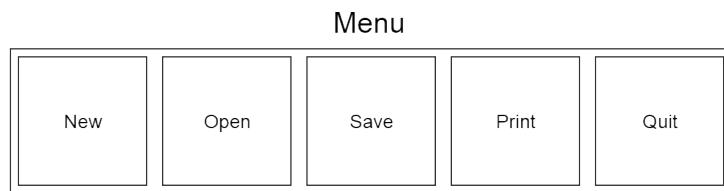


FIGURE 2.8 – Structure de l'élément Menu

L'élément *Menu* est accessible via l'élément *Window* par le déclenchement d'un outil adéquat de l'élément *Toolbar*. Cet élément *Menu* est composé pour sa part par des boutons lesquels permettent d'exécuter des fonctionnalités données liées à la manipulation usuelle d'un document comme sa création, sa sauvegarde et autres.

Toolbar

La figure suivante illustre la structure de l'élément *Toolbar* :

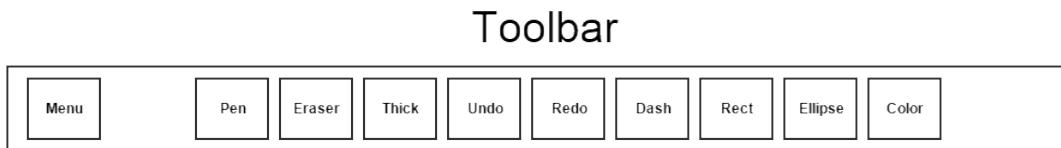


FIGURE 2.9 – Structure de l'élément Toolbar

L'élément *Toolbar* est constitué d'un ensemble de boutons. Un premier bouton s'écarte toutefois du lot ; il s'agit du bouton avec pour label "Menu". Celui-ci permet évidemment d'ouvrir l'élément *Menu* de la fenêtre. Le reste des autres boutons est à part. Certains d'entre eux permettent d'esquisser des formes géométriques ou de dessiner à main levée. D'autres permettent de personnaliser les traits tant au niveau de la couleur que de l'épaisseur. Enfin, certains permettent d'agir sur l'historique des actions, annuler ou rétablir une action autrement dit.

2.2.2.3 Virtual Keyboard

Cette sous-section décrit le composant *Virtual Keyboard*. Il s'agit de présenter son architecture, de la décomposer pour ensuite introduire individuellement ses constituants.

Le composant *Virtual Keyboard* correspond en un clavier virtuel lequel permet à l'utilisateur de saisir des informations au sein par le biais du stylet.

Architecture

La figure suivante illustre l'architecture du composant *Virtual Keyboard* du sous-système **Drawing Subsystem** :

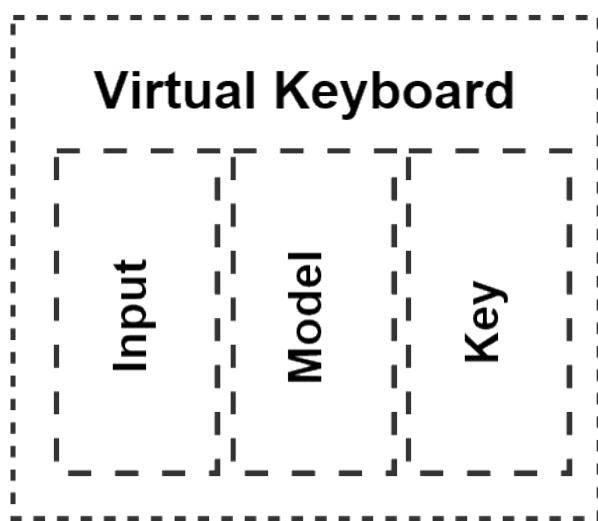


FIGURE 2.10 – Architecture du composant Virtual Keyboard

Le composant *Virtual Keyboard* est caractérisé avant tout par un élément *Model*, correspondant au modèle du clavier, c'est-à-dire à la disposition de ses touches. Chacune des touches du clavier correspondent pour leur part à des éléments *Key*. L'élément *Input* consiste en ce qui le concerne en un champ texte éditable lequel est lié à un clavier virtuel. Autrement dit, la saisie de texte dans ce champ n'est possible que par le composant *Virtual Keyboard* qui lui aurait été associé.

Model

L'élément *Model* correspond pour le composant *Virtual Keyboard* la disposition de ses touches représentés par des éléments *Key*.

Les figures suivantes illustrent l'agencement des touches du clavier :

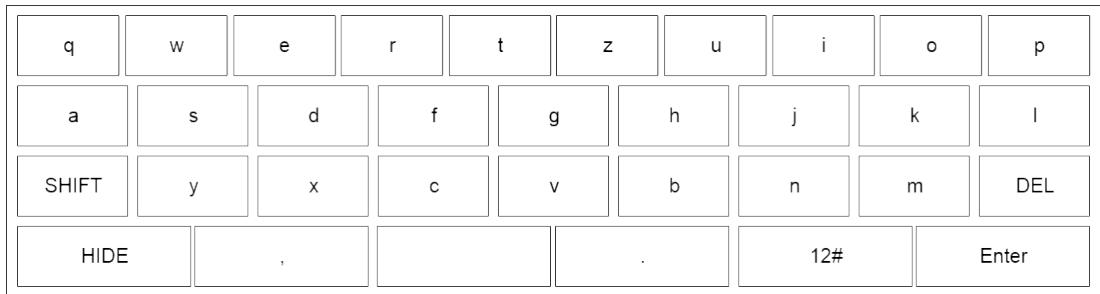


FIGURE 2.11 – Agencement des touches du clavier virtuel selon le mode alphabétique

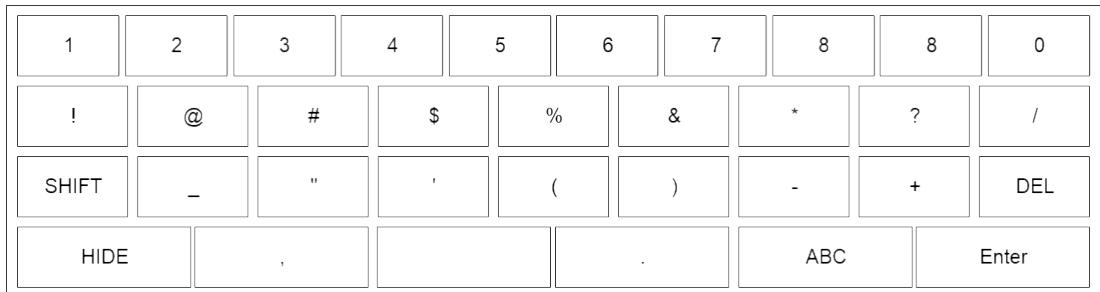


FIGURE 2.12 – Agencement des touches du clavier virtuel selon le mode symbole

Key

L'élément *Key* représente une touche du clavier virtuel représenté par le composant *Virtual Keyboard*. Ce dernier faisant l'objet de deux agencements, certains de ses éléments *Key* sont amenés à changer de valeur. Ainsi, on dénombre deux modes différents pour un élément *Key* :

- Le mode mono-évalué : en mode symbole ou en mode alphabétique, la même valeur est retournée.
- Le mode multi-évalué : en mode symbole ou en mode alphabétique, une valeur différente est retournée.

Les figures suivantes illustrent les deux formes possibles que peut prendre un élément *Key* :

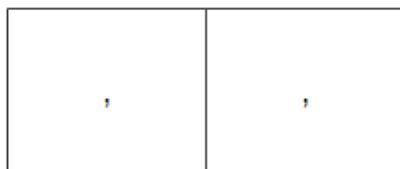


FIGURE 2.13 – Elément Key mono-evalué

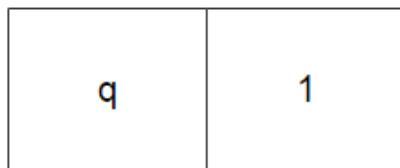


FIGURE 2.14 – Elément Key multi-evalué

Input

L'élément *Input* correspond en un champ texte éditable uniquement par le biais d'un composant *Virtual Keyboard*. Ainsi, lors du focus sur ce champ texte, le clavier virtuel est aussitôt affiché.

2.2.2.4 Drawing

Cette sous-section décrit le composant *Drawing*. Il s'agit de présenter son architecture, de la décomposer pour ensuite introduire individuellement ses constituants.

Le composant *Drawing* correspond au dessin de l'utilisateur.

Architecture

La figure suivante illustre l'architecture du composant *Drawing* du sous-système **Drawing Subsystem** :

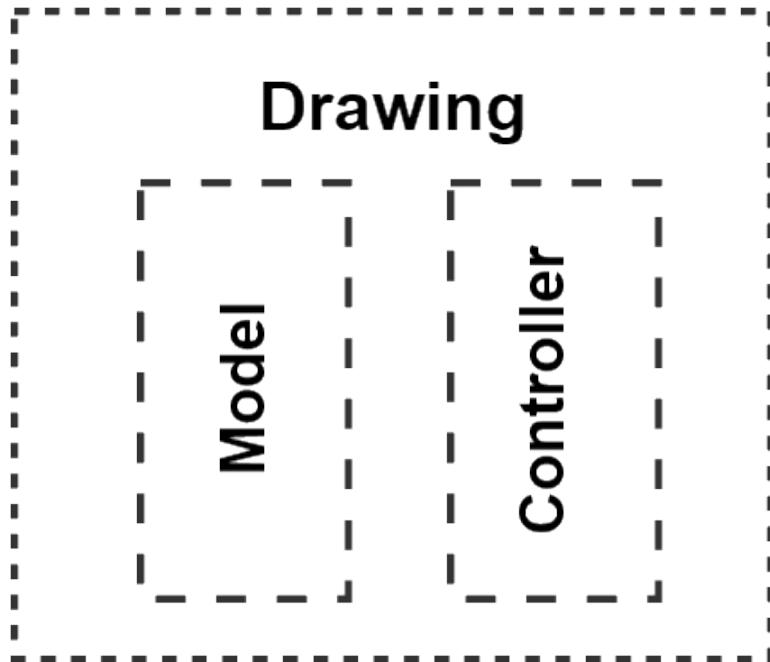


FIGURE 2.15 – Architecture du composant Drawing

Le composant *Drawing* est composé d'un élément *Model* qui représente le dessin aussi bien logiquement que visuellement. Ce composant est également caractérisé par l'élément *Controller*. Celui-ci assure la reproduction des esquisses de l'utilisateur sur demande du système **Mouse Simulator System**.

Model

L'élément *Model* représente le dessin de manière logique et graphique. Il se repose sur une structure de type *bitmap*, c'est-à-dire une matrice où chaque élément correspond à un pixel. La valeur stockée dans un tel élément correspond à une couleur. Cette couleur est représentée quant à elle par un code hexadécimal. En guise d'exemple, la figure suivante illustre une telle structure.

```
0000FF 0000FF 0000FF 0000FF 0000FF 0000FF 0000FF 0000FF  
0OFFOO FF0000 FFFFFF FF0000 FFFFFF FF0000 FFFFFF OOFFOO  
0OFFOO FFFFFF FF0000 FFFFFF FF0000 FFFFFF FF0000 OOFFOO  
0OFFOO FF0000 FFFFFF FF0000 FFFFFF FF0000 FFFFFF OOFFOO  
0OFFOO FFFFFF FF0000 FFFFFF FF0000 FFFFFF FF0000 OOFFOO  
0OFFOO FF0000 FFFFFF FF0000 FFFFFF FF0000 FFFFFF OOFFOO  
0OFFOO FFFFFF FF0000 FFFFFF FF0000 FFFFFF FF0000 OOFFOO  
0000FF 0000FF 0000FF 0000FF 0000FF 0000FF 0000FF 0000FF
```

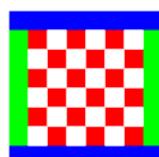


FIGURE 2.16 – Structure logique d'un dessin

Controller

L'élément *Controller* veille à la bonne reproduction des esquisses de l'utilisateur lorsqu'il est sollicité par le système **Mouse Simulator System**. Il délègue la reproduction de l'esquisse à un élément du composant *Drawing Tools*.

2.2.2.5 Drawing Tools

Le composant *Drawing* correspond en un ensemble d'éléments associant aux outils accessibles via le composant *Core* un contrôleur respectif. Ce contrôleur reproduit le geste de l'utilisateur selon l'outil sélectionné au moment donné. Il est à noter toutefois que cette association ne concerne que les outils susceptibles d'ajouter du contenu au dessin, esquisser à main levée ou esquisser des formes géométriques autrement dit.

2.2.2.6 File System I/O

Le composant *File System I/O* met à disposition de l'application, et de l'utilisateur dans une moindre mesure, des mécanismes de sérialisation de dessin. Ceux-ci sont rendus accessibles à l'utilisateur par des interfaces graphiques se présentant sous la forme de boîtes de dialogue. Compte tenu du fait que ces boîtes de dialogue attendent de l'utilisateur de la saisie textuelle, les interfaces en question intègrent le composant *Virtual Keyboard* lequel fournit un clavier virtuel permettant l'édition de champ texte et ce, en recourant au stylet.

2.2.3 Tracking Subsystem

Le sous-système **Tracking Subsystem** est un composant essentiel de l’application ayant pour but de gérer le pointeur de l’utilisateur. En effet, étant donné que ce type de pointeur n’est pas une entrée standard comme pourrait l’être une souris ou un clavier, l’application a besoin d’un module prenant en charge exclusivement le suivi du stylet.

Ce module doit fournir une abstraction suffisante au **Drawing Subsystem** afin que les modules aient un minimum de dépendance et donc, que l’évolution d’un module n’influence pas l’autre.

2.2.3.1 Architecture

Les composants de ce sous-système sont les suivants :

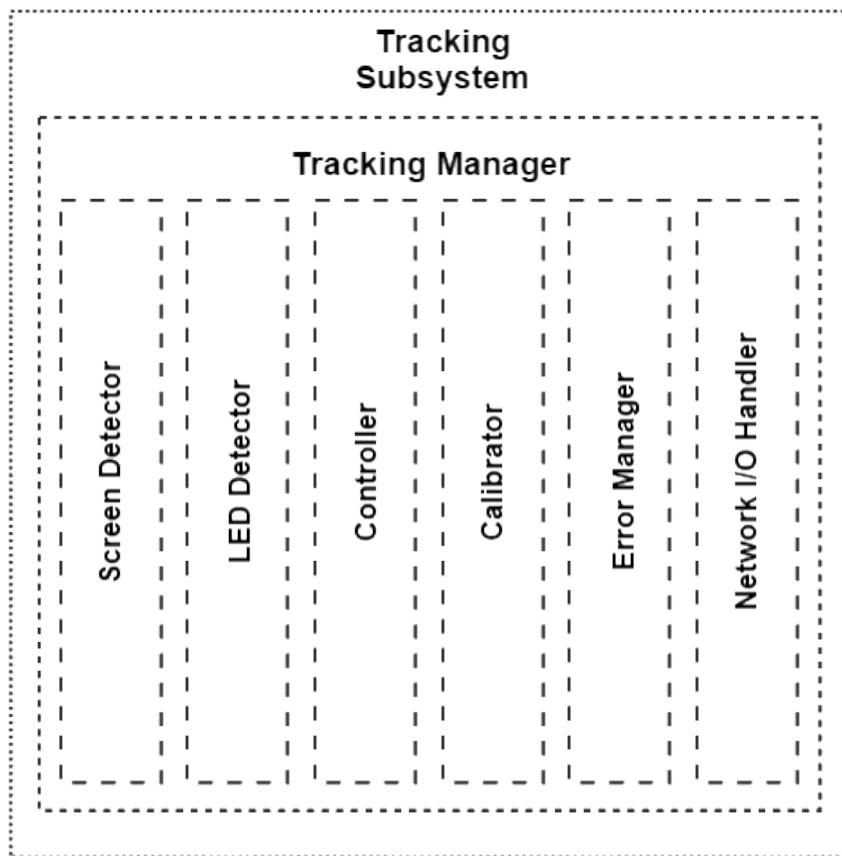


FIGURE 2.17 – Architecture du sous-système Tracking Subsystem

Ils peuvent être classé du niveau le plus bas au plus élevé :

- *Calibrator* : ce constituant initialise et calibre l’infrastructure matériel de manière à ce qu’il puisse interagir avec l’application et inversement.
- *LED Detector* : ce constituant a pour fonction de détecter et traquer en permanence la/les LED(s) du stylet de manière à le localiser dans le champ de vision de la caméra. Cela est utilisé aussi bien lors de la phase d’initialisation du système que lors de sa phase post-initialisation, c’est-à-dire lors du suivi du stylet.

- *Screen Detector* : ce constituant a pour fonction de détecter l'écran de l'utilisateur. Cela est notamment utilisé lors de la phase préliminaire de calibrage du matériel. Il permet de décrire la position de la LED dans le référentiel de l'écran.
- *Error Manager* : ce constituant fait office de gestionnaire d'erreur. Il est notamment sollicité lors d'une erreur rencontrée durant le processus de calibrage.
- *Controller* : ce constituant contrôle les mouvements et les actions du stylet selon les informations reçues par la caméra et selon leur analyse également.
- *Network I/O Handler* : ce constituant fait office d'interface de communication réseau entre les systèmes **LibreDraw System** et **Mouse Simulator System**. Il est notamment sollicité une fois le traitement d'informations liés aux faits et gestes du stylet terminé afin de les reproduire au sein de la partie dessin du système gérée entre autres par le sous-système **Drawing Subsystem**.

Comme nous pouvons le voir, une fois que le calibrage est effectué, la caméra pourra détecter la LED du stylet. Ensuite, cette position sera transformée pour que l'on connaisse sa position par rapport à l'image du vidéo-projecteur (donc du programme) et sera ensuite utilisée pour envoyer des événements de mouvement de souris via le réseau.

2.2.3.2 Controller

Le rôle du **Controller** est d'interpréter les informations du **LED Detector** afin de définir si l'utilisateur vient d'allumer, de bouger ou d'éteindre la LED de son pointeur, pour ensuite émettre les bons événements associés : appuyer, bouger et relâcher le bouton de la souris.

Il a donc le rôle de traducteur d'événements.

2.2.3.3 Screen Detector

Le **Screen Detector** est un composant permettant de détecter l'écran vu depuis la caméra et d'en calculer les déformations par rapport à l'image qu'affiche le vidéo-projecteur. En effet, le pointeur de l'utilisateur se déplace dans le monde réel en trois dimensions, mais la caméra ne voit qu'une projection bidimensionnelle de ce monde et le projecteur envoie une image également bidimensionnelle. Dans le cas où la caméra et le projecteur étaient exactement alignés, ils auraient la même perception du monde et aucun calcul ne serait nécessaire. Or, ce n'est pas le cas : la caméra peut se trouver plusieurs dizaines de centimètres à côté du projecteur. Ainsi, une matrice de transformation permettant de passer d'un référentiel à l'autre doit être calculé par le programme. En termes plus mathématiques, cette matrice nous permet d'effectuer une rotation tridimensionnelle, un déplacement et une mise à l'échelle de l'image afin de reporter les points du référentiel de la caméra à celui du programme.

2.2.3.4 LED Detector

Ce composant gère le suivi de la LED. Il peut être considéré comme bas niveau dans le sens qu'il s'occupe de traiter les données brutes venant directement de la caméra. Il y a donc plusieurs traitements à effectuer sur les images afin d'en faire ressortir le peu d'information utile, c'est-à-dire une position bidimensionnelle dans le référentiel de la caméra. Ces traitements ont été une partie très importante de ce travail car ils demandent beaucoup de tests dans des conditions différentes, ainsi que de l'imagination

pour trouver la meilleure méthodologie à appliquer. Du temps a aussi été nécessaire pour trouver les paramètres les plus optimaux de nos filtres. Certains paramètres sont d'ailleurs fixes alors que d'autres sont calculés durant la phase de calibration.

2.2.3.5 Calibrator

Cette partie du programme s'occupe de calibrer le suivi de la LED en estimant et redéfinissant certains paramètres du **LED Detector**. En effet, selon l'environnement (éclairage, caméra, projecteur, LED, etc.), le suivi de la LED peut s'avérer plus ou moins difficile. Certaines caméras s'adaptent très bien à la luminosité et à la teinte de l'image projetée alors que d'autres ont plus de mal à s'adapter. Le **Calibrator** a pour mission de contrecarrer ses différences en utilisant divers méthodes de traitement de l'image.

2.2.3.6 Error Manager

Le gestionnaire d'erreur est un petit composant permettant simplement de détecter quand une erreur de calibrage est survenue et d'en avertir l'utilisateur. En outre, il offre la possibilité de relancer le processus de choix de caméra et de calibrage.

2.2.3.7 Network I/O Handler

Étant donné que nous voulions que la souris soit le plus proche possible du système d'exploitation, tout en restant multi-plateforme, nous avons exploré plusieurs possibilités offertes par les frameworks déjà utilisés, mais aucune solution n'était assez puissante et pratique par rapport à ce que nous souhaitions faire. Entre autres, l'un de nos souhait était que la souris nous permette aussi de cliquer sur les éléments propres au système d'exploitation et pas uniquement ceux de notre programme.

Nous avons alors choisi d'utiliser un petit programme en Java afin de déplacer la souris et de cliquer selon une position donnée. Afin d'établir une connexion entre l'utilitaire Java et notre application, nous avons choisi d'utiliser des sockets. Cela nous aurait permis de pouvoir changer de stratégie très facilement dans le cas où cette solution aurait soulevé de nouveaux problèmes.

2.2.4 Camera Subsystem

Le sous-système **Camera Subsystem** permet de gérer la/les caméra(s) de l'utilisateur. Il s'occupe de détecter les différentes caméras connues du système d'exploitation et d'afficher à l'utilisateur leurs images afin de permettre à ce dernier de sélectionner la caméra qu'il souhaite utiliser.

Il peut paraître étrange de parler des caméras (au pluriel). Pourtant, il n'en est rien puisqu'un grand nombre d'ordinateurs portables possèdent des caméras intégrées. Or, une caméra externe, se branchant sur un port série, peut s'avérer très pratique car l'ordinateur n'a pas besoin d'être déplacer. Il suffit de viser l'écran de projection avec la caméra externe. C'est pourquoi un système de sélection de caméra a été mis en place.

2.2.4.1 Architecture

Ce sous-système est assez simple. Il consiste en deux partie, le **Manager** et l'**Updater**.

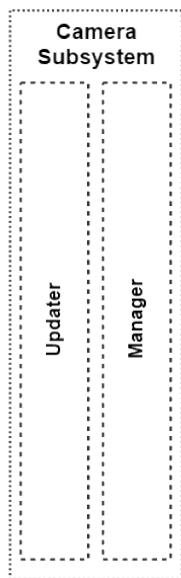


FIGURE 2.18 – Architecture du sous-système Tracking Subsystem

2.2.4.2 Manager

Le **Manager** s'occupe de détecter, puis d'initialiser les caméras de l'utilisateur. Il va ensuite créer les différentes fenêtres qui afficheront plus tard les vues en direct des caméras. Cette affichage sera gérée par un fil d'exécution séparé, l'**Updater**.

2.2.4.3 Updater

Le rôle de l'**Updater** est de récupérer les images des caméras et mettre à jour les fenêtres de sélection de caméra en direct. Cela donne alors un aperçu en direct de toutes les caméras, permettant à l'utilisateur de replacer sa caméra si besoin (par exemple si l'écran n'est pas totalement visible par la caméra) et de la choisir.

2.3 Mouse Simulator System

Le système *Mouse Simulator System* permet de simuler les actions de l'utilisateur avec le stylet, le but étant de rendre possible la reproduction du dessin de l'utilisateur dans l'application de dessin. Pour ce faire, ce système prend le contrôle de la souris de l'ordinateur et reproduit avec les gestes et les actions qu'a pu effectué l'utilisateur avec le stylet.

2.3.1 Architecture

La figure suivante illustre l'architecture du système **Mouse Simulator System** :

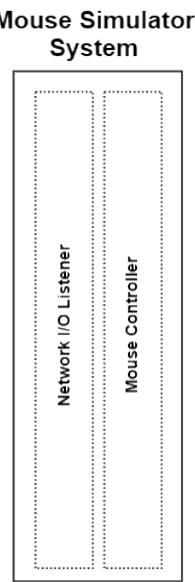


FIGURE 2.19 – Architecture du système Mouse Simulator System

Le **Mouse Simulator System** est caractérisé par les composants *Network I/O Listener* et *Mouse Controller*. Le premier est à l'écoute du système de suivi de stylet qui lui communique des informations quant au tracking. Le second reproduit les faits et gestes de l'utilisateur en les simulant par une prise de contrôle de la souris de l'ordinateur.

2.3.2 Network I/O Listener

Le composant *Network I/O Listener* a pour tâche d'écouter sur le réseau tout message en provenance du sous-système **Tracking Subsystem** du système **Drawing System**. Ces messages sont censés représenter le dernier état du stylet, c'est-à-dire ses coordonnées et l'éventuelle action émise.

2.3.3 Mouse Controller

Le composant *Mouse Controller* permet de déclencher la simulation de la souris lors de la réception de messages par le composant *Network I/O Listener*.

2.4 Beamer

L'élément *Beamer* est un composant matériel. Il permet la projection du dessin représenté par le composant *Drawing* du système **LibreDraw System**. Il est à noter que cet élément n'est pas essentiel au fonctionnement du système dans son ensemble.

2.5 Camera

L'élément *Camera* est un composant matériel. Il permet de capturer les faits et gestes de l'utilisateur réalisés par le biais du stylet à condition de se trouver dans le champ de vision de ce dispositif matériel. Les informations capturées par cet élément sont transmises au composant *Tacking Manager* du système **LibreDraw System**. Cette transmission est réalisée de manière à pouvoir par la suite reproduire l'action utilisateur aussi du sous-système **Drawing Subsystem**.

2.6 Stylus

L'élément *Stylus* correspond au stylet interagissant avec le système **LibreDraw System**. L'utilisateur l'exploite pour esquisser virtuellement un dessin sur l'élément *Drawing Area*.

2.7 Drawing Area

L'élément *Drawing Area* correspond au support de travail de l'utilisateur. Il s'agit de la surface, à priori plane, sur laquelle l'utilisateur esquisse son dessin d'une part et constate la projection de la reproduction de son esquisse.

2.8 Stylet

Cette section décrit la conception du stylet devant remplacer la souris de l'ordinateur pour dessiner sur un support physique. Sa conception ayant évolué en cours de projet, les différentes variantes y sont présentées. Chacune suit la même procédure de description. Les différents composants constituant ce stylet sont décrits de manière détaillée. Cela permet à tout un chacun de pouvoir reproduire cet outil en suivant les mêmes étapes de fabrication avec les mêmes composants voire des composants équivalents. Le prototype issu de la fabrication de l'outil de dessin est par la suite exposé. Finalement, le principe d'utilisation du stylet est présenté et commenté ce qui constitue un guide d'utilisation de celui-ci par la même occasion. Certains choix de conception y sont d'ailleurs justifiés.

2.8.1 Modèle à une LED

Le modèle à une LED du stylet a été défini lors de la définition du cahier des charges.

2.8.1.1 Composants

Le premier modèle du stylet est caractérisé par les composants suivants :

- Une LED rouge

- Un interrupteur
- Une résistance
- Une batterie
- Un boîtier

2.8.1.2 Prototype

La figure suivante propose une ébauche de prototype du modèle à une LED du stylet :



FIGURE 2.20 – Prototype initial du stylet

2.8.1.3 Principe d'utilisation

Ce modèle de stylet aurait dû pouvoir être utilisé comme un gros crayon, mais l'utilisateur aurait dû faire attention à l'inclinaison du stylet (voir ci-dessous). Son manque d'intuitivité aurait demandé à l'utilisateur de s'y habituer durant une période d'apprentissage indésirée.

2.8.1.4 Inconvénients

Après de rapides tests durant lesquels le modèle à une LED a été filmé, nous nous sommes rendus compte qu'il était parfois impossible de déterminer la position précise de la pointe du stylet en ayant pour unique information la position d'une seule LED. En effet, les mouvements du poignets lors du dessin étant complexes, l'inclinaison du stylet peut varier rapidement et l'alignement entre la LED et la pointe change. Nous avons alors imaginé l'évolution de ce modèle, le modèle à deux LEDs.

2.8.2 Modèle à deux LEDs

Le modèle à deux LED du stylet a été défini durant la première partie du projet. Il remplace le modèle à une LED.

2.8.2.1 Composants

Le deuxième modèle du stylet est caractérisé par les composants suivants :

- Une LED rouge
- Une LED verte
- Un interrupteur momentané
- Une résistance
- Une batterie
- Un boîtier

2.8.2.2 Fabrication

Le but est de faire en sorte que les deux LEDs s'allument lorsque l'interrupteur momentané est appuyé. Pour cela, nous avons simplement connecté en série l'interrupteur, la LED rouge, la LED verte et la résistance à la batterie. L'ensemble a ensuite été installé dans un boîtier modélisé à l'aide du logiciel Blender, puis créé à l'aide d'une imprimante 3D à extrusion.

2.8.2.3 Prototype

La figure suivante propose une ébauche de prototype du modèle à deux LEDs du stylet :



FIGURE 2.21 – Seconde version du prototype du stylet

2.8.2.4 Fonctionnement

Le fonctionnement de ce stylet est le même que celui à une LED. Simplement, une seconde LED permet de connaître l'orientation du stylet dans l'espace.



FIGURE 2.22 – Utilisation de notre prototype de stylet à deux LEDs

2.8.2.5 Principe d'utilisation

Ce modèle de stylet est censé pouvoir être utilisé comme un gros crayon. Il est alors relativement intuitif à utiliser et ne demande pas une longue période d'apprentissage.

2.8.2.6 Inconvénients

Après de nombreux essais avec ce modèle, nous avons remarqué qu'il était difficile d'obtenir une position calculée précise de la pointe du stylet en ne se basant que sur deux LEDs posées sur son côté. En effet, les problèmes d'inclinaisons n'étaient que partiellement résolu, et nous nous sommes rendus compte que la rotation du stylet avait un lourd impact sur la précision. Nous en avons conclu qu'il serait plus simple et plus efficace de suivre directement la pointe du stylet grâce à la caméra et nous nous sommes alors résolu à l'utilisation d'une LED jouant le rôle de mine ou à l'utilisation d'un pointeur laser. Les dispositifs étant tous deux perçus comme une tâche de lumière monochrome par la caméra.



FIGURE 2.23 – Exemple de problème d'alignement des LEDs avec la pointe

2.8.3 Pointeur lumineux

Le pointeur lumineux est une évolution du stylet qui a été défini durant la seconde partie du projet. Il remplace le modèle à deux LEDs.



FIGURE 2.24 – Exemple de pointeur lumineux à LED

2.8.3.1 Fabrication

Étant donné que nous avions déjà un pointeur laser puissant ainsi qu'un petite LED munie d'un interrupteur, nous n'avons pas eu besoin de fabriquer ce modèle. Néanmoins, fabriquer un tel prototype serait même plus facile que les modèles présentés précédemment.

2.8.3.2 Fonctionnement

Au lieu d'essayer de suivre des objets permettant d'extrapoler la position à laquelle l'utilisateur souhaite dessiner, ce système suit directement la position pointée par l'utilisateur à l'aide d'une source lumineuse suffisamment focalisée. Ainsi, l'utilisateur à le choix d'utiliser une LED pointée sur l'écran à une petite distance (moins de 5 centimètres) ou encore un pointeur laser assez puissant pour être détecté par la caméra.

2.8.3.3 Principe d'utilisation

L'utilisation du pointeur lumineux est relativement simple. Il suffit à l'utilisateur de pointer l'écran avec un dispositif émettant de la lumière monochrome, comme par exemple une LED de couleur ou un pointeur laser.

Cette méthode utilise la lueur de la lumière afin de détecté la position où l'utilisateur veut dessiner ou cliquer. Il est donc important d'avoir une tâche lumineuse assez grande et assez puissante.

2.9 Environnement de dessin

Cette section présente la conception de l'environnement de dessin. Ce travail de conception est nécessaire dans la mesure où le système ne peut être opérationnel et fonctionnel qu'à la seule condition de disposer d'un environnement de travail adéquat répondant à un certain nombre de pré-requis. Concrètement, il s'agit d'introduire le matériel nécessaire à sa mise en place, de décrire précisément celle-ci et de présenter un exemple d'environnement jugé idéal pour l'utilisation de l'application.

2.9.1 Matériel requis

Les points suivants constituent le matériel requis pour l'utilisation de l'application :

- Un prototype de stylet faisant office d'outil de dessin
- Une caméra permettant de traquer les mouvements du stylet
- Un projecteur vidéo permettant de retranscrire le dessin de l'utilisateur
- Un plan de travail permettant de dessiner
- Un support prévu pour la disposition de la caméra et du projecteur au-dessus du plan de travail

Plusieurs remarques s'imposent quant à cette énumération du matériel requis. En premier lieu, la caméra mentionnée correspond à n'importe quel dispositif permettant de capturer des informations vidéos ; il peut donc s'agir aussi bien d'une webcam, d'une micro-caméra d'un smartphone, d'une caméra intégrée à un ordinateur ou même d'une caméra professionnelle. La différence dans le choix de ce matériel influera toutefois la qualité du système de tracking du stylet. Par ailleurs, le projecteur vidéo n'est pas réellement obligatoire dans la mesure où l'utilisateur peut tout à fait dessiner sur un support donné. Il peut tout aussi bien constater le résultat de ses esquisses sur le moniteur de son ordinateur.

2.9.2 Mise en place du matériel

La mise en place du matériel requis est décrite par la procédure suivante :

1. Disposez le support de travail.
2. Posez le stylet sur le support de travail.
3. Placez la caméra au-dessus du support de travail.
4. Placez le projecteur vidéo au-dessus du support de travail.

2.9.3 Contraintes

La procédure de mise en place n'est présentée ici qu'en guise de suggestion d'une disposition possible du matériel requis. Le but n'est pas d'imposer des contraintes matérielles et environnementales à l'utilisateur. Aussi, cette procédure peut effectivement être adaptée selon les besoins de l'utilisateur et/ou selon les possibilités que lui offre son environnement. L'important est de veiller à ce que le stylet demeure visible autant que possible dans le champ de vision de la caméra de manière à pouvoir le détecter et suivre ses mouvements de la manière la plus précise qui soit.

2.9.4 Exemple d'environnement

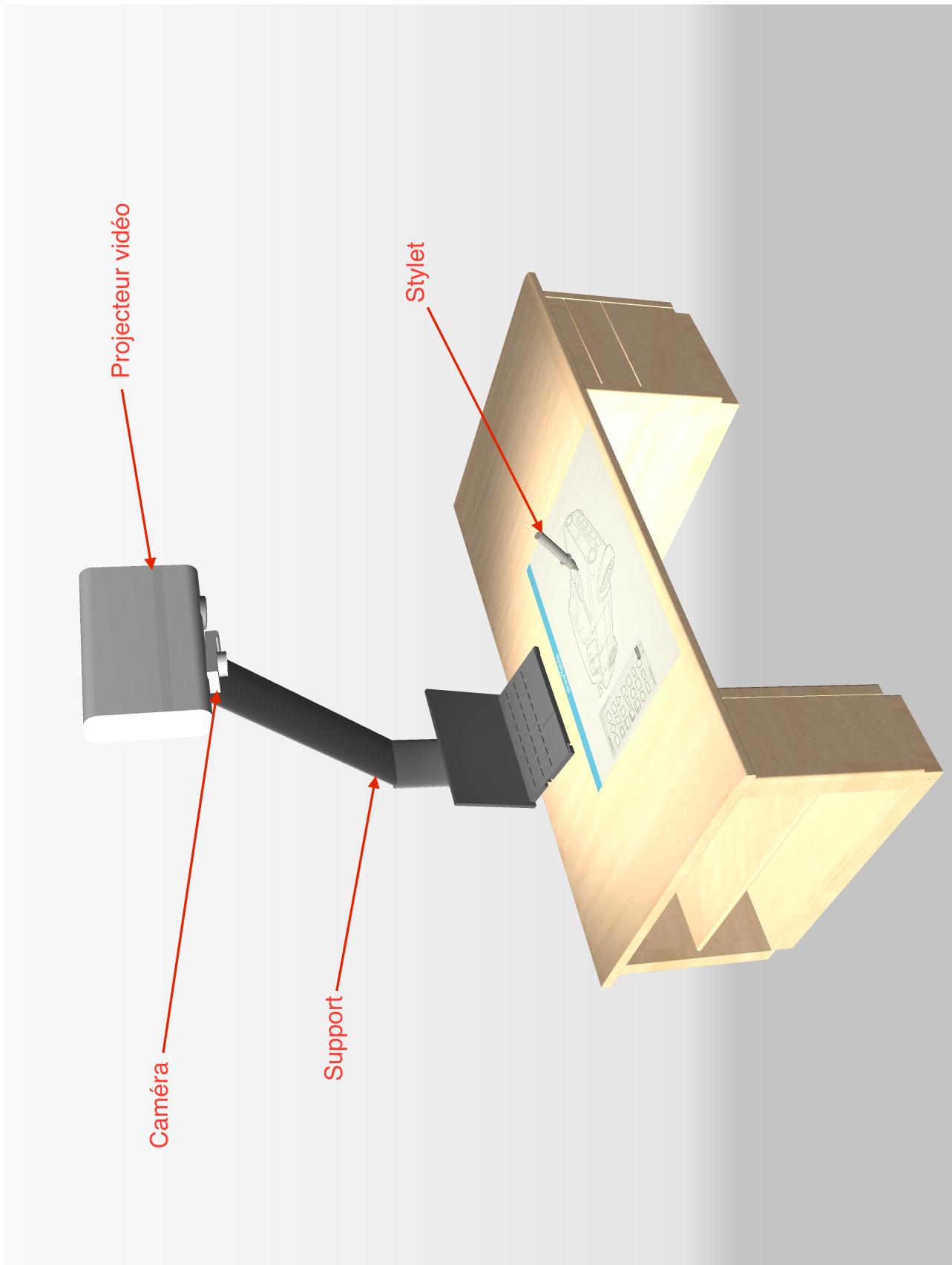


FIGURE 2.25 – Exemple d'environnement de dessin - vue en perspective

Tracking du stylet grâce à la caméra

- La caméra détecte la LED du stylet
- La LED s'allume lors d'une pression sur la mine.

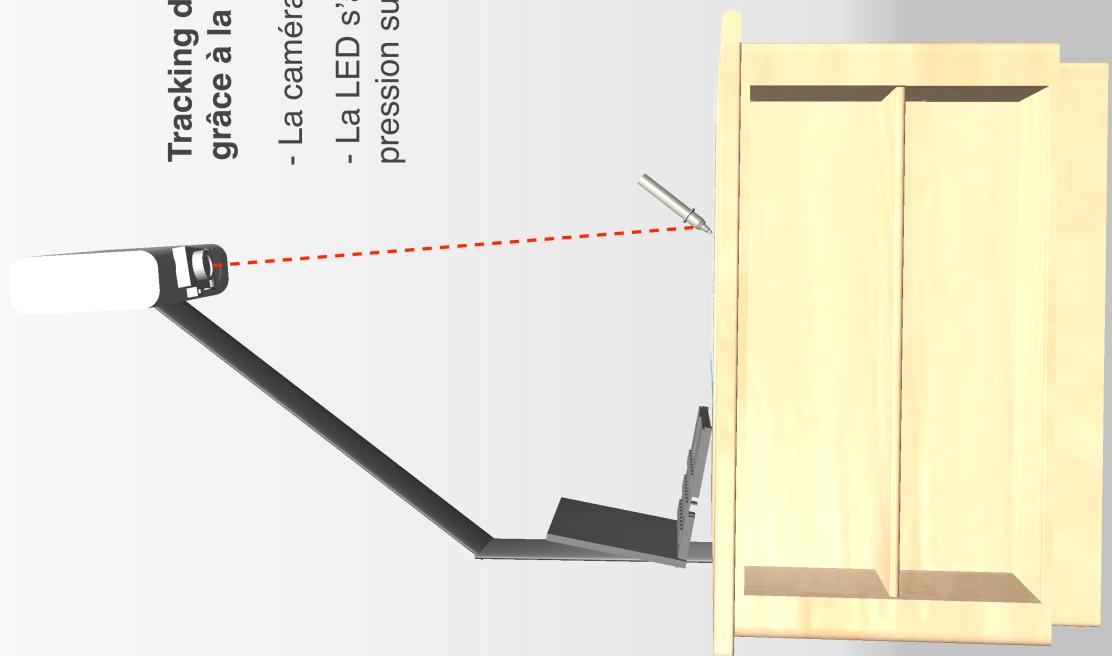


FIGURE 2.26 – Exemple d'environnement de dessin - vue de gauche

Chapitre 3

Description technique

Ce chapitre se veut être une description technique de l'implémentation de l'application. Cette description débute par l'introduction de la structure du projet. Celle-ci est relativement similaire à l'architecture introduite dans le chapitre correspond à la conception. Les patrons de conception mis en place durant l'implémentation sont mentionnés et font l'objet de brefs commentaires. Les librairies utilisées pour le développement de l'ensemble du projet sont rappelées ce qui conclut la première partie de ce chapitre.

La seconde partie du chapitre présente en premier lieu de l'interface graphique utilisateur de l'application. Il s'agit concrètement de décomposer et de décrire ses différentes composantes. Le complexe mécanisme du suivi du stylet est ensuite présenté de manière détaillée. Chacun des éléments formant l'ensemble de ce système de tracking est présenté en long et en large. Vient ensuite la description du dessin à proprement parlé et des différents outils mis à disposition pour son esquisse.

La dernière partie de ce chapitre technique présente l'application tiers qui fait office de simulateur de souris. Ce simulateur permet entre autres de prendre le contrôle de la souris de l'ordinateur et de reproduire les faits et gestes de l'utilisateur. En dernière instance, différentes fonctionnalités mineures de l'application sont introduites. Il s'agit entre autres des fonctionnalités de sérialisation et d'impression. Au préalable, le clavier virtuel de l'application est introduit au moyen de code partiel et de figures.

3.1 Structure du projet

La liste suivante se veut être une représentation de la structure du projet :

- LibreDraw
 - camera
 - CameraManager
 - CameraManagerUpdater
 - core
 - MainWindow
 - Menu
 - ToolBar
 - dialog
 - FileDialog
 - OpenFileDialog
 - SaveFileDialog
 - drawing
 - Drawing
 - DrawingController
 - error
 - ErrorManager
 - keyboard
 - Input
 - KeyboardButton
 - KeyboardModel
 - KeyButton
 - VirtualKeyboard
 - tool
 - DashController
 - EllipseController
 - EraserController
 - PenController
 - RectangleController
 - tracking
 - LedDetection
 - LedDetector
 - Controller
 - ScreenDetector
 - TrackingManager
 - Worker
 - Main

3.2 Patrons de conception

Trois patrons de conception ressortent clairement du projet. Il s'agit d'une part du *Singleton*. Celui-ci est utilisé pour différentes classes ne nécessitant qu'une seule instance et ce, durant toute l'exécution de l'application. Les classes faisant mettant en place ce patron sont les classes *ErrorManager*, *CameraManager* et les spécialisations de la classe abstraite *AbstractToolController*.

Le second patron de conception est le Modèle-vue-contrôleur. Celui-ci est mis en place pour l'implémentation des boîtes de dialogue devant fournir une interface permettant de naviguer à travers le système de fichier de la machine hôte. Il s'agit de cette navigation qui est caractérisée par ce patron de conception puisqu'il y a une séparation distincte de la vue avec le modèle.

Le troisième et dernier patron de conception est la Délégation. Un nombre important de classes de l'ensemble du projet délègue certaines tâches à d'autres classes. L'exemple le plus flagrant est la classe *DrawingController* qui lors de la réception d'un message devant reproduire une esquisse utilisateur à l'aide d'un outil donné. Cette classe délègue effectivement l'accomplissement de cette tâche à une des spécialisations de la classe abstraite *AbstractToolController*.

3.3 Librairies

Plus que des librairies, ce sont deux puissants frameworks qui ont été utilisé sur ce projet.

3.3.1 Qt

Qt est un framework multi-plateforme en C++ permettant le développement de grosse application graphique. Son plus grand avantage est de fournir une couche d'abstraction aux API graphiques de Windows, Mac OS X, GTK, etc. Ainsi, la création et la gestion des éléments graphiques de l'application se fait de manière uniforme au travers des méthodes fournies par Qt.

Afin de proposer des mécanismes plus haut niveau que ceux inclus dans C++, Qt met à disposition un pré-compilateur (qmake) qui transformera certaine partie du code propre à Qt (les Q_OBJECT, notamment) en C++. Il s'occupera aussi de la génération d'un Makefile compatible avec les différentes plateformes.

Dans ce projet, nous avons beaucoup utilisé Qt pour la gestion de fenêtre, mais aussi pour l'environnement de dessin. En effet, Qt met aussi à disposition des éléments comme la zone de dessin, la sélection et la gestion de couleur, etc.

3.3.2 OpenCV

OpenCV est un framework destiné au traitement en temps réel des images. Dans ce projet, nous l'avons surtout utilisé pour suivre le stylet de l'utilisateur et pouvoir ainsi reporter sa position dans le monde réel au référentiel de l'écran. Les algorithmes rapides d'OpenCV permettant de filtrer des couleurs, de travailler sur des formes géométriques et de faire des projections tridimensionnelles nous ont été très utiles.

3.4 Interface graphique utilisateur

Cette section présente l'interface graphique utilisateur de l'application. Cette présentation débute par une introduction de sa structure. Elle est suivie d'une décomposition de ses différents constituants. Chaque constituant fait l'objet d'une description individuelle tant sur sa forme que sur son contenu.

L'interface graphique utilisateur a été développée de manière à ce que l'interaction du stylet avec celle-ci soit la plus agréable, la plus naturelle et la plus ergonomique que possible. Compte tenu du fait que le stylet peut s'avérer quelque peu moins précis que la souris de d'un ordinateur et ce, quel que soit la qualité du système de tracking, les éléments cliquables de l'interface utilisateur doivent en conséquent être adaptés. Concrètement, la taille des éléments doit être ajustée, c'est-à-dire être plus grands qu'ils ne le seraient dans le cas d'une application desktop traditionnelle.

Les applications susceptibles d'être développées pour des dispositifs tels que des smartphones et/ou des tablettes font l'objet d'une telle problématique. En effet, les éléments graphiques susceptibles de réagir à une action utilisateur sont plus grands qu'usuellement de manière à ce que l'interaction avec le doigt d'une personne soit aisée. Cette remarque s'applique également à cette application de dessin avec pour seule différence que les interactions s'effectuent par le biais d'un stylet.

Les éléments de l'interface graphique utilisateur ont été développés selon la problématique exposée dans le paragraphe précédent. En conséquence, l'interface dispose d'éléments graphiques plus grands. Par ailleurs, ces éléments sont la plupart du temps caractérisés par des icônes plutôt que par du texte devant décrire l'action qu'ils sont supposés représenter, une image étant souvent bien plus explicite que des mots si elle est bien choisie.

3.4.1 Structure de l'interface utilisateur

La figure suivante présente l'interface graphique utilisateur finale de l'application :

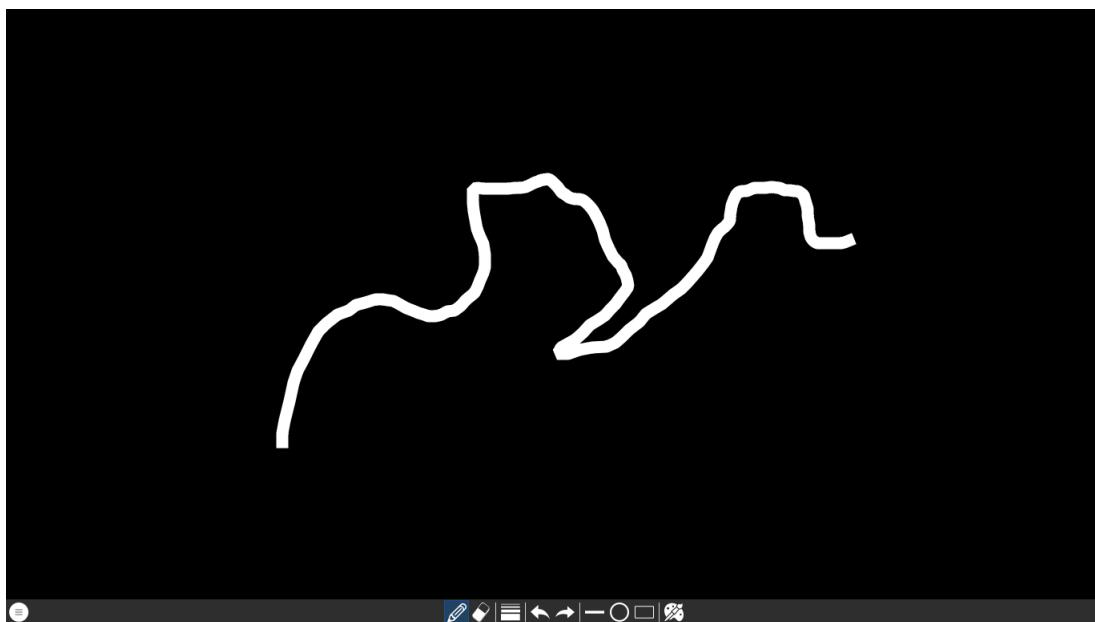


FIGURE 3.1 – Interface graphique utilisateur finale

La figure suivante présente la structure de l'interface graphique utilisateur finale de l'application :

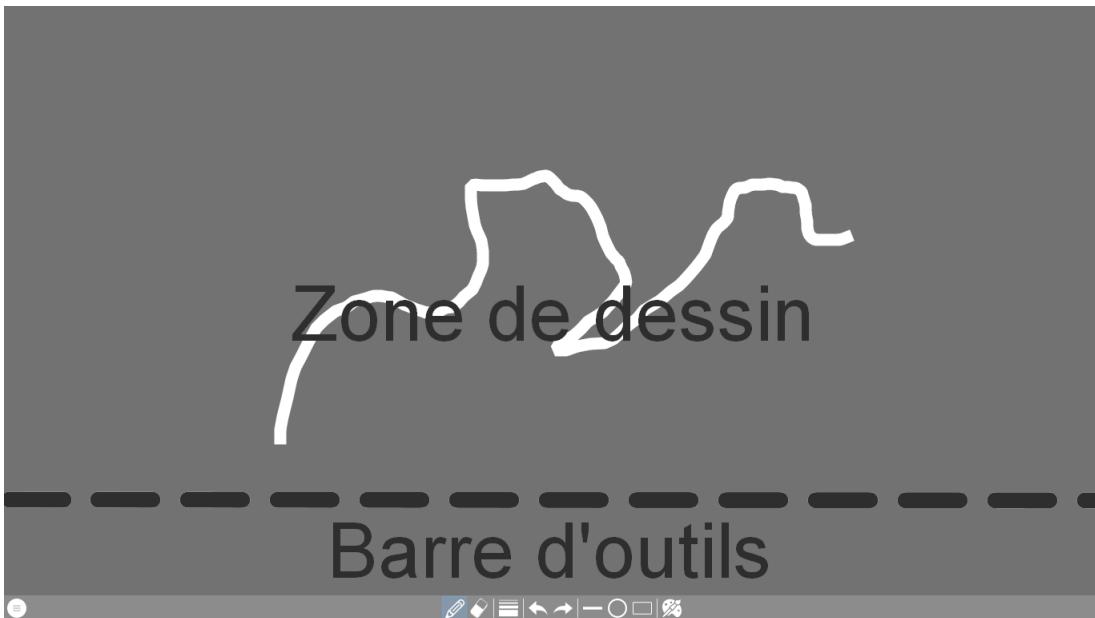


FIGURE 3.2 – Structure de l'interface graphique utilisateur finale

La structure de l'interface graphique utilisateur est relativement rudimentaire ; elle dispose d'une part d'une zone de dessin dans laquelle l'utilisateur est supposé y dessiner. D'autre part, elle dispose d'une barre d'outils laquelle fournit des outils de dessin, des outils de formes, des outils de couleurs et des outils d'épaisseur. Par ailleurs, cette barre d'outils met à disposition un outil supplémentaire permettant d'afficher le menu de l'application.

En termes d'implémentation, cette interface graphique utilisateur correspond à la classe `MainWindow`. Celle-ci hérite de la classe graphique `QMainWindow`. Cette classe fournie par le framework Qt propose entre autres une structure de base pour des applications caractérisées par une barre de menus, par une barre d'outils, par une zone centrale et par une éventuelle barre d'état.

Dans le cas de l'application de dessin, la barre de menus n'est pas conservée dans la mesure où un menu personnalisé a été développé pour des raisons ergonomiques et est mis à disposition de l'utilisateur au moyen d'une boîte de dialogue. La zone centrale et la barre d'outils y figurent en ce qui les concerne. Pour ce qui est de la barre d'état, elle n'a pas été ajoutée à l'interface étant donné que son utilité pour une telle application ne présente que peu d'intérêt.

Il peut être constaté que l'interface utilisateur n'est pas caractérisée par une barre de titre que l'on rencontre dans les programmes fenêtrés. En effet, dès le lancement de l'application est affichée en mode plein écran. Il s'agit-là du mode par défaut de l'application qui ne peut faire l'objet d'une modification. Cela se justifie par le fait que le système de tracking analyse la totalité de l'écran et que l'affichage d'une partie d'une fenêtre native du système d'exploitation pourrait dégrader la qualité de détection du stylet d'une part, le mécanisme de calibrage d'autre part.

En dernière instance, l'ébauche de l'interface graphique définie lors de l'établissement du cahier des charges a été adaptée selon les besoins ergonomiques de l'application de dessin.

3.4.2 Zone de dessin

La figure suivante illustre la zone de dessin de l'application de dessin :

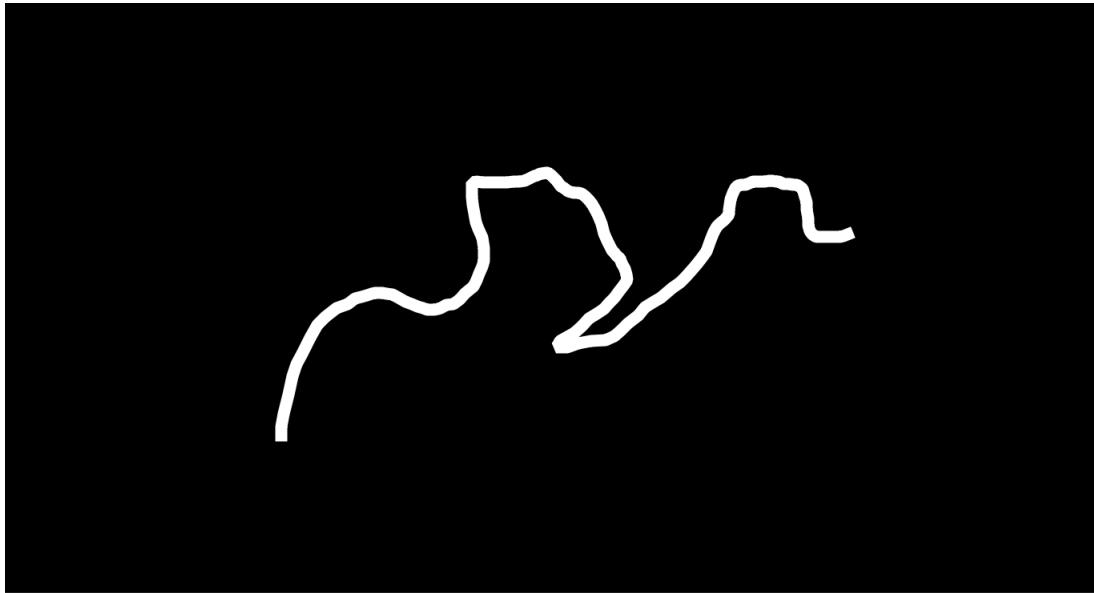


FIGURE 3.3 – Zone de dessin

La zone de dessin consiste bien évidemment en un espace ouvert dans lequel il est possible à tout un chacun d'y esquisser son dessin. Aussi, si ce n'est cet espace, aucun autre élément graphique n'y figure à l'exception des éventuelles esquisses de l'utilisateur. Dans le cas de la figure précédente, il s'agit simplement d'une esquisse réalisée à l'aide de l'outil crayon.

En termes d'implémentation, cette zone de dessin correspond à la classe Drawing. Il s'agit de la vue du dessin, c'est-à-dire de la représentation virtuelle du dessin esquissé sur le support physique de l'utilisateur. Cette classe fait l'objet d'une spécialisation de la classe graphique QGraphicsView mise à disposition par le framework Qt.

3.4.3 Barre d'outils

La figure suivante illustre la barre d'outils de l'application de dessin :



FIGURE 3.4 – Barre d'outils

La barre d'outils propose les outils suivants selon leur ordre d'apparition :

Outil	Icône	Action
Menu		Déclenche l'ouverture du menu de l'application.
Crayon		Déclenche la sélection de l'outil crayon.
Gomme		Déclenche la sélection de l'outil gomme.
Épaisseur		Déclenche l'ouverture d'un menu de sélection d'épaisseur.
Annuler		Annule la dernière action utilisateur sur le dessin.
Rétablissement		Rétablissement la dernière action utilisateur sur le dessin.
Trait		Déclenche la sélection de l'outil trait.
Cercle		Déclenche la sélection de l'outil cercle.
Rectangle		Déclenche la sélection de l'outil rectangle.
Couleur		Déclenche l'ouverture de la palette d'outils.

TABLE 3.1 – Outils de la barre d'outils de l'interface graphique utilisateur

La sélection d'un outil peut avoir pour conséquence la modification de l'icône du curseur selon l'outil donné. Ceci améliore l'expérience utilisateur dans la mesure où le curseur représente au mieux l'outil actuellement manipulé. Ainsi, la sélection de l'outil crayon aura pour conséquence l'affiche d'un curseur avec une icône représentant un crayon. En termes d'implémentation, la barre d'outils correspond directement à un objet de la classe *QToolBar* fournie par le framework Qt. Il n'a pas été jugé nécessaire d'en faire une spécialisation compte tenu du fait qu'elle proposait déjà toutes les fonctionnalités nécessaires à l'application. Par ailleurs, cette barre d'outils fait l'objet d'une personnalisation visuelle plutôt que de laisser le style de barre natif du système hôte. Ceci est réalisé par le CSS suivant :

Listing 3.1 – CSS de la barre d'outils de l'interface graphique utilisateur

```
1 QToolBar{  
2     background: rgb(46, 46, 46);  
3     border: 0px;  
4 }
```

3.4.4 Menu

En plus de la barre d'outils, le programme met à disposition de l'utilisateur un bouton de menu. Lors de l'appui sur ce dernier, un menu apparaît au centre de son écran lui proposant les options décrites ci dessous.

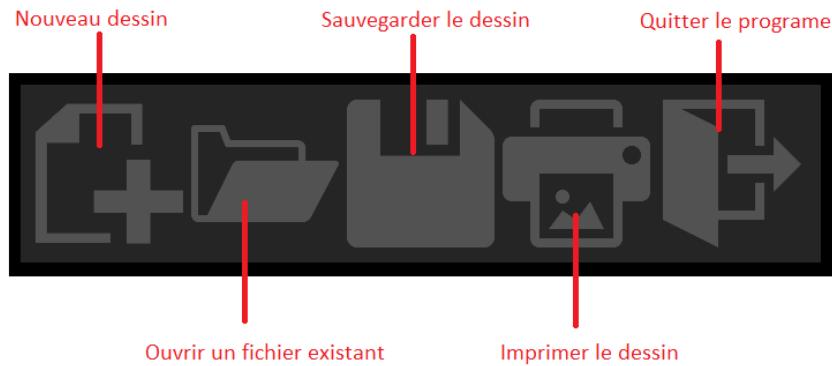


FIGURE 3.5 – Menu

3.4.4.1 Nouveau Fichier

Cette option permet de créer un nouveau dessin. En cas de modification du dessin courant avant la dernière sauvegarde, une fenêtre en informera l'utilisateur en lui proposant un des trois choix suivants :

1. Sauvegarder : ouvrira une nouvelle fenêtre pour sauvegarder le dessin. Une fois cela fait, un nouveau dessin vide apparaîtra sur le plan de travail.
2. Ne pas sauvegarder : le dessin actuel sera perdu et un dessin vide le remplacera.
3. Annuler : reviendra simplement au plan de travail sans modifier le dessin courant.

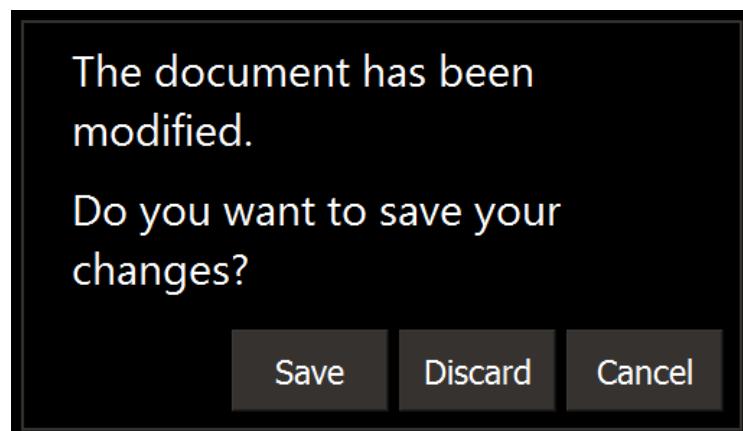


FIGURE 3.6 – Interface - Demande de sauvegarde

3.4.4.2 Ouvrir un fichier existant

Cette option ouvre un explorateur de fichier afin que l'utilisateur puisse importer dans le programme un fichier image existant. Il peut ensuite ensuite le modifier et l'enregistrer.

3.4.4.3 Sauvegarder

Cette option permet de sauvegarder les dernières modification amenées au dessin. Après avoir appuyé dessus, un explorateur de fichier ainsi qu'un clavier virtuel apparaîtront et l'utilisateur pourra définir l'emplacement ainsi que le nom sous lequel il souhaite l'enregistrer.

Le format de l'image par défaut est PNG, mais en ajoutant l'extension '.jpg' ou '.bmp', l'utilisateur peut utiliser les formats JPEG et Bitmap, respectivement.

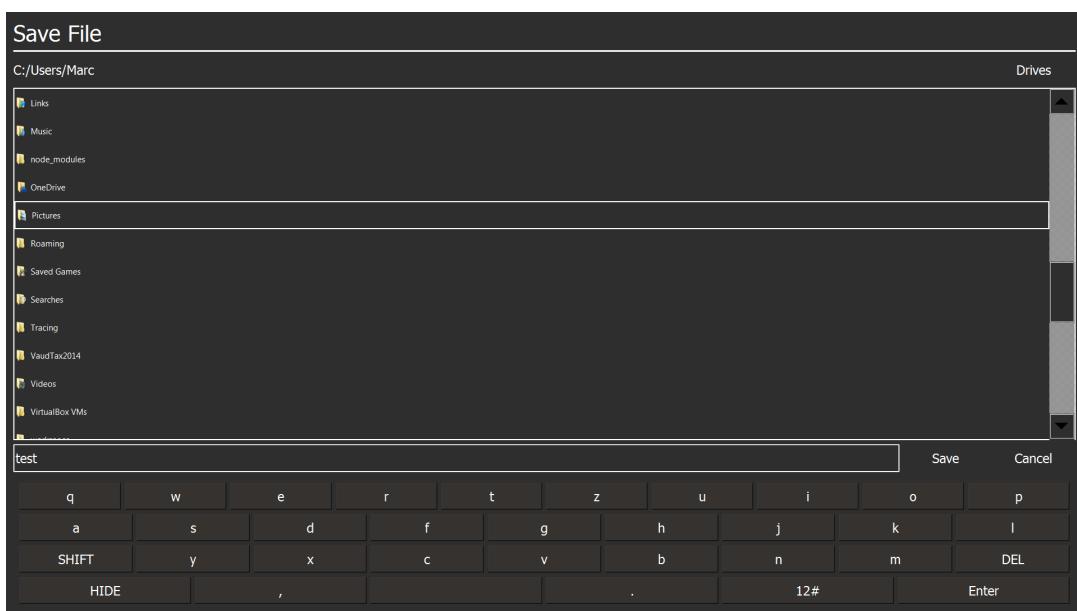


FIGURE 3.7 – Interface - Fenêtre de sauvegarde

3.4.4.4 Imprimer

Cette option permettra à l'utilisateur d'imprimer ses créations moyennant qu'une imprimante soit reliée à sa machine.

3.4.4.5 Quitter

Cette option permet à l'utilisateur de quitter le programme en toute sécurité, si une modification sur le dessin courant n'a pas été sauvegardée, la même fenêtre que pour la création d'un nouveau fichier s'ouvrira afin de proposer à l'utilisateur de sauver les modifications avant de quitter complètement le programme.

3.4.5 Ressources

Notre interface graphique utilise de nombreuses icônes personnelles, afin de pouvoir les utiliser au sein de notre programme nous avons dû créer un fichier de configuration liant ces différentes icônes au logiciel.

3.4.5.1 Fichier source

Contenu du fichier de configuration associant chaque icône à un chemin reconnu par le logiciel afin de pouvoir les utiliser.

```
1 <RCC>
2     <qresource prefix="/menu">
3         <file>icons/new_on.png</file>
4         <file>icons/open_on.png</file>
5         <file>icons/print_on.png</file>
6         <file>icons/quit_on.png</file>
7         <file>icons/save_on.png</file>
8         <file>icons/new_off.png</file>
9         <file>icons/open_off.png</file>
10        <file>icons/print_off.png</file>
11        <file>icons/quit_off.png</file>
12        <file>icons/save_off.png</file>
13    </qresource>
14    <qresource prefix="/tool">
15        <file>icons/color.png</file>
16        <file>icons/dash.png</file>
17        <file>icons/ellipse.png</file>
18        <file>icons/eraser.png</file>
19        <file>icons/menu.png</file>
20        <file>icons	mode.png</file>
21        <file>icons/pen.png</file>
22        <file>icons/rectangle.png</file>
23        <file>icons/thickness.png</file>
24        <file>icons/thickness1.png</file>
25        <file>icons/thickness2.png</file>
26        <file>icons/thickness3.png</file>
27        <file>icons/thickness4.png</file>
28        <file>icons/thickness5.png</file>
29        <file>icons/thickness6.png</file>
30        <file>icons/redo.png</file>
31        <file>icons/undo.png</file>
32    </qresource>
33    <qresource prefix="/cursor">
```

```
34      <file>icons/eraser.ico</file>
35      <file>icons/pen.ico</file>
36    </qresource>
37  </RCC>
```

3.4.5.2 Exemple d'utilisation

Dans l'exemple suivant nous créons le bouton menu de notre *Toolbar* et nous souhaitons lui assigner une icône personnelle (2ème ligne). Grâce au fichier de configuration présenté plus haut le chemin vers nos icônes personnelles est reconnu par la classe *QIcon*.

```
1 menu = new QAction(this);
2 menu->setIcon(QIcon(":/tool/icons/menu.png"));
3 connect(menu, SIGNAL(triggered()), this, SLOT(onMenuTriggered()));
```

3.5 Tracking

Cette partie du projet à été un réel défi car le traitement d'image était tout un sujet peu connu mais qui demande en plus de bonne notions de programmation en C++ et des mathématiques. Nous avons pu à présent affirmer que la qualité d'un logiciel de traitement d'image ou de vision par ordinateur dépend fortement de ces dernières notions.

En concevant un tel logiciel il y a 3 principaux problèmes qui à prendre comptes, qui n'ont :

- **La rapidité :** En effet, le fait d'utiliser un algorithme ou l'autre pour effectuer une certaine tâches peut avoir d'énorme répercussion sur la rapidité et la fluidité du logiciel. Sachant qu'il est question de traiter un certain nombre d'images par secondes, quelques millisecondes de moins ou de plus se remarques immédiatement à l'utilisation du logiciel. Ce problème a été traité en choisissant le C++ comme langage de programmation, OpenCV comme librairies pour manipuler des images et en se renseignant toujours sur l'utilisation et les limites des fonctions offertes par cette librairie.
- **Les variations de lumières ambiante :** Ici il s'agit de créer un logiciel pouvant fonctionner de la manière attendue indépendamment de l'environnement dans le quel il est. Ce problème a été résolu en utilisant l'espace de couleur HSV (Hue, Saturation et Value) pour représenter et pour manipuler les images reçues par la caméra.
- **Précision et fiabilité :** Ce dernier problème n'étant pas des moindres, est lié à la qualité d'images de la caméra et aux algorithmes utilisé pour les traiter. Cependant la qualité de la caméra est le seul facteur pouvant vraiment nous empêcher d'atteindre notre but. En effet, il existe plusieurs moyen d'écrire un code plus efficace ou plus fiable mais quand nous atteignons les limites de la qualité de l'image, il devient impossible de faire mieux.

Le projet a commencé par une phase d'étude des différentes manière de représenter des images, les différents espaces de couleurs et comment tirer certaines informations d'une image. C'est pourquoi nous commenceront par fixer les principes de bases utilisé dans le cadre de ce projet.

Ensuite, nous détaillerons nos deux algorithmes de détection d'écran et de Tracking d'une Led.

3.5.1 Introduction à la vision par ordinateur

3.5.1.1 Stockage et manipulation d'image

Une image numérique est une matrice de pixel de taille $w * h$ dans laquelle chaque cases représente l'intensité de la couleur d'un pixel à une paire de coordonée (x,y) . Pour une image en niveau de gris, chaque pixel est stocké en une valeur comprise dans l'intervalle $[0,255]$. Plus l'intensité est forte, plus la valeur est grande. Ce principe s'applique aussi au images couleur. Il n'y aura cependant plus une unique matrice mais plusieurs afin stocker plusieurs canaux. Ainsi, chaque pixel aura plusieurs valeurs associées à chacun des canaux.

La librairie OpenCV nous fournit une interface pour lire une image, accéder à ses pixels et effectuer divers manipulation courante comme : redimensionnement, conversion, copie etc ...

3.5.1.2 Formats d'images

Profondeur

Une image numérique est tout d'abord caractérisée par sa profondeur. La profondeur est lié à la taille mémoire d'un pixel. Vous avez certainement entendu parlé d'images 8, 24, 32 bits. Nous avons dit précédemment que la valeur d'un pixel était compris dans l'intervalle [0,255]. Et bien, cela ne s'applique uniquement aux images de profondeurs 8 bits. La librairie OpenCV supporte différents types de profondeur en permettant d'exprimer les valeurs sur 8, 24, 32 et 64 bits signé ou non signé. Nous avons travaillé de manière exclusive sur des intensités comprises entre 0 et 255 en utilisant le type CV_8U, le 8 pour 8 bits et le U étant pour "unsigned".

Couleur

En physique, la couleur étant une onde est porteuse de plus d'informations due à sa fréquence et son amplitude. On choisit donc le format d'une image en fonction des informations que l'on veut en tirer.

Comme dit plus tôt, une image en niveau de gris représente une intensité lumineuse pour chaque pixel. Nous avons donc accès à une seule information appelée luminance qui est caractérisée par l'amplitude de l'onde lumineuse qui vient frapper un certain point.

La fréquence est porteuse d'une seconde information, la couleur, appelée chrominance. Il existe différentes manières de coder la couleur d'un pixel sur trois nombres et c'est ici qu'entre en jeu les différents espaces de couleur.

Nous utilisons deux différents espaces de couleur dans notre logiciel que nous allons brièvement décrire.

RGB : C'est la manière la plus courante de représenter la couleur d'un pixel car il est possible de représenter toutes les couleurs en superposant trois composantes : le rouge, le vert et le bleu. Une image RGB est représentée par une matrice à trois canaux. Le type OpenCV correspondant est CV_8UC3 où C3 correspond à 3 canaux.

Dans notre logiciel de tracking, nous utilisons cet espace de couleur pour récupérer les images de la caméra. C'est le type par défaut utilisé pour la lecture d'un flux d'images provenant d'une caméra. Nous ne pouvons pas tirer énormément d'informations sur une image avec un tel espace de couleur. Les couleurs étant un simple dosage de rouge, vert et bleu, il est très difficile d'isoler une couleur précise en acceptant toutefois un intervalle de différences car la couleur et la luminosité sont entièrement mixées entre les 3 valeurs.

HSV : Cette espace de couleur est composé de 3 canaux : Hue, Saturation et Value. C'est une représentation cylindrique de l'ensemble des couleurs.

La Teinte (Hue) est exprimée par une valeur en degré. Celle-ci représente la couleur d'un pixel. Habituellement, la teinte s'étend sur l'intervalle [0 : 360]. OpenCV a choisi par défaut d'utiliser un intervalle plus restreint [0 : 180], certainement pour des raisons de performances.

La Saturation est l'intensité de la couleur ou autrement dit à quel proportion est-elle mixée avec du blanc.

La Valeur correspond à la brillance de la couleur ou autrement dit à quel proportion elle est mixée avec du noir.

La Saturation est la valeur peuvent varier selon la lumière ambiante, tandis que la teinte est entièrement indépendante. C'est la raison pour laquelle cette espace de couleur est souvent utilisé dans le domaine de la vision par ordinateur, lorsqu'il s'agit de détecter des objets ayant une certaine couleur.

3.5.2 Système de détection

En utilisant l'espace de couleur HSV, nous sommes capable de détecter et de suivre n'importe quel objet en fonction de sa couleur. Pour autant que les images reçues de la caméra soit suffisamment de bonne qualité, c'est à dire pas trop saturée, si notre oeil peut percevoir un objet sur une image, il est possible de créer un programme pouvant en faire autant.

On faisant des recherches dans le domaine du tracking, nous avons pu constater un très grand nombre de différentes approches ou d'algorithmes dédié à cela. Heureusement, la librairie OpenCV à tout d'abord été créée pour ne pas avoir à réécrire des algorithme souvent utilisé et parfois très complexes. Elle réunis le résultat de plusieurs années de recherche dans le domaine de la vision par ordinateur et nous à permis de tester et comparer plusieurs approches.

3.5.2.1 Détection d'écran

Approche implémentée Dans notre logiciel, l'écran projeté par un beamer est caractérisé par plusieurs informations (features) que nous devons décrire à l'ordinateur afin qu'il puisse le détecter. Si le but était de détecter n'importe quel type d'écran quelque soit sa taille et ce qu'il afficher, cela reviendrait un problème très complexe. Comme nous avons une certaine maîtrise sur le logiciel de dessin, il nous est possible d'imposer des conditions au moment du calibrage afin d'apporter plus d'informations au logiciel lorsqu'il analysera l'image. Ces conditions sont : L'écran possède une couleur précise et il est objet qui prend le plus d'espace sur l'image. Ainsi, la première tâche de la classe *ScreenDetector* consiste à détecter le plus grand rectangle vert visible sur une image.

La première étape après avoir récupéré l'image RGB contenant potentiellement un écran vert, est de la convertir dans l'espace de couleur HSV afin de pouvoir isoler la couleur verte située dans l'intervalle HUE[41 :90]. Selon la luminosité de la pièce, la brillance de l'écran illuminant les objets proches leur donne à eux aussi une couleur verte. Si l'on applique un filtre sur une telle image en prenant uniquement en compte la composante HUE, l'écran détecté ne se limitera pas à son contour mais aussi jusqu'à tout objet illuminé par cette écran. Dans une pièce très sombre, l'image entière serait considérée comme un écran.

C'est pourquoi nous avons soigneusement choisi l'intervalle des composantes de saturation et de valeurs en effectuant plusieurs tests. En choisissant une saturation S minimum fixée à 20%, et une Valeur V minimum de 37%, le logiciel est capable d'isoler le vert de l'écran en supprimant les pixels vert causé par la lumière de l'écran. Lorsque nous appliquons un filtre sur une image, nous obtenant en résultat une

matrice M (une seul canal), contenant soit des pixels blancs où $M(x,y) = 255$ soit des pixels noirs où $M(x,y) = 0$.

Cependant il est possible que plusieurs zones blanche (acceptée) apparaissent lors de l'application du filtre. Une première chose à faire est de supprimer les tout petits éléments étant considéré comme des bruits. Pour cela nous utilisons les méthodes `cv : :erode` et `v : :dilate` sur cette matrice. De plus, il est possible d'avoir d'autres zones blanches, trop grande pour être considéré comme bruit. Par exemple un pot de fleur vert ou une personne habillée en vert. Pour résoudre ce problème nous utilisons l'une des conditions fixée préalablement : *déetecter le plus grand rectangle vert visible*. Pour cela nous allons devoir comparer l'aire des masses blanches et garder uniquement la plus grande.

Pour trouver l'aire d'une masse blanche nous devons la contourer puis calculer son aire. Nous utilisons la méthode `cv : :findContours` qui retourne un vecteur de tous les contours trouvé dans l'image, puis enfin `cv : :contourArea(Contour)` qui retourne l'aire comprise à l'intérieur d'un contour. Ces deux méthodes nous permettent de trier nos contours par surfaces et ainsi récupérer le plus grand contour.

Une fois que nous avons obtenu le plus grand contour, il est nécessaire d'approximer ce contour afin de limiter le nombre de points qu'il contient. `cv : :approxPolyDP` nous permet de trouver d'approximer un contour de manière à l'exprimer en un nombre minimum de points. Nous vérifiant que cette forme possède 4 points afin d'être sûr que c'est le rectangle de l'écran qui a été détecté.

Finalement, le point clé de cette étape de calibration est la création d'une matrice de transformation permettant de convertir un point dans le plan vue de la caméra en un point sur l'écran de l'utilisateur. Comme nous avons eu un rectangle *source* déformé grâce à l'étape précédente, nous allons créer un autre rectangle *destination* de la même taille que l'écran de l'utilisateur puis créer une matrice de transformation 3D obtenue par la fonction `cv : :getPerspectiveTransform(source, destination)`. Ainsi nous pouvons convertir un point *p1* dans le plan de la source en un point *p1'* en passant la matrice de transformation et ce point à la fonction `cv : :perspectiveTransform`.

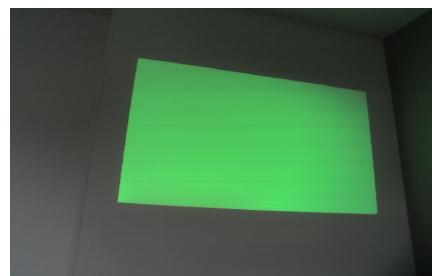


FIGURE 3.8 – Image source de la caméra



FIGURE 3.9 – Filtre HSV

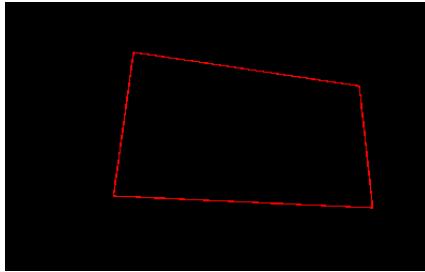


FIGURE 3.10 – Rectangle approximés de l'écran

3.6 Dessin

Comme expliqué précédemment, la structure logique modélisant un dessin est une structure de type *bitmap*, il s’agit d’une matrice de pixel ayant leur propre couleur. Le tout formant ainsi le dessin créé.

Afin de mettre en place cette modélisation, nous avons utilisés les propriétés de la classe *QGraphicsScene*.

Cette dernière représente une surface 2D pouvant contenir des objets graphiques. A savoir des *QGraphicsItem*.

Dans le cadre de notre projet, elle fait office de matrice de pixels et s’occupe de gérer les modifications de ces derniers lors d’ajout d’objets graphiques.

L’avantage de la classe *QGraphicsScene* est qu’elle gère les objets graphiques à la manière d’une superposition de calque. Il est donc aisément possible de modifier les objets graphiques insérés, de les retirer ou de les modifier.

Une *QGraphicsScene* seule ne suffit pas, cette classe n’est qu’un manager d’objets graphiques, il lui faut un support pouvant gérer graphiquement l’affichage du contenu d’une scène. Pour ce faire nous utilisons la classe *QGraphicsView* ou devrait-on dire la classe personnalisée *Drawing* héritant des propriétés requises de *QGraphicsView* tout en nous permettant de lui assigner un contrôleur (*DrawingController*) gérant la transition entre les caractéristiques du trait ainsi que les différents outils mis à disposition de l’utilisateur par le logiciel en plus de gérer l’historique des actions effectuées.

Dans les sous-chapitres qui vont suivre, une action fait référence à l’interaction de l’utilisateur avec le programme de dessin aboutissant sur la modification de la scène et donc du dessin.

Notre programme contient différents outils permettant à l'utilisateur d'effectuer des actions. Nous allons voir la liste complète des outils et expliquer l'implémentation de ces derniers. Mais avant ça, nous allons voir comment est implémentée la gestion permettant de choisir un outil et la transition entre ces derniers.

3.6.1 Transition entre outils

Comme nous pouvons le voir dans la figure qui suit, la *Toolbar* mise à disposition par le logiciel comporte différents outils sélectionnable.



FIGURE 3.11 – Sélection d'un outil de la barre d'outils

Chaque élément de cette *Toolbar* est créé et ajouté à cette dernière lors de l'initialisation de la classe *ToolBar* dans la classe *MainWindow*, classe gérant la fenêtre principale de l'application. Pour expliquer l'implémentation de la transition entre les outils, nous allons suivre le fonctionnement d'un seul des boutons de la *Toolbar*. Le fonctionnement et l'implémentation des autres boutons étant identiques.

Prenons le bouton permettant la sélection de l'outil ellipse. Lors de la création de la *Toolbar* le bouton ellipse est créé à l'aide du code suivant :

Listing 3.2 – Exemple de création d'une action de la barre d'outils

```

1  ellipse = new QAction(this);
2  ellipse->setCheckable(true);
3  ellipse->setIcon(QIcon(":/tool/icons/ellipse.png"));
4  connect(ellipse, SIGNAL(triggered(bool)), this,
5  SLOT(onEllipseTriggered(bool)));

```

Et ajouter à la toolbar à l'aide de la ligne suivante :

```
1  addAction(ellipse);
```

Il est tout d'abord à noter que ce que l'on pensait être un bouton est en réalité une *QAction*, une classe permettant de représenter une action au sein d'une interface utilisateur.

Nous utilisons le mécanisme de signal / slot afin d'associer une interaction de notre choix lorsque le signal ‘*triggered()*’ de l'action est levé.

Dans notre cas lorsque l'on clique sur le bouton ‘ellipse’ la *QAction* *ellipse* émet le signal ‘*triggered()*’ ce qui a pour effet d'activer le slot ‘*onEllipseTriggered(bool)*’ de notre classe *Toolbar*.

Si l'on se penche sur le code de ce slot :

```

1 void ToolBar::onEllipseTriggered(bool checked) {
2     MainWindow* window = qobject_cast<MainWindow*>(parent());
3     if (!window) { return; }

```

```

4     window->onEllipseTriggered (checked) ;
5 }
```

Nous pouvons observer que lorsque le slot est activé, il appelle la méthode de même nom de la classe *MainWindow* en lui fournissant comme paramètre un boolean correspondant au statut de l'action à savoir est-ce qu'il est déjà sélectionné ou non.

```

1 void MainWindow::onEllipseTriggered(bool checked) {
2     if (checked) { controller->setDrawController(
3         EllipseController::getInstance()); }
4     else { toolBar->getEllipse()->setChecked(true); }
5     drawing->setCursor(Qt::CrossCursor);
6 }
```

La méthode appelée s'occupe de modifier l'outil sélectionné de la toolbar si l'outil voulu n'était pas déjà sélectionné ce qui a pour effet d'émettre à nouveau le signal *triggered()* de la *QAction*, ce qui impliquera un nouveau passage dans cette méthode avec cette fois le boolean *checked* à true.

Si le paramètre est true, la méthode s'occupe de récupérer le contrôleur, dans notre cas *EllipseController*, gérant la création d'ellipse et l'assigne comme étant le controller actif à l'aide de la méthode *setDrawController()* de notre contrôleur de dessin *DrawingController*.

```

1 void DrawingController::setDrawController(AbstractToolController
2 * drawController) {
3     this->drawController = drawController;
4 }
```

Cette dernière s'occupe simplement de remplacer le contrôleur actif.

Voilà comment est implementée la transition entre les différents outils. Chaque fois que l'on sélectionne un nouvel outil, le contrôleur gérant le dessin est remplacé par le contrôleur gérant l'outil courant.

3.6.2 Outils de dessin

Les outils de dessin regroupent deux outils différents à savoir le crayon et la gomme. La particularité de ces derniers étant qu'ils dessinent/effacent pixels par pixels en fonction du déplacement du stylet. Ils ne suivent pas une forme géométrique précise et préconçue autre que les mouvements de poignet de l'artiste.

3.6.2.1 Crayon

L'outil crayon permet de dessiner librement les traits et formes voulues. Il modifie la scène pixel après pixel en fonction des déplacements du stylet.

L'implémentation de cet outil s'approche grandement de cette description. En effet si l'on observe le code du contrôleur gérant l'outil crayon nous pouvons observer ceci :

```
1 void PenController::mouseMoveEvent (QGraphicsScene *scene,
```

```

2 QMouseEvent *event) {
3     Q_UNUSED(scene)
4     QPointF point = event->pos();
5     path->lineTo(point);
6     pathItem->setPath(*path);
7 }
8 void PenController::mousePressEvent (QGraphicsScene *scene,
9 QMouseEvent *event, QPen* pen) {
10    path = new QPainterPath(event->pos());
11    pathItem = scene->addPath(*path, *pen);
12 }
13 QGraphicsItem* PenController::mouseReleaseEvent (QGraphicsScene *scene,
14 QMouseEvent *event) {
15     Q_UNUSED(scene)
16     Q_UNUSED(event)
17     return pathItem;
18 }
```

Il est à noter que l'on utilise les événements de la souris. En effet nous avons fait en sorte que les mouvements et actions du stylet soient considérés comme étant des mouvements et actions de la souris. Ceci est vrai pour tous les autres contrôleurs et ne sera plus souligné.

Pour en revenir sur l'implémentation de cet outil on peut observer que lorsque le stylet émet l'événement comme quoi il est appuyé (ce qui correspond à la LED allumée) nous créons un objet de type *QPainterPath* en lui fournissant la position (x,y) actuelle du stylet correspondant à la position de départ de notre figure.

Cet objet nous permet de chainer une succession de point afin de créer un chemin les traversant. Comme nous pouvons le voir dans la méthode *mouseMoveEvent()*, chaque fois que le stylet est activé et bouge, nous récupérons la nouvelle position que nous fournissons à notre *QPainterPath* qui lui s'occupe de relier la dernière position qu'il contient avec la nouvelle position puis l'ajoute à son chemin.

Tout comme la *QGraphicsScene* ne s'occupe que de manager des objets graphiques, il en va de même pour notre *QPainterPath* qui ne s'occupe que de générer un chemin entre différents points.

C'est pourquoi nous utilisons en parallèle un *QGraphicsPathItem*, un objet gérant notre *QPainterPath* et le rendant compréhensible pour notre *QGraphicsScene* qui à son tour permet son affichage dans la *QGraphicsView*.

C'est pourquoi chaque fois que notre *QPainterPath* est créé ou modifié, nous l'ajoutons à notre *QGraphicsPathItem* afin que le trait apparaisse de manière dynamique à l'écran.

Lorsque le stylet se désactive (la LED s'éteint), cela signifie que l'utilisateur a terminé son trait. Ce dernier est alors retourné à notre *DrawingController* qui s'occupe alors de conserver la référence sur l'item créé afin de l'ajouter à l'historique des actions effectuées.

Le code correspondant se trouve ci-dessous :

```
1 void DrawingController::mouseReleaseEvent(QMouseEvent* event) {  
2     if (isEnabled) {  
3         lastActions.append(drawController->mouseReleaseEvent(  
4             drawing->scene(), event));  
5         nextActions.clear();  
6         modifToSave = true;  
7     }  
8 }
```

Ceci est vital pour nous permettre d'annuler les actions effectuées mais nous reviendrons là-dessus plus tard.

3.6.2.2 Gomme

L'implémentation de notre gomme est identique à notre implémentation du crayon. En effet nous avons pris le parti de ne pas effacer les modifications de couleurs sur les pixels qui croisent le chemin de notre gomme mais plutôt de les repeindre avec la couleur de fond.

```
1 void EraserController::mouseMoveEvent (QGraphicsScene *scene,
2 QMouseEvent *event) {
3     Q_UNUSED(scene)
4     QPointF point = event->pos();
5     path->lineTo(point);
6     pathItem->setPath(*path);
7 }
8
9 void EraserController::mousePressEvent (QGraphicsScene *scene,
10 QMouseEvent *event, QPen* pen) {
11     Q_UNUSED(pen);
12     path = new QPainterPath(event->pos());
13     pathItem = scene->addPath(*path, QPen(QBrush(Qt::black), pen->width()));
14 }
```

Comme on peut le constater le code est identique à l'exception de la ligne suivante :

```
1 pathItem = scene->addPath(*path, QPen(QBrush(Qt::black), pen->width()));
```

Où l'on précise lorsque l'on ajoute l'item à la scène que la couleur à utiliser est le noir, couleur correspondant à la couleur de fond au lieu d'utiliser la couleur courante stockée dans notre *DrawingController*.

3.6.3 Outils de formes

Les outils de formes regroupent les outils utilisant des formes précises tel que les rectangles, les ellipses ou encore les segments.

3.6.3.1 Segment

Cet outil fournit la possibilité de dessiner un segment entre le point de départ et le point d'arriver. Bien entendu, tant que le stylet est actif, le segment doit être modifié dynamiquement en fonction des déplacements du stylet.

Pour ce faire lorsque l'on active le stylet, nous utilisons la méthode *addLine()* de notre scène nous permettant de créer un objet *QGraphicsLineItem*. Un objet permettant de relier deux points par un segment.

Nous définissons comme point de départ ainsi que d'arrivée la position initiale du stylet et étant un objet créé par l'appel d'une méthode de la scène, ce dernier est directement intégré à cette dernière comme nous pouvons le voir dans le code suivant :

```

1 void DashController::mousePressEvent (QGraphicsScene *scene,
2 QMouseEvent *event, QPen* pen) {
3     item = scene->addLine (event->x(), event->y(), event->x(), event->y(), *pen);
4 }
```

Lorsque l'on bouge le stylet, il nous suffit de modifier la position du point d'arrivée de l'objet *QGraphicsLineItem* précédemment créé pour que la modification visuelle soit dynamique.

```

1 void DashController::mouseMoveEvent (QGraphicsScene *scene,
2 QMouseEvent *event) {
3     Q_UNUSED (scene)
4     if (item) { item->setLine (item->line ().x1(),
5         item->line ().y1(), event->x(), event->y()); }
6 }
```

Tout comme pour le crayon, lorsque le stylet est désactivé. Nous récupérons la référence sur l'item afin de l'ajouter à l'historique des actions de notre *DrawingController*.

```

1 QGraphicsItem* DashController::mouseReleaseEvent (
2 QGraphicsScene *scene, QMouseEvent *event) {
3     Q_UNUSED (scene)
4     Q_UNUSED (event)
5     return item;
6 }
```

3.6.3.2 Rectangle

Cet outil fournit la possibilité de dessiner un rectangle à partir d'un point de départ, une hauteur et une largeur. Bien entendu, tant que le stylet est actif, le rectangle doit être modifié dynamiquement en fonction des déplacements du stylet.

Pour ce faire nous avons eu recours à la méthode *addRect()* de notre scène qui nous retourne un objet de la classe *QGraphicsRectItem*. Nous définissons comme position de départ la position initiale du stylet ainsi qu'une hauteur et une largeur nulle et étant un objet créé par l'appel d'une méthode de la scène, ce dernier est directement intégré à cette dernière comme nous pouvons le voir dans le code suivant :

```

1 void RectangleController::mousePressEvent (
2 QGraphicsScene *scene, QMouseEvent *event, QPen* pen) {
3     origin.setX (event->x());
4     origin.setY (event->y());
5     item = scene->addRect (event->x(), event->y(), 0, 0, *pen);
6 }
```

Il est à noté que nous conservons les positions de l'origine du rectangle car de par les spécificités de la classe *QGraphicsRectItem* il nous était impossible de dessiner un rectangle en lui fournissant des valeurs négatives que ce soit pour la hauteur ou la largeur.

Du coup, nous avons dû adapter notre code afin de modifier l'origine du rectangle dessiné palier à cette restriction de la manière suivante :

```
1 void RectangleController::mouseMoveEvent(
2     QGraphicsScene *scene, QMouseEvent *event) {
3     Q_UNUSED(scene)
4     if (item) {
5         int x = origin.x() > event->x() ? event->x() : origin.x();
6         int y = origin.y() > event->y() ? event->y() : origin.y();
7         int w = abs(origin.x() - event->x());
8         int h = abs(origin.y() - event->y());
9         item->setRect(x, y, w, h);
10    }
11 }
```

Comme vous pouvez le constater, lors de chaque déplacement du stylet activé nous récupérons la position courante de ce dernier. On définit comme nouvelle origine du rectangle la position la plus éloignée entre l'origine initiale et la nouvelle position puis on récupère la hauteur et la largeur du nouveau rectangle avant de re-dessiner le nouveau rectangle.

Comme toujours, une fois l'action terminée, nous retournons la référence vers l'item créé pour le stocker dans l'historique des actions comme vous pouvez le voir dans le code qui suit :

```
1 QGraphicsItem* RectangleController::mouseReleaseEvent (
2     QGraphicsScene *scene, QMouseEvent *event) {
3     Q_UNUSED(scene)
4     Q_UNUSED(event)
5     return item;
6 }
```

Il est à noté que nous conservons les positions de l'origine du rectangle car de par les spécificités de la classe *QGraphicsRectItem* il nous était impossible de dessiner un rectangle en lui fournissant des valeurs négatives que ce soit pour la hauteur ou la largeur.

Du coup, nous avons dû adapter notre code afin de modifier l'origine du rectangle dessiné palier à cette restriction de la manière suivante :

```
1 void RectangleController::mouseMoveEvent (
2     QGraphicsScene *scene, QMouseEvent *event) {
3     Q_UNUSED(scene)
4     if (item) {
5         int x = origin.x() > event->x() ? event->x() : origin.x();
6         int y = origin.y() > event->y() ? event->y() : origin.y();
7         int w = abs(origin.x() - event->x());
8         int h = abs(origin.y() - event->y());
9         item->setRect(x, y, w, h);
10    }
11 }
```

Comme vous pouvez le constater, lors de chaque déplacement du stylet activé nous récupérons la position courante de ce dernier. On définit comme nouvelle origine du rectangle la position la plus éloignée entre l'origine initiale et la nouvelle position puis on récupère la hauteur et la largeur du nouveau rectangle avant de re-dessiner le nouveau rectangle.

Comme toujours, une fois l'action terminée, nous retournons la référence vers l'item créé pour le stocker dans l'historique des actions comme vous pouvez le voir dans le code qui suit :

```
1 QGraphicsItem* RectangleController::mouseReleaseEvent (
2     QGraphicsScene *scene, QMouseEvent *event) {
3     Q_UNUSED(scene)
4     Q_UNUSED(event)
5     return item;
6 }
```

3.6.3.3 Ellipse

Cet outil fournit la possibilité de dessiner une ellipse à partir d'un rectangle dans lequel elle est inscrite. Pour ce faire il nous faut, tout comme pour le rectangle, un point de départ, une hauteur et une largeur. Bien entendu, tant que le stylet est actif, l'ellipse doit être modifiée dynamiquement en fonction des déplacements du stylet.

Pour ce faire nous avons eu recours à la méthode `addEllipse()` de notre scène qui nous retourne un objet de la classe `QGraphicsEllipseItem`. Nous définissons comme position de départ la position initiale du stylet ainsi qu'une hauteur et une largeur nulle et étant un objet créé par l'appel d'une méthode de la scène, ce dernier est directement intégré à cette dernière comme nous pouvons le voir dans le code suivant :

```

1 void EllipseController::mousePressEvent (
2     QGraphicsScene *scene, QMouseEvent *event, QPen* pen) {
3     origin.setX(event->x());
4     origin.setY(event->y());
5     item = scene->addEllipse(event->x(), event->y(), 0, 0,*pen);
6 }
```

Il est à noté que nous conservons les positions de l'origine de l'ellipse car, tout comme pour le rectangle, de par les spécificités de la classe `QGraphicsEllipseItem` il nous était impossible de dessiner une ellipse inscrite dans un rectangle dont la hauteur ou la largeur est négative.

Pour pallier cette restriction, nous avons utilisé la même méthode que pour le rectangle à savoir :

```

1 void EllipseController::mouseMoveEvent (
2     QGraphicsScene *scene, QMouseEvent *event) {
3     Q_UNUSED(scene)
4     if (item) {
5         int x = origin.x() > event->x() ? event->x() : origin.x();
6         int y = origin.y() > event->y() ? event->y() : origin.y();
7         int w = abs(origin.x() - event->x());
8         int h = abs(origin.y() - event->y());
9         item->setRect(x, y, w, h);
10    }
11 }
```

Comme toujours, une fois l'action terminée, nous retournons la référence vers l'item créé pour le stocker dans l'historique des actions comme vous pouvez le voir dans le code qui suit :

```

1 QGraphicsItem* EllipseController::mouseReleaseEvent (
2     QGraphicsScene *scene, QMouseEvent *event) {
3     Q_UNUSED(scene)
4     Q_UNUSED(event)
5     return item;
6 }
```

3.6.4 Outil de sélection d'épaisseur de trait

L'outil de sélection d'épaisseur de trait est implémenté de manière légèrement différente des autres outils dans le sens où modifier l'épaisseur du trait modifie un aspect cosmétique mais ne remplace pas l'outil actuellement sélectionné. Il ne possède pas de contrôleur, il s'occupe uniquement de modifier le style courant stocké dans notre *DrawingController*.

Premièrement, regardons comment il a été implémenté et intégré à la *Toolbar* à l'aide du code suivant qui se trouve dans *ToolBar* :

```
1 QList<QIcon> icons;
2 for (int i = 0; i < 6; ++i) {
3     icons << QIcon(QString(":/tool/icons/thickness%1.png").arg(i + 1));
4 }
5
6 QStringList thicknesses;
7 thicknesses << "1 px" << "3 px" << "5 px" << "7 px" << "10 px" << "15 px";
8
9 QMenu* thicknessMenu = new QMenu(this);
10 for (int i = 0; i < thicknesses.count(); ++i) {
11     QAction *action = new QAction/icons.at(i), thicknesses.at(i), this);
12     action->setData(thicknesses.at(i).split(" ")[0].toInt());
13     action->setCheckable(true);
14     connect(action, SIGNAL(triggered()), this, SLOT(onThicknessChanged()));
15     thicknessMenu->addAction(action);
16 }
17 thicknessMenu->actions().first()->setChecked(true);
18 thicknessMenu->setStyleSheet("background: rgb(46, 46, 46);");
19
20 thickness = new QToolButton(this);
21 thickness->setIcon(QIcon(":/tool/icons/thickness.png"));
22 thickness->setPopupMode(QToolButton::InstantPopup);
23 thickness->setMenu(thicknessMenu);
24 thickness->setStyleSheet("QToolButton::menu-indicator{image: none;}");
25
26
27 addWidget(thickness);
```

Dans un premier temps nous créons un *QMenu* contenant une *QAction* par taille de trait que l'on propose (1 px, 3 px, 5px, ...). On connecte le signal *triggered()* de chacune de ces actions au slot *onThicknessChanged()*. Finalement on ajoute le menu contenant ces options dans notre classe *QToolButton* qui est à son tour ajoutée à la toolbar.

Ainsi lorsque l'on cliquera sur l'outil épaisseur de trait, un menu s'ouvrira permettant de choisir parmi une des épaisseurs disponible.

Lorsque l'on clique sur une épaisseur, le signal *triggered()* est émis activant ainsi le slot suivant :

```
1 void ToolBar::onThicknessChanged() {
2     const QList<QAction*>& actions = thickness->menu()->actions();
3     foreach (QAction* a, actions) {
4         a->setChecked(false);
5     }
6     QAction* action = qobject_cast<QAction*>(sender());
7     action->setChecked(true);
8
9     MainWindow* window = qobject_cast<MainWindow*>(parent());
10    if (!window) { return; }
11    window->onThicknessChanged(action);
12 }
```

Comme on peut l'observer lorsqu'une épaisseur est choisie on s'occupe d'abord de modifier dans l'interface graphique l'indicateur correspondant à l'épaisseur sélectionnée.

Puis on fait appel à la méthode *onThicknessChanged()* de notre *MainWindow* :

```
1 void MainWindow::onThicknessChanged(QAction* action) {
2     QPen* pen = controller->getPen();
3     pen->setWidth(qvariant_cast<int>(action->data()));
4 }
```

Qui, elle, s'occupe de récupérer la taille liée à l'action et de mettre à jour le style du crayon conservé dans notre *DrawingController*.

3.6.5 Outil de gestion de pile d'actions

Comme précisé dans les sous-chapitres précédents, notre *DrawingController* conserve les actions effectuées afin de pouvoir les supprimer ou les restaurer. Pour ce faire notre contrôleur contient deux piles LIFO (Last In First Out).

La pile *lastActions* contient toutes les actions ayant été effectuées jusqu'à maintenant afin de nous permettre de les supprimer (Undo) tandis que la pile *nextActions* est utilisée pour stocker les actions que nous avons supprimé afin de nous permettre de les rétablir. Comme nous pouvons le voir dans le code suivant :

```
1 void DrawingController::mouseReleaseEvent (QMouseEvent* event) {
2     if (isEnabled) {
3         lastActions.append(drawController->mouseReleaseEvent (
4             drawing->scene(), event));
5         nextActions.clear();
6         modifToSave = true;
7     }
}
```

Nous pouvons observer que la pile *nextActions* est vidée après chaque insertion d'une nouvelle action. Donc dans le cas où des actions ont été supprimées, il sera impossible de les rétablir si une action a été effectuée entre temps.

3.6.5.1 Undo

Comme nous pouvons le voir dans le code ci-dessous, l'outil de suppression d'actions est implémenté de telle sorte que si la pile contenant les actions passées (*lastActions*) n'est pas vide, on récupère la dernière action effectuée et on la retire de la scène. L'action retirée est ajoutée à la pile contenant les actions supprimées (*nextActions*). Ceci afin de permettre de les rétablir. De plus le contrôleur est notifié qu'une action a eu lieu et qu'une sauvegarde doit être effectuée sous peine de perdre des changements.

```
1 void DrawingController::undo() {
2     if (canUndo()) {
3         QGraphicsItem* temp = lastActions.takeLast();
4         nextActions.append(temp);
5         drawing->scene()->removeItem(temp);
6         modifToSave = true;
7     }
8 }
9 bool DrawingController::canUndo() {
10     return !lastActions.isEmpty();
11 }
```

3.6.5.2 Redo

Comme nous pouvons le voir dans le code ci-dessous, l'outil de rétablissement d'actions est implémenté de telle sorte que si la pile contenant les actions supprimées n'est pas vide, la dernière action supprimée est retirée de la pile, est rétablie et rajoutée à la pile des actions effectuées. De plus le contrôleur est notifié qu'une action a eu lieu et qu'une sauvegarde doit être effectuée sous peine de perdre des changements.

```
1 void DrawingController::redo() {
2     if (canRedo()) {
3         QGraphicsItem* temp = nextActions.takeLast();
4         lastActions.append(temp);
5         drawing->scene()->addItem(temp);
6         modifToSave = true;
7     }
8 }
9 bool DrawingController::canRedo() {
10     return !nextActions.isEmpty();
11 }
```

3.6.5.3 Réinitialisation

Cette fonctionnalité n'est pas exactement un outil mis à disposition de l'utilisateur mais une fonctionnalité appelée lorsque l'utilisateur ouvre un nouveau fichier ou crée un nouveau fichier.

Notre *DrawingController* étant déjà instancié, on se doit de s'assurer que les piles d'actions sont vides.

Pour ce faire quand les slots gérant l'ouverture d'un fichier ou la création d'un nouveau fichier sont activés, il nous faut appeler une méthode vidant ces piles.

```
1 void MainWindow::onOpenTriggered() {
2     if(controller->toSave()) {
3         //
4         switch (ret) {
5             case QMessageBox::Save:
6                 onSaveTriggered();
7             // Clear actions history
8                 controller->resetUndoHistory();
9                 openFile();
10            case QMessageBox::Discard:
11            // Clear actions history
12                controller->resetUndoHistory();
13                openFile();
14                break;
15            case QMessageBox::Cancel:
16                // Nothing Happend
17                break;
18            default:
19                // should never be reached
20                break;
21        }
22    }
23 }
24 // ...
25 }
26 void MainWindow::onNewTriggered() {
27     if(controller->toSave()) {
28         //
29         switch (ret) {
30             case QMessageBox::Save:
31                 onSaveTriggered();
32                 drawing->scene()->clear();
33             // Clear actions history
```

```
34         controller->resetUndoHistory();
35     case QMessageBox::Discard:
36         drawing->scene()->clear();
37         // Clear actions history
38         controller->resetUndoHistory();
39         break;
40     case QMessageBox::Cancel:
41         // Nothing Happend
42         break;
43     default:
44         // should never be reached
45         break;
46     }
47
48 }
49 // ...
50 }
51
52 void DrawingController::resetUndoHistory() {
53     lastActions.clear();
54     nextActions.clear();
55     modifToSave = false;
56 }
```

3.7 Interfaçage tracking-dessin

3.7.1 Simulation et contrôle de la souris

Afin de transmettre les informations récupérées par le tracking à l'application de dessin, nous avons envisagé deux solutions différentes. La première fût qu'avec les coordonnées récupérées, le programme calcule la position relative dans l'application et exécute les actions correspondante. Il s'est vite avéré que cette solution deviendrait vite compliquée à utiliser en fonction de la sous fenêtre utilisée. En effet, une position donnée pourrait être associée à plusieurs actions différentes en fonction d'une sous-fenêtre ouverte (taille du trait, sélection de la couleur...) ce qui impliquerait donc de devoir garder un état. Un autre problème de cette solution est que nous voulons une application qui soit multi-plataforme, le rendu de l'interface de *Qt* n'étant pas le même sur tous les OS, rien ne nous garantissait qu'une position de curseur puisse être interprétée de la même façon quelque soit l'OS utilisé.

La solution qui a été retenue est de prendre directement le contrôle de la souris de l'ordinateur. Le gros avantage de cette manière de procéder est que l'interface graphique peut être implantée de manière standard en gérant des *MouseEvent* sur les différents éléments. Après quelques recherches il s'est vite avéré que les librairies fournies à cet effet par *Qt* sont limitées aux widgets de l'application. Nous avons donc décidé de créer un programme java annexe qui utilise la classe Robot qui permet de prendre le contrôle de la souris d'une manière très simple et intuitive. Ce programme communique via socket avec notre application principal qui lui indique comment déplacer la souris en fonction des information récupérées par le tracking. Après avoir reçu les instructions, il lui suffit de les exécuter pour que la souris bouge de manière synchronisée avec le stylet.

3.8 Clavier virtuel

Ce clavier virtuel implémenté reprend la spécification définie dans le cahier des charges. Pour rappel, celle-ci doit fournir une interface clavier laquelle doit faciliter une saisie utilisateur à l'aide du stylet. Pour ce faire, le modèle du clavier reprend la forme des claviers usuels dans les smartphones et les tablettes. Le concept de touche pouvant être mono-évalué et multi-évalué est bel et bien présent comme en témoigne les figures suivantes :

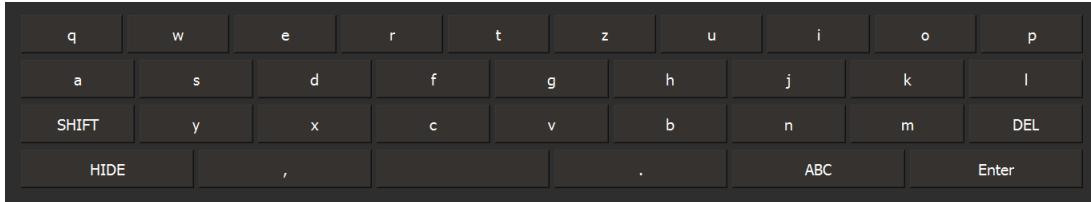


FIGURE 3.12 – Clavier virtuel selon le mode alphabétique



FIGURE 3.13 – Clavier virtuel selon le mode symbole

Du point de vue de son implémentation, un modèle de clavier est défini dans la classe *KeyboardModel* laquelle définit pour chaque ligne les touches du clavier. Chacune de ces touches fait l'objet d'une définition de deux valeurs. C'est ainsi que les modes mono-évalué et multi-évalué sont gérés. Une structure est définie en guise de modèle à cet effet :

```
1 struct KeyModel {
2     QString letter;
3     QString symbol;
4     KeyModel(QString letter, QString symbol) {
5         this->letter = letter;
6         this->symbol = symbol;
7     }
8 };
```

Le modèle du clavier correspondant à un ensemble de lignes de touches, la définition de la première ligne du clavier correspondrait à :

```
1 const QList<KeyModel> KeyboardModel::FIRST_ROW = QList<KeyModel>()
2     << KeyModel("q", "1") << KeyModel("w", "2")
3     << KeyModel("e", "3") << KeyModel("r", "4")
4     << KeyModel("t", "5") << KeyModel("z", "6")
5     << KeyModel("u", "7") << KeyModel("i", "8")
6     << KeyModel("o", "9") << KeyModel("p", "0");
```

Au moment d'afficher la disposition du clavier avec les symboles, un parcours de l'ensemble des touches du clavier sera réalisé. Celui-ci intervertira la valeur "alphabétique" par la valeur "symbole" et inversement.

Les contrôles graphiques utilisées pour la réalisation de ce clavier sont au nombre de trois :

- Un layout vertical correspondant à la classe *QVBoxLayout* regroupe les lignes du clavier.
- Un layout horizontal correspond à la classe *QHBoxLayout* représente une ligne du clavier et regroupe ses touches.
- Un bouton est utilisé en guise de touche clavier et correspond à une spécialisation de la classe *QPushButton* à savoir la classe *KeyButton*.

Certaines touches clavier sont caractérisées par des comportements différents de ceux des touches alphabétiques et des touches symboles. Concrètement, il s'agit des touches SHIFT, HIDE, DEL et BACKSPACE dont le comportement correspond à celui attendu d'un clavier usuel..

Sérialisation

Le menu propose différentes opérations or certaines telles que la sauvegarde du dessin ou l'ouverture d'un fichier nécessite une opération de sérialisation ou de-dé-sérialisation afin de pouvoir récupérer les données de notre vue afin de générer un rendu sous un format d'image utilisable ou réciproquement de récupérer les données d'une image et les rendre utilisable par notre vue.

Ces deux différents processus sont le sujet des sous-chapitres qui vont suivre.

3.8.1 Ouverture de fichier

Lorsque l'utilisateur souhaite ouvrir un fichier existant le slot suivant est activé au travers du menu de la GUI :

```
1 void MainWindow::onOpenTriggered() {
2     if(controller->toSave()) {
3         QMessageBox msgBox;
4         msgBox.setText("The document has been modified.");
5         msgBox.setInformativeText("Do you want to save your changes?");
6         msgBox.setStandardButtons(QMessageBox::Save | QMessageBox::Discard | QMessageBox::Cancel);
7         msgBox.setDefaultButton(QMessageBox::Save);
8         msgBox.setStyleSheet("QMessageBox { background: black; border: 2px solid #000; color: white; }");
9         msgBox.setWindowFlags(Qt::FramelessWindowHint);
10        int ret = msgBox.exec();
11
12        switch (ret) {
13            case QMessageBox::Save:
14                onSaveTriggered();
15                openFile();
16            case QMessageBox::Discard:
17                openFile();
18                break;
19            case QMessageBox::Cancel:
20                // Nothing Happend
21                break;
22            default:
23                // should never be reached
24                break;
25        }
26
27    }else{
28        openFile();
29    }
30 }
```

Comme nous pouvons le voir, dans un premier temps nous nous occupons de savoir si notre vue courante contient un dessin contenant des modifications non-sauvegardée. Si c'est le cas l'utilisateur se voit proposé la possibilité de sauvegarder les changements ou les ignorer.

Dans le cas où l'utilisateur décide de sauvegarder les modifications, onSaveTriggered() est appelé. Son comportement fait l'objet d'un sous-chapitre à venir et ne sera donc pas discuté ici.

Dans le cas où l'utilisateur décide d'ignorer les changements non-sauvegardés et d'ouvrir un nouveau fichier ou dans le cas où aucun changement n'a été effectué la méthode openFile() est appelé dont le code est le suivant :

```
1 void MainWindow::openFile() {  
2     QString fileName = SystemFileDialog::getOpenFileName();  
3     QPixmap img(fileName);  
4     drawing->scene()->addPixmap(img);  
5 }
```

La première ligne de cette méthode ouvre un dialogue permettant de sélectionner un fichier et retournant le nom du fichier souhaité. L'implémentation de ce dialogue est le sujet du sous-chapitre suivant.

Une fois le nom connu nous utilisons la classe QPixmap qui permet de fournir une représentation du fichier image. L'avantage de cette classe étant que la représentation générée par cette classe est utilisable par notre QGraphicsScene.

Nous utilisons donc cette représentation du fichier ouvert que nous ajoutons à notre scène.

3.8.2 Sauvegarde de fichier

Lorsque l'utilisateur souhaite sauvegarder le dessin qu'il a vient de réaliser ou des changements apportés à un fichier existant le slot suivant est activé au travers du menu de la GUI :

```
1 void MainWindow::onSaveTriggered() {  
2     QString fileName = SystemFileDialog::getSaveFileName();  
3     // creation du conteneur  
4     QPixmap pixmap(drawing->width(), drawing->height());  
5     // creation du painter allant servir      effectuer notre rendu  
6     QPainter painter(&pixmap);  
7     // selection qualite  
8     painter.setRenderHint(QPainter::HighQualityAntialiasing);  
9     // generation du rendu  
10    drawing->render(&painter);  
11    // enregistrement  
12    pixmap.save(fileName);  
13    painter.end();  
14    controller->setToSave(false);
```

Dans un premier temps un dialogue est ouvert permettant à l'utilisateur de choisir le répertoire de destination ainsi qu'un fichier existant à écraser ou donner un nouveau nom au fichier.

L'implémentation de ce dialogue est le sujet du sous-chapitre suivant.

Lorsque le nom du fichier est connu, nous utilisons la classe `QPixmap`, classe permettant de représenter une image et pouvant être utilisée comme service de dessin. Nous lui assignons les dimensions de notre vue qui correspondent aux dimensions de l'image que l'on souhaite sauvegarder.

Puis nous utilisons la classe `QPainter` qui s'occupe de réaliser des dessins de bas niveau et fournit le rendu au service de dessin associé (dans notre cas `QPixmap`).

Puis nous utilisons la méthode `render(QPainter)` de notre `QGraphicsView` qui s'occupe d'utiliser le `QPainter` fourni pour récupérer le contenu de la vue et générer un rendu qui, dans notre cas, est directement lié au `QPixmap` qui génère une représentation de l'image de la vue. Une fois notre représentation générée, il nous reste plus que la sauvegarder dans le fichier choisi précédemment.

3.8.3 Interface d'utilisation

3.9 Impression

Le menu nous propose de pouvoir imprimer notre image, pour ce faire nous avons utilisé la classe *QPrinter*. Cette classe est un service permettant de réaliser des dessins dans un format image utilisable par une imprimante.

Comme on peut le voir dans le code suivant :

```
1 void MainWindow::onPrintTriggered() {
2     QPrinter printer;
3     printer.setPageSize(QPrinter::A4);
4     printer.setOrientation(QPrinter::Landscape);
5     if (QPrintDialog(&printer).exec() == QDialog::Accepted) {
6         QPainter painter(&printer);
7         painter.setRenderHint(QPainter::Antialiasing);
8         drawing->render(&painter);
9         painter.end();
10    }
11 }
```

Nous créons notre *QPrinter* et lui fournissons certaines configurations par défaut tels que l'impression en mode paysage ainsi que le format A4 qui sont récupérés lorsqu'on utilise la boîte de dialogue *QPrintDialog*, classe fournissant une interface graphique permettant la gestion complète d'une impression.

Si l'on décide d'imprimer, soit que le bouton ‘ok’ est cliqué dans le dialogue d'impression, nous utilisons la classe *QPainter* qui s'occupe de réaliser des dessins de bas niveau et fournit le rendu au service de dessin associé (dans notre cas le *QPrinter*).

Puis nous utilisons la méthode *render(QPainter)* de notre *QGraphicsView* qui s'occupe d'utiliser le *QPainter* fourni pour récupérer le contenu de la vue et générer un rendu qui, dans notre cas, est directement lié au *QPrinter* qui envoie le résultat à l'imprimante.

Chapitre 4

Tests & Validation

Comme ce projet comporte une partie assez expérimentale ainsi que des technologies que nous n'avions jamais utiliser auparavant, les tests ont parfois été fastidieux. En effet, toute la partie du projet gérant le suivi du stylet a dû être testée et revue de nombreuses fois.

4.1 Stratégie de tests

Les stratégies de tests du système de dessin et du système de suivi du pointeur étant relativement différente, elles sont décrites en deux sous-sections.

De manière générale, la création de tests unitaires pour ce genre de projet est très délicate. Nous pensions pouvoir en développer pour tester une partie du programme, mais il n'y avait suffisamment de fonction pouvant être testée unitairement.

Tout au long du projet, des tests de bout en bout ont été effectué afin de tester l'entièreté du programme comme le ferait un utilisateur lambda. Cela nous a permis de relever plusieurs bugs, mais aussi d'améliorer grandement l'expérience utilisateur, notamment en agrandissant la taille des éléments cliquables, d'ajouter des extensions par défaut aux fichiers enregistrés, de guider l'utilisateur, etc.

4.1.1 Tracking Subsystem

Le développement du système de suivi de stylet était essentiellement guidé par les tests. Ainsi, une vingtaine de programmes de tests ont vu le jour afin de mieux comprendre les mécanismes et les moyens mis à disposition pour faire du traitement d'image.

En outre, ces programmes d'essais nous ont permis de découvrir le framework OpenCV en profondeur et de découvrir certaines possibilités auxquels nous n'avions pas pensé auparavant. Par exemple, lors de l'écriture du cahier de charges, nous pensions devoir placer la caméra le plus proche possible du projecteur afin d'éviter des calculs compliqués pour passer du référentiel de la caméra à celui du programme. Or, grâce à certaines fonctions puissantes d'OpenCV nous avons pu permettre à l'utilisateur de placer la caméra où bon lui semble, du moment que cette dernière à une vue complète de l'écran.

Ces tests ont aussi souvent conduit à une réflexion profonde sur la faisabilité des méthodes employées,

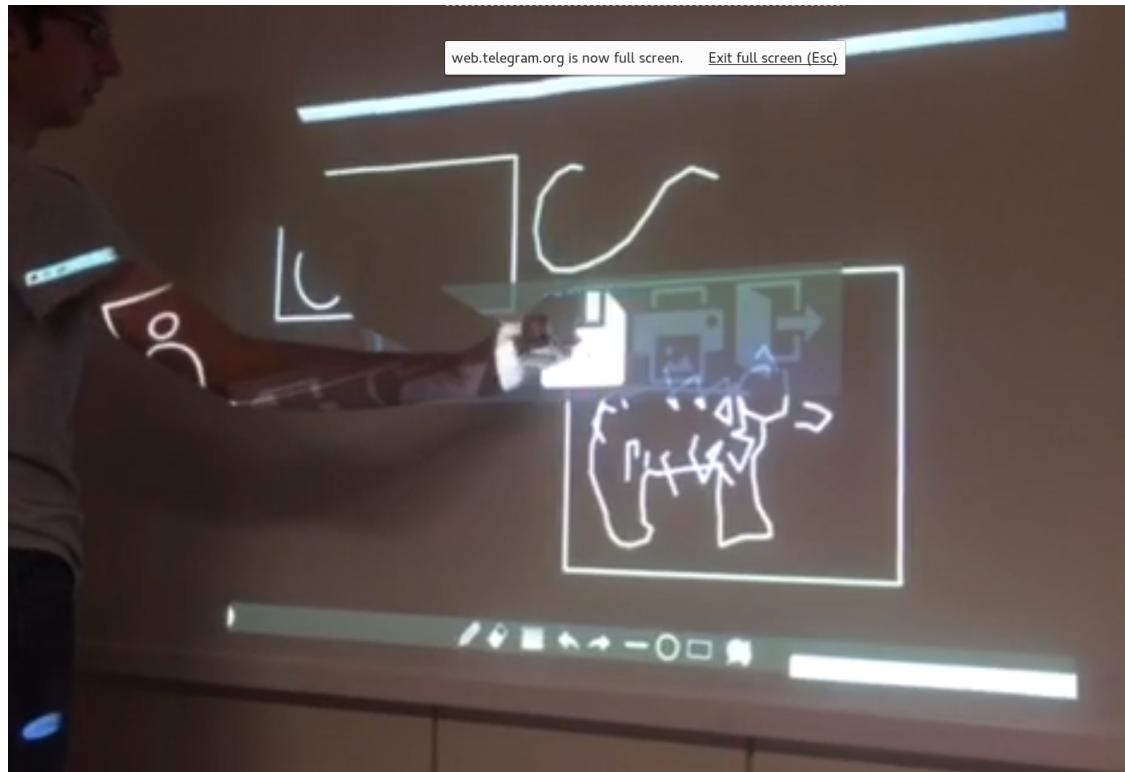


FIGURE 4.1 – Test grandeur nature de l’application lors de son développement

notamment sur la conception du stylet. En effet, au fur et à mesure des essais, nous nous sommes rendus compte que certaines méthodes de suivi étaient plus faciles à réaliser que d’autres, ce qui nous a poussé à transformer le stylet et le système de détection.

4.1.2 Drawing Subsystem

La méthodologie de test pour la zone dessin et ses outils étaient relativement simple. Comme chaque outil à son propre contrôleur, il nous a suffit de dessiner avec les outils un par un, en essayant à chaque fois de modifier leur épaisseur et leur couleur.

4.2 Matériel

L'application ayant besoin d'une caméra et d'un projecteur vidéo, il était parfois fastidieux d'effectuer des tests car ceux-ci demandaient une certaine installation. L'utilisation d'écrans alternatifs (tablette, télévision) et même d'objets simples (feuille de papier A4 verte) nous a permis d'effectuer des tests de suivi de stylet sans avoir besoin d'un projecteur vidéo à chaque instant.



FIGURE 4.2 – Lancement d'une calibration en utilisant une tablette comme écran vert

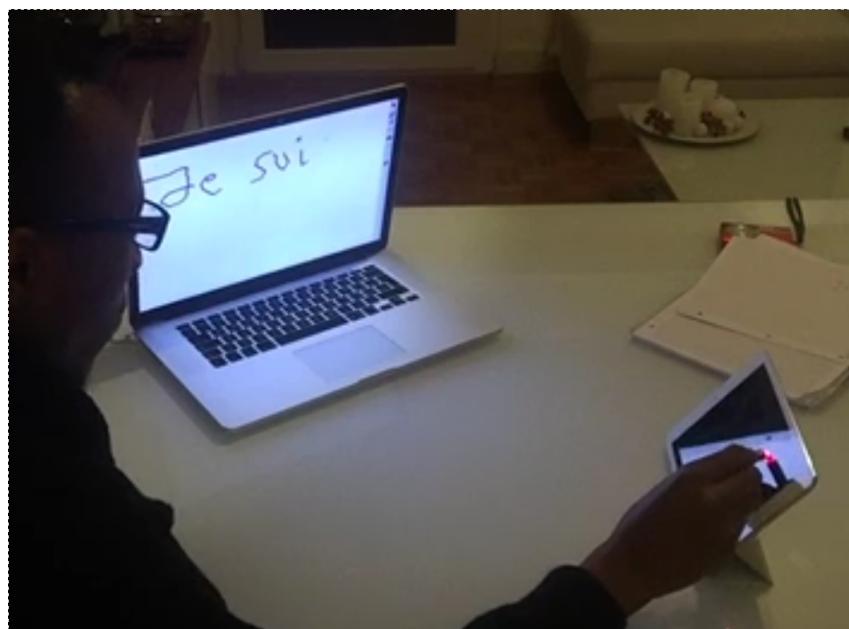


FIGURE 4.3 – Dessin en utilisant la tablette comme surface

4.3 Procédures de tests

4.4 Résultats

Les fonctionnalités suivantes ont été testées et ont passé les tests :

- Outils
 - Crayon
 - Gomme
 - Trait
 - Rectangle
 - Ellipse
 - Épaisseur du trait
 - Palette de couleurs
 - Undo
 - Redo
- Menu
 - Nouveau fichier
 - Sauvegarde de fichier
 - Ouverture de fichier
 - Impression
 - Quitter le programme
- Gestion des caméras
 - Détection des caméras
 - Sélection de la caméra
 - Relancement du processus de calibrage en cas d'échec
 - By-pass du calibrage si l'utilisateur n'a pas de caméra
- Tracking
 - Détection de l'écran vert (Estimation : 90% de réussite)
 - Détection et calibrage de la LED (Estimation : 80% de réussite)
- Multi-plateforme
 - Windows 7 ou supérieur
 - MacOS X
 - Linux (environnement Qt)
- Proposition de sauvegarde si le fichier a été modifié
- Clavier virtuel
- Ajout automatique de l'extension aux fichiers sauvegardés

Les fonctionnalités suivantes ont été testées mais n'ont pas passé les tests :

- Multi-plateforme
 - Linux (environnement GTK) : OpenCV et Qt utilisent deux versions incompatibles de GTK dans le même processus.

Chapitre 5

Problèmes connus

Après une série de tests, nous avons constatés que certains problèmes persistent et n'ont pas réussi à être réglés. Pour certain, ceci est dû à une incapacité de les régler à cause de leur grande complexité, pour les autres c'est juste qu'ils ont été découvert trop tard pour que nous puissions modifier le code et rendre le projet à temps.

Premièrement la luminosité de l'écran peut poser problème. En effet, si l'écran est trop clair et que la lumière du pointeur n'est pas suffisamment puissante, le tracking peine à reconnaître l'emplacement du stylet et engendre une mauvaise qualité de dessin. C'est pour cette raison que nous avons choisi que le fond du dessin soit de couleur noir et que tous les boutons du menu sont relativement sombres. Cependant si l'utilisateur rempli le fond de blanc (ou une autre couleur claire), il peut s'attendre à avoir des soucis.

Ensuite nous avons trouvé que si l'utilisateur quitte de manière forcée le programme durant la calibration, il peut arriver que l'affichage des caméra ne se ferme pas correctement et que l'ont soit obligé de les forcer à terminer.

Finalement, durant le calibrage, les fenêtres ne sont pas toutes forcées en premier plan et si l'utilisateur clique sur l'interface elle passe en premier plan cachant ainsi la procédure de calibrage qui n'est pas terminée. Cela a pour effet que le stylet n'est pas encore reconnu, par contre le menu peut être ouvert à l'aide de la souris. Le fonction de dessin n'est pas utilisable, le fait de quitter le programme à ce moment-là peut mener au bug décris juste au dessus.

Chapitre 6

Conclusion

Ce chapitre constitue la conclusion de ce document. Elle débute par l'état des lieux de la solution proposée : commenter les fonctionnalités implémentées et expliquer la non implémentation d'autres fonctionnalités. Ce premier point est conclut par une liste non exhaustive de propositions d'amélioration du projet. La suite de cette conclusion fait part des problèmes rencontrés et ce, durant la totalité du projet. Les problèmes peuvent être d'ordre organisationnel, technique et de planning. Vient ensuite une mise au point quant au bon respect du planning durant le projet. Pour ce faire, la planification initiale est rappelée. Celle-ci est ensuite comparé avec le planning final constaté à la fin de ce projet. Le déroulement du projet fait pour sa part une analyse critique tant sur les points positifs à retenir que sur les points négatifs à éviter en vue d'une potentielle suite de projet. En dernière instance, une synthèse générale du travail effectué conclut ce rapport.

6.1 Solution proposée

Cette section présente l'état des lieux de la solution proposée. Il s'agit d'une part de proposer une mise au point quant aux fonctionnalités qui ont pu être implémentées. D'autre part, il s'agit mentionner et de justifier la fonctionnalités manquantes à cette première version de l'application.

6.1.1 Fonctionnalités implémentées

Au terme de notre travail nous pouvons être content car nous avons réussi à nous en tenir à notre cahier des charges. Toutes les fonctionnalités de base que nous avions prévues sont présentes dans notre programme bien que certaines puissent encore être améliorées. Nous avons même pu ajouter le fait de permettre à l'utilisateur le rétablissement d'une action annulée. Ceci n'a pas été spécialement difficile à implémenter car tout (ou presque) avait déjà été mis en place pour l'annulation.

6.1.2 Fonctionnalités manquantes

Faute de temps, nous n'avons pas pu implémenter toutes les fonctionnalités supplémentaires que nous avions mentionnées dans notre cahier des charges. Certaines tel que le outils supplémentaires, pourraient être ajoutées relativement facilement grâce à la structure que nous avons décris précédemment.

6.1.3 Propositions d'amélioration

Bien que nous soyons satisfait du résultat de notre travail, un programme peut toujours être amélioré. C'est pourquoi nous avons pensé à quelques fonctionnalités qui nous semblerait possible d'ajouter. En voici la liste :

- Ajouter d'outils supplémentaires, ce qui était déjà proposé initialement dans le cahier des charges comme fonctionnalités supplémentaires.
- Amélioration du tracking, on pourrait même imaginer la possibilité d'utiliser une caméra infrarouge. Ceci nécessiterait par contre du matériel moins commun et rendrait donc moins accessible le programme.
- Ajouter la possibilité de re-calibrer le programme si la précision n'est pas satisfaisante ou dans le cas où le matériel aurait été déplacé pendant l'utilisation. Le tout sans avoir à relancer le programme en entier.

6.2 Problèmes rencontrés

Nous allons maintenant aborder les problèmes que nous avons rencontrés au cours du développement de ce projet. Nous les avons séparés en trois catégories ; technique, organisation et planification.

6.2.1 Problèmes techniques

La première difficulté que nous avons rencontrée est apparue bien plus tôt que nous ne l'aurions imaginé. En effet, dès que le cahier des charges a été accepté, nous avons tous dû installer l'environnement de développement. Cette tâche fut bien plus ardue que nous ne l'aurions cru. Malgré la facilité de trouver un tutoriel pour installer la librairie OpenCV pour sur l'environnement Qt, aucun d'eux ne s'est avéré exact. Nous avons donc dû chercher à exécuter un mélange de toutes les informations trouvées afin de finalement arriver à l'installer sur une machine. Cependant pour arriver à ce que les cinq membres du groupe puisse l'installerm il fallut encore bon nombre de recherches car nous ne travaillons pas tous sur le même système d'exploitation.

La plus grosse difficulté de ce projet a été d'obtenir un tracking d'une précision suffisante pour donner à l'utilisateur l'impression de pouvoir dessiner naturellement. Il nous a donc fallu tester plusieurs sortes de sources lumineuses pour trouver laquelle correspondait le mieux à notre programme. Ce soucis rend notre projet plus polyvalent car il peut maintenant être utilisé avec plusieurs outils différents.

6.2.2 Problèmes organisationnels

Nous avons très rapidement remarqué que le fait de travailler sur des machines différentes pose un véritable problème dans les importations de librairies. En effet, nous avons chacun des chemins d'accès différents. Le désagrément que cela nous a causé est clairement apparu lorsque nous avons voulu partager notre projet *QtCreator* grâce au gestionnaire de version. En effet, les chemins d'importation doivent être dans un fichier de configuration. Nous avons tous d'abord pensé à le rajouter dans les fichiers à ignorer au moment de la synchronisation. Cependant, il contenait aussi beaucoup d'informations relatives au projet qui sont mise à jours, notamment à l'ajout de nouvelles classes. Il nous était donc impossible de l'ignorer.

Nous avons donc dû travailler en modifiant le chemin manuellement à chaque partage sur le repository principal. Cette opération courte mais répétitive nous a plus fait perdre autant de patience que de temps.

Bien que la communication entre les membres du groupe n'a pas été mauvaise, elle nous a quand même fait perdre un temps précieux. Il n'est effectivement pas facile de bien coordonner le travail de cinq personnes. Certains oublis de communication à des moments clés ont entraîné la production de travail à double alors que d'autres parties du travail n'avaient pas. La rédaction de ce rapport en est un bon exemple, nous avons tout d'abord commencé à utiliser *Google Docs*, puis nous avons décidé de rédiger ce document en *LATEX* et donc de changer d'emplacement où stocker le fichier. Cette information n'ayant pas été bien comprise par tous les membres, certains ont continué à travailler sur l'ancien document alors que les autres avaient sur le nouveau.

6.2.3 Problèmes de planification

La planification est toujours un grand défi, pour ce travail nous estimons que notre planning a plutôt bien été. On pourra néanmoins noter quelques soucis de synchronisation, qui ont fait que certaines de nos ressources ont été bloquées en attendant que d'autres parties soient finies. Ceci dit, cela n'a pas causé de perte de temps majeur.

6.3 Respect du planning

Dans l'ensemble la planification a été respectée il y a malgré tout quelques différences que nous allons aborder ici.

Premièrement, nous avons du refaire un deuxième stylet pour notre programme car la version initiale n'était pas assez précise. En effet, nous étions parti sur la base d'un stylet à une seule lumière cependant la lumière ne pouvant pas être à même la surface de dessin, il y avait une marge d'erreur. Nous avons donc ajouté une deuxième lumière plus haute afin d'avoir un segment de longueur connue pour extrapoler de manière plus efficace la position de la pointe.

Ensuite nous avons grandement du allonger le temps nécessaire pour le tracking du stylet ainsi que l'amélioration de la précision. Ces deux étapes avaient été largement sous estimées et nous on fait prendre un certain retard. C'est pour cela que la partie de test avec le tracking souris pour l'application a été abandonnée. Nous avons effectué ces tests plus tard au moment de la mise en commun des deux parties du projet. Ce retard nous a forcer à allonger le temps prévu pour l'ensemble du projet et à prendre une semaine sur les vacances pour terminer le travail.

La rédaction du rapport a moins avancé que prévu au cours du projet que nous l'espérions du au nombreux changements d'orientation que nous avons du prendre en cours de route. Nous avons donc allongé le temps de rédaction en fin de travail pour rattraper ce qui n'avait pas été fait au préalable.

6.3.1 Planification initiale

Voici le diagramme de la planification initiale.

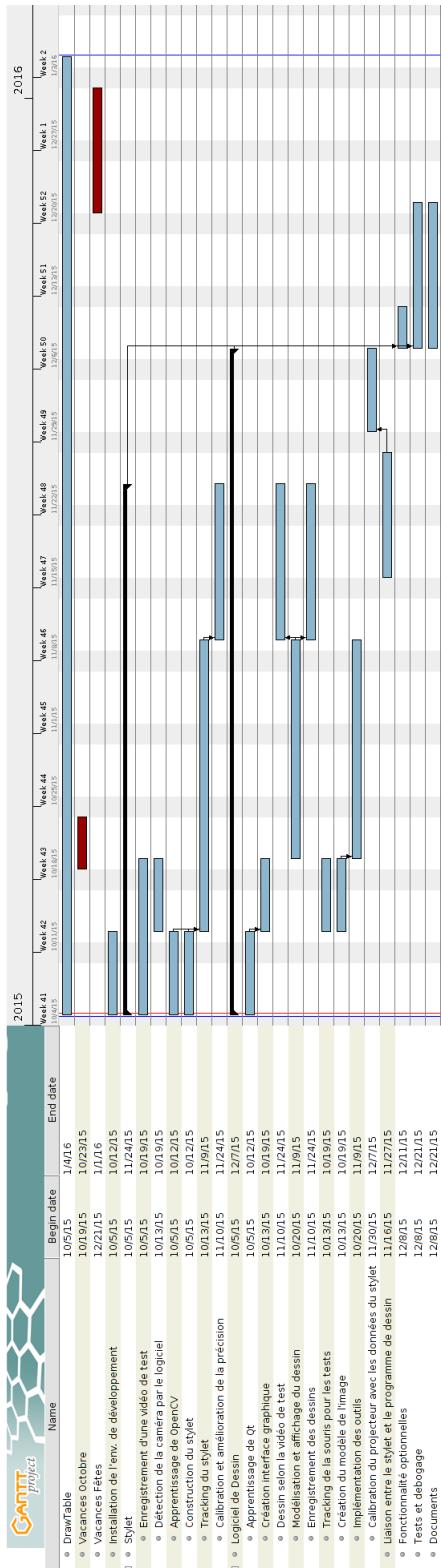


FIGURE 6.1 – Planification initiale des tâches

6.3.2 Évolution

Voici le diagramme de la planification finale.

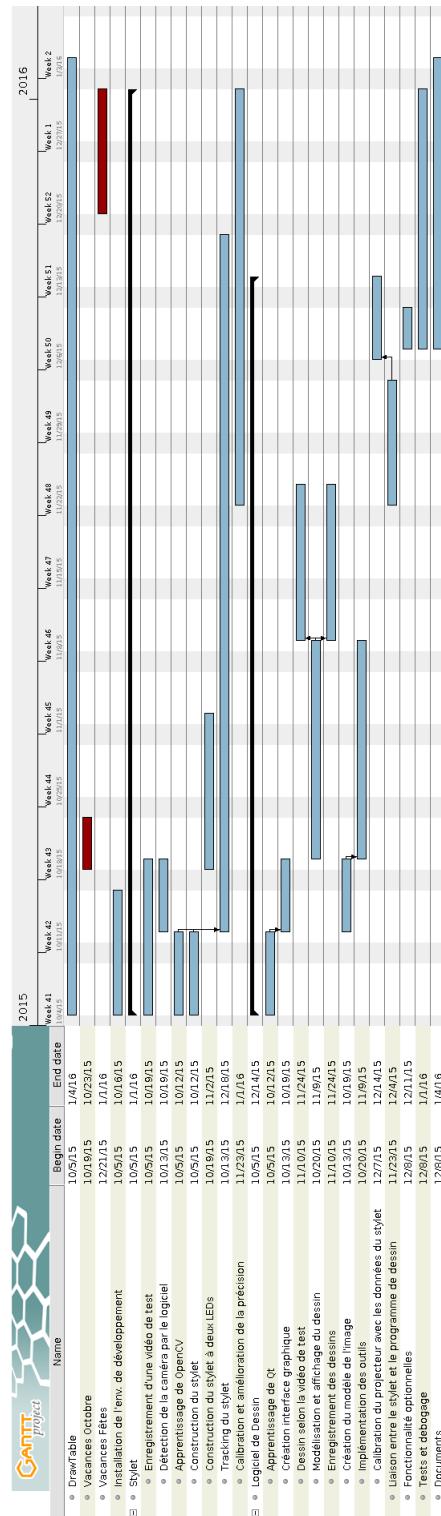


FIGURE 6.2 – Planification initiale des tâches

6.4 Déroulement du projet

6.4.1 Points positifs

Nous nous sommes lancés dans un projet touchant à des technologies dont nous avions aucune connaissance. Même si un grand travail a été fait pour se mettre à niveau, il est toujours bon de se rendre compte après coup du travail accompli ainsi que des connaissances acquises tout au long du projet. Connaissances qui pourront, nous l'espérons, nous être utile à l'avenir.

L'idée de tirer les groupes au hasard semblait être plutôt pénalisante à prime abord, cela c'est finalement révélé être une expérience enrichissante. Cela nous a permis de voir d'autre manière d'appréhender le travail que ce dont nous avions l'habitude. De plus cela nous a forcés à nous organiser mieux pour communiquer en tant que groupe alors que nous ne nous connaissions pas forcément bien avant de commencer. Même si ça n'a pas été facile au début, nous en retirons une bonne expérience.

6.4.2 Points négatifs

Il nous a été difficile d'organiser notre temps entre ce cours et les autres. Lors des rendus des travaux pour les autres cours du semestre, nous avons du laisser de côté ce projet ce qui nous a fait prendre un peu de retard que nous avons dû rattraper en faisant des heures d'affilée qui ont été pénibles et qui ont sûrement nuit à la qualité finale de notre travail. Une meilleure appréhension de la charge travail à l'avenir serait une bonne idée.

6.5 Synthèse

Ce projet nous a permis de nous familiariser avec des technologies de traitement d'images. C'est un domaine qui nous intéressait particulièrement et c'était une bonne occasion de pouvoir explorer ce domaine, étant donné que ce ne sont pas des choses que nous voyons durant les cours. Or, le traitement d'image est un domaine important dans l'industrie et maîtriser ces technologies peut être un atout important.

Le projet étant relativement expérimental, nous avons aussi été confrontés à des problèmes inconnus et nous avons dû adapter notre planning et notre cahier de charges. Cela était parfois difficile à faire, mais la motivation dans le projet nous a permis de nous accrocher et de résoudre les problèmes qui survenaient. Un des avantages de ce projet était que nous étions préparés à être confrontés à plusieurs problèmes difficiles et nous avons ainsi pu les aborder dès les premiers jours.

Le partage des tâches durant le projet était agréable pour chacun, la charge de travail était relativement bien répartie et chaque membre du groupe a fourni une bonne quantité de travail. En effet, selon notre expérience, les groupes de plus de trois personnes fonctionnent rarement aussi bien, étant donné que certains comptent sur les autres. Dans notre cas, chacun y mettait du sien et cela nous a permis de relever les défis que nous nous étions imposés.

Annexe A

Cahier des charges

A.1 Contexte

Ce projet de groupe se déroule dans le cadre de la formation Bachelor HES de la Haute-Ecole d'Ingénierie et de Gestion du canton de Vaud. Il compte pour une unité d'enseignement du plan d'étude de l'orientation Informatique logiciel (IL) du département des Technologies de l'Information et de la Communication (TIC) de l'école. Il se réalise durant le cinquième semestre de l'année de formation 2015/16.

A.2 Membres du groupe

Les personnes suivantes constituent les membres du groupe de l'équipe de projet :

- Sacha Bron - *Chef de groupe*
- David Villa - *Chef suppléant*
- Paul Ntawuruhunga
- Yassin Kammoun
- Marc Pellet

A.3 Objectif du projet

L'objectif de ce projet est de concevoir un outil de dessin assisté par ordinateur permettant à l'utilisateur de réaliser ses dessins de la manière la plus naturelle possible. À terme, l'utilisateur dessinera directement sur sa table ou n'importe quelle autre surface plane à l'aide d'un stylet et son dessin sera projeté sur son plan de travail, donnant à l'utilisateur l'impression de dessiner avec un crayon et une feuille.

A.4 Fonctionnalités principales

Les fonctionnalités principales du projet sont les suivantes :

- Dessin à l'aide d'un stylet.
- Projection de l'image sur le plan de travail.

- Alignement de l'image avec la position du stylet.
- Calibrage du stylet.
- Plan de travail avec barre d'outils.
 - Une barre d'outils sera projetée dans une zone du plan travail permettant à l'utilisateur de sélectionner un outil.
 - Outils de dessin : crayon, gomme.
 - Outils de forme : ligne droite, carré, cercle.
 - Outils de couleur : palette de couleurs.
 - Outils de choix d'épaisseur.
- Annulation d'une action
 - Une pile d'actions est enregistrée et un bouton dans la barre d'outils permet de remonter cette pile, annulant ainsi les dernières modifications apportées au dessin.
- Enregistrement et importation du dessin
 - Possibilité d'enregistrer ou importer son dessin au format PNG.

A.5 Fonctionnalités supplémentaires

Les fonctionnalités supplémentaires du projet sont les suivantes :

- Site vitrine.
- Pipette.
- Remplissage.
- Rétablissement d'une action.

A.6 Mockup

La figure suivante illustre une ébauche de l'interface graphique utilisateur devant correspondre au plan de travail :

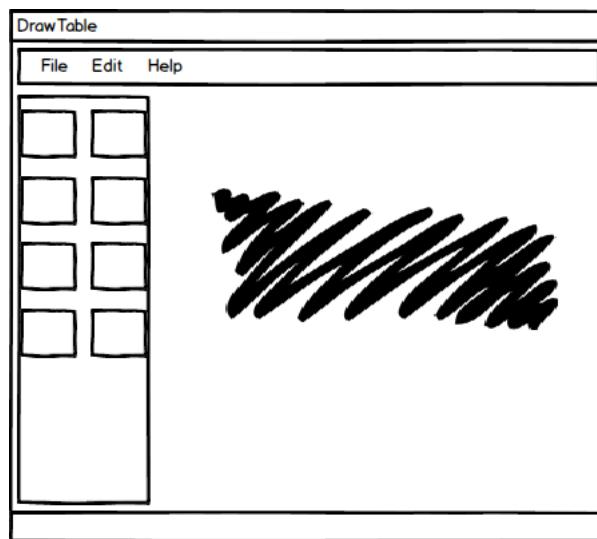


FIGURE A.1 – Mockup de l'interface graphique utilisateur

L'interface graphique utilisateur est divisée en trois parties à savoir la barre des menus, la barre d'outils et la zone de dessin.

A.7 Prototype du stylet

Le prototype de stylet est conçu par le chef de projet. Sa conception nécessite les composants suivants :

- LED.
- Un interrupteur.
- Une résistance.
- Une batterie.
- Un boîtier.

La figure suivante propose une ébauche de prototype du stylet :



FIGURE A.2 – Prototype initial du stylet

A.8 Principe de fonctionnement

Ce projet contient trois principales difficultés : la détection de contact du stylet sur la table, le suivi de la position du stylet (*tracking*) et la précision de l'entier du système.

Pour cela nous avons imaginé un stylet muni d'une mine spéciale. Lors d'une pression sur le plan de travail, cette dernière presse sur un bouton qui enclenche une diode électroluminescente placée proche de la mine. Une caméra placée au-dessus du plan de travail filme la scène et envoie l'image au logiciel chargé de suivre la position de la mine du stylet en détectant cette LED.

Le programme de dessin récupère les coordonnées du stylet et les interprète de manière à pouvoir projeter le dessin sur le plan de travail à l'aide du projecteur vidéo.

L'application comportera différents threads : un thread de l'application sera consacré au tracking de la LED, un autre récupérera les coordonnées pour les interpréter sur le plan de travail en fonction de l'outil sélectionné.

A.9 Public-cible

L'application vise avant toute chose tout amateur de dessin souhaitant se passer d'une feuille.

A.10 Technologies utilisées

Les technologies utilisées pour le développement de l'application et le suivi du stylet sont les suivantes :

- Librairie OpenCV - Pour le suivi du stylet en temps réel.
- Framework Qt - Pour l'interface graphique du plan de travail.

A.11 Matériel requis

Les points suivants constituent le matériel requis pour l'utilisation de l'application.

- Un prototype de stylet faisant office d'outil de dessin.
- Une caméra permettant de traquer les mouvements du stylet.
- Un projecteur vidéo permettant de retranscrire le dessin de l'utilisateur.
- Un plan de travail permettant de dessiner.
- Un support prévu pour la disposition de la caméra et du projecteur au-dessus du plan de travail.

A.12 Plateformes supportées

De par la nature du framework Qt, l'application est automatiquement multi-plateforme ; elle est donc supportée par les environnements usuels. Il s'agit entre autres des éditions Windows, des systèmes Mac OS X et des différentes distributions Linux. Un simple travail de recompilation du code source sur l'environnement désiré permet de disposer d'une version compatible de l'application.

A.13 Déploiement

A.13.1 Installation de l'application

Le déploiement de l'application suit la procédure standard des environnements supportés :

- Pour un environnement Windows, le déploiement se réalise par le biais d'un installateur.
- Pour un environnement Mac OS X, le déploiement se réalise par le biais d'un fichier DMG.
- Pour une distribution Linux (Arch), le déploiement se réalise par le biais d'un paquetage.

A.13.2 Mise en place du matériel

La mise en place du matériel requis est décrite par la procédure suivante :

1. Disposez le support de travail.
2. Posez le stylet sur le support de travail.
3. Placez la caméra au-dessus du support de travail.
4. Placez le projecteur vidéo au-dessus du support de travail.

Les schémas suivants illustrent un environnement de travail idéal :

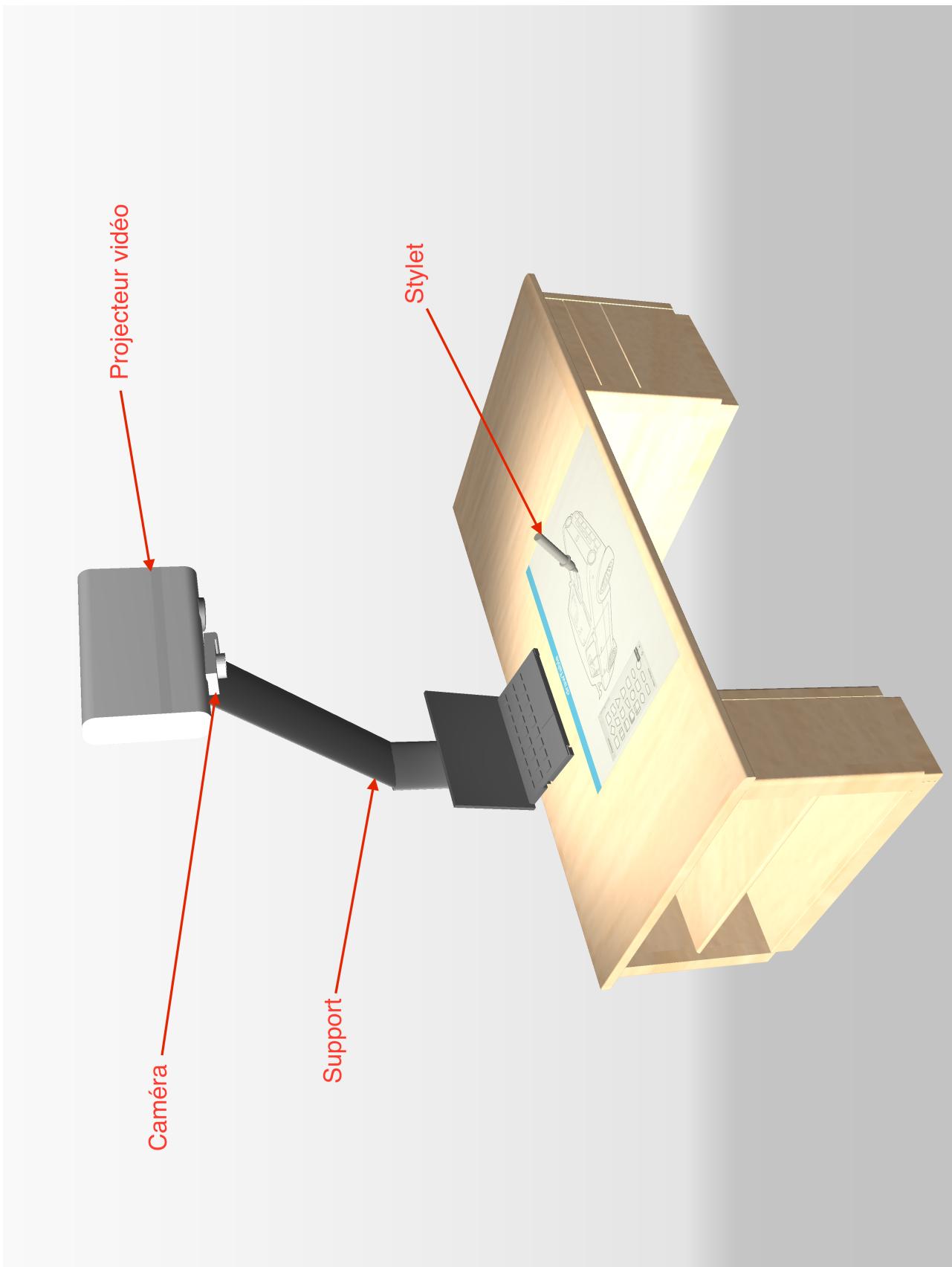


FIGURE A.3 – Vue en perspective d'un environnement idéal

- Tracking du stylet grâce à la caméra**
- La caméra détecte la LED du stylet
 - La LED s'allume lors d'une pression sur la mine.

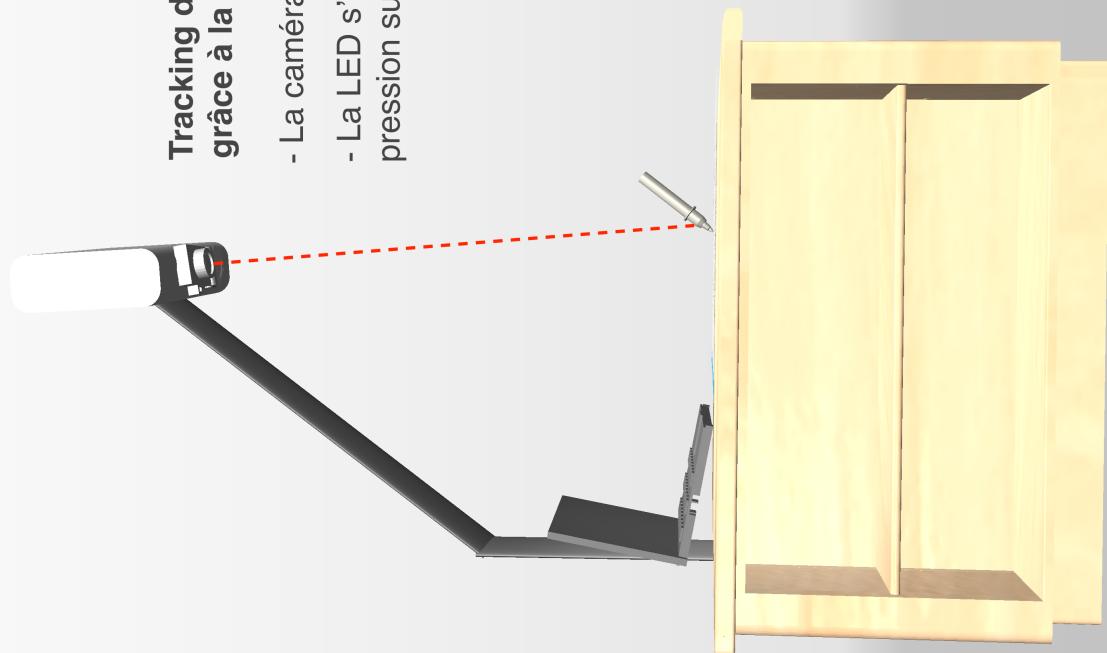


FIGURE A.4 – Vue de côté d'un environnement idéal

A.14 Déroulement du projet

A.14.1 Planification des tâches

Le diagramme de Gantt qui suit présente la planification des tâches du projet :

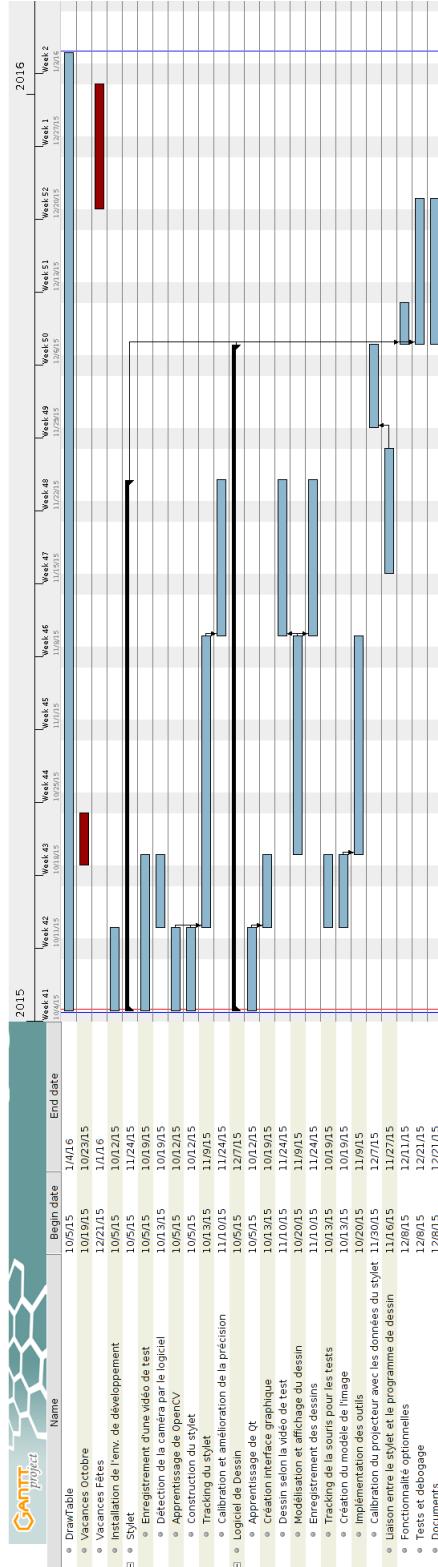


FIGURE A.5 – Planification initiale des tâches

A.14.2 Description des tâches

Les points suivants constituent un bref descriptif des tâches du projet :

- Enregistrement d'une vidéo de test
 - Afin de pouvoir tester les différents outils du logiciel de dessin avant que l'outil de tracking ne soit terminé, une vidéo contenant différents mouvements sera enregistrée.
- Apprentissage d'OpenCV
 - Pour toutes les opérations de tracking du stylet, nous allons utiliser la librairie OpenCV. Il sera donc nécessaire de prendre le temps d'en apprendre les bases avant de se lancer à corps perdu dans l'implémentation.
- Construction d'un stylet
 - Un stylet de notre propre création sera utilisé pour cette application, le matériel nécessaire à la fabrication de ce dernier est détaillé dans la description du prototype du stylet.
- Tracking du stylet
 - A l'aide d'une caméra, nous détecterons les mouvements du stylet, il faudra donc récupérer ces informations et les transmettre au logiciel de dessin.
- Apprentissage de Qt
 - Apprentissage des librairies Qt pour ce qui concerne principalement les interfaces graphiques et les librairies de dessin.
- Création de l'interface graphique
 - Conception de l'interface graphique du logiciel. Celle-ci contiendra un menu permettant d'importer/exporter des images, d'un plan de travail sur lequel travailler ainsi qu'une boîte à outils contenant différentes fonctionnalités telles que le crayon, la gomme, une palette de couleurs, le choix de l'épaisseur du trait.
- Tracking intermédiaire de la souris
 - Afin de permettre de tester le bon fonctionnement de l'implémentation des différents outils, ceci d'une manière accessible à tous les développeurs, il faut implémenter une fonctionnalité permettant de dessiner à l'aide de la souris.
- Création du modèle de l'image
 - Conception et création du modèle qui sera utilisé par le logiciel pour représenter notre image.
- Implémentation des outils
 - Implémentation des fonctionnalités obligatoires disponibles dans notre boîte à outils. Se référer à la description des fonctionnalités pour en connaître l'inventaire.
- Enregistrement des dessins
 - Exportation de notre représentation de l'image en une image au format .PNG.
- Modélisation et affichage du dessin
 - Implémentation des méthodes gérant l'affichage et la modélisation de notre image.
- Calibrage du projecteur avec les données du stylet
 - Calibrage permettant de faire concorder la position de l'image projetée avec la position effective du stylet.
- Dessin selon la vidéo de test
 - Utilisation d'une vidéo de test pré-enregistrée utilisant les méthodes de tracking implémentées

permettant de tester les différents outils sans pour autant avoir le matériel à disposition.

- Liaison entre le stylet et le logiciel
 - Liaison entre les données collectées par le système de tracking avec les méthodes d'affichage du logiciel.

Annexe B

Journal de travail

B.1 Semaine 1-2 : 14-27 sep 2015

Avant toute chose, il a fallu déterminer le sujet de notre travail. Après de longues discussions au sein du groupe, nous avons soumis deux propositions de sujet. La première est un outil utilisant l'API web du jeu *League of Legends* afin de déterminer de manière statistique, avant le lancement d'une partie, laquelle des deux équipes a le plus de chance de gagner. La deuxième proposition est un logiciel de dessin utilisant le tracking vidéo comme système de commande. Après discussion avec notre professeur, nous optons pour la deuxième proposition.

B.2 Semaine 3-4 : 21 sep - 4 oct 2015

La deuxième étape a été de rédiger le cahier des charges qui serait notre fil conducteur tout au long de ce travail. Ceci ne fût pas chose aisée notamment car nous nous lancions dans l'inconnu. En effet, aucun de nous n'avait encore travailler de près ou de loin avec le tracking. Il nous a donc été difficile d'imaginer le temps que pouvait nécessiter la prise en main d'une telle technologie.

B.3 Semaine 5 : 5-11 oct 2015

Notre cahier des charges ayant été validé malgré un petit manque de précision dans notre planning, nous avons pu commencer à travailler. La première tâche que nous avons tous eu à accomplir a été d'installer l'environnement de développement. Cela ne fût pas aussi banal que nous ne l'imaginions. En conséquence, nous y avons passé beaucoup plus de temps que prévu. Entre temps, Sacha nous a créé la première ébauche de ce qui deviendra plus tard notre stylo de test, dit le stylet.

B.4 Semaine 6-7 : 26 oct - 8nov 2015

L'environnement étant prêt, nous avons pu nous lancer dans la première partie du développement à proprement parlé. Nous nous séparons en deux groupes avec deux objectifs bien distincts. Sacha et Paul se lancent dans l'apprentissage d'OpenCV et mettent en place les bases du tracking. Pendant ce temps-là, le reste du groupe met en place le programme de dessin qui devra être contrôlé plus tard par le stylet.

B.5 Semaine 8 : 9-15 nov 2015

La présentation intermédiaire étant prévu d'ici une semaine, chacun des deux groupes finalise sa partie tout en préparant une démonstration. En parallèle nous préparons le PowerPoint pour notre présentation. À la fin de la semaine, nous sommes fin prêt pour la présentation du lundi suivant.

B.6 Semaine 9-10 : 16 nov - 29 nov 2015

La présentation s'est bien passée, nous avons pu annoncer que nous étions dans les temps. Nous continuons donc sur notre lancée. Le groupe du tracking continue à affiner la précision ainsi qu'à trouver une manière de calibrer notre programme. Une modification du prototype initial du stylet est apportée car il ne nous est pas possible d'extrapoler avec une précision suffisante la position de la mine pour dessiner. Marc quant à lui commence à chercher comment transmettre les données récupérées par le tracking à notre interface graphique et développe le contrôleur de la souris. Le reste du groupe continue sur le logiciel en ajoutant les fonctionnalités nécessaires à accueillir le programme de tracking.

B.7 Semaine 11-12 : 30 nov - 13 déc 2015

Le moment de mettre toutes les briques ensemble est arrivé. Cependant la grande charge de travail que nous avons à fournir dans le cadre des cours à côté nous fait prendre un peu de retard. Nous nous en rendons compte un peu tard et le projet stagne plus que nous l'avions imaginé. Nous arrivons cependant à commencer la mise en commun du projet.

B.8 Semaine 13 : 14-19 déc 2015

Le groupe se rend compte que le rendu approche à grand pas et que le retard devra être rattrapé. Nous décidons que nous passerons nos vacances à travailler s'il le faut. La mise en commun est terminée et nous avons le plaisir de voir que nous arrivons enfin à dessiner. Paul continue à affiner le tracking pour améliorer la précision pendant que le reste du groupe se lance dans la rédaction de toute la documentation qui sera à rendre avec le projet.

B.9 Semaine 14 : 4 janvier 2016

Quelques dernières modifications sont apportées au rapport avant de le relier et de graver le CD contenant notre rendu. Le rendu étant passé, nous pouvons commencer à préparer la présentation finale qui aura lieu la semaine suivante.

B.10 Semaine 15 : 11 janvier 2016

Présentation finale.

Annexe C

Planification

C.1 Planification initiale

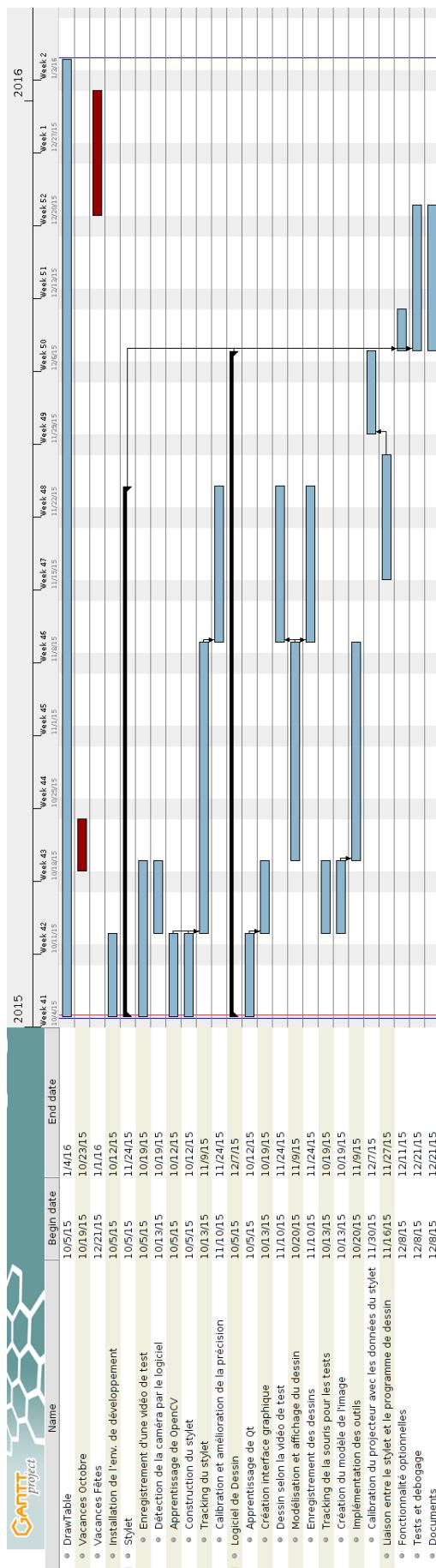


FIGURE C.1 – Planification initiale du projet

C.2 Planification finale