# Rapport de Stage fin de DUT - SuperBeeLive

Olivia SERENELLI-PESIN

July 26, 2019

# Remerciements

J'adresse mes remerciements aux personnes qui m'ont permis de réaliser ce stage dans l'équipe de SuperBeeLive.
Tout d'abord Matthieu ROUSSET, initiateur du projet SuperBeeLive à l'IBMM (Insitut Biomoléculaire Max Mousseron) qui m'a accueilli au sein de son équipe.
Ensuite, Capucine CARLIER pour ses nombreuses explications biologiques sur les abeilles ainsi que sa disponibilité afin de comprendre au mieux les enjeux et les besoins des biologistes pour mon projet.
Enfin, Sebastien DRUON pour m'avoir encadré, aidé à m'intégrer dans le milieu de la recherche et aidé sur une multitude de sujets, aussi bien du point de vu universitaire que sur les tâches qui m'ont été confiées.

# Contents

# Chapter 1

# Introduction : Présentation du projet de recherche

## 1.1 La structure d'accueil

Le projet étant réalisé par plusieurs laboratoires de recherches, j'ai dû évoluer au seins de différentes organisations. Tout d'abord il y a mon employeur, l'Université de Montpellier, qui englobe d'autres structures où j'ai pu évoluer. L'université a été, pour moi, une entitée administrative.
Avec elle, le CNRS (Centre National de Recherche Scientifique) accueille dans ses locaux le rucher expérimental où nous pouvons effectuer nos tests d'installation pour le projet. J'ai pu m'y rendre plusieurs fois afin d'observer les abeilles et voir le travail déjà effectué et évoluer au fur et à mesure. C'est également chez eux que nous aurons un serveur d'installé.
Ensuite, l'IBMM est le laboratoire qui a engagé l'argent lié au projet afin de pouvoir me recrutrer lors de ce stage. C'est également une structure qui a été seulement administrative de mon point de vu puisqu'ils m'ont envoyés travailler dans les bureaux du LIRMM (Laboratoire d'informatique, de robotique et de microéléctronique de Montpellier), autre laboratoire de recherche, afin que je puisse être aux côtés de Sébastien DRUON qui m'aura donné la grande majorité de mes tâches à effectuer. J'ai pu y avoir mon bureau en face de M. DRUON, me permettant d'avoir une certaine autonomie mais aussi de pouvoir faire appel à lui facilement lorsque j'en avais besoin.

C'est dans ce contexte de recherche que j'ai pu découvrir et travailler sur le projet SuperBeeLive.

## 1.2 Projet SuperBeeLive

### 1.2.1 Le Global

La santé et le développement des abeilles est aujourd'hui une question de plus en plus étudiée. Les bouleversements majeurs de notre planète et de l'activité humaine se traduisant par une augmentation alarmante de la mortalité des colonies et une chute de la production du miel dans nos pays développés, il est urgent de se préoccuper de leur futur. La situation des abeilles domestiques alerte le pouvoir public sur l'accélération de la dégradation de la biodiversité des pollinisateurs domestiques et sauvages, et de la flore qui en dépend. Ces dégâts sont dûs, entre autre, à l'apparition et la prolifération d'espèces invasiges pour les abeilles, provoquant maladies et déteriorations.

Le projet consiste en la structuration de plusieurs collaborations existantes ou nouvelles autours du developpement d'une ruche plate instrumentée destinée au monitorage détaillé de la santé de l'abeille et des écosystèmes. Son but est de pouvoir répondre à des questions clés, notamment autours des mécanismes physiopathologiques et des maladies chroniques dûes au parasites ainsi qu'aux altérations de l'écosystèmes et des qualités nutritives des produits des ruches.
Répondre à ces questions permettra de regrouper différentes solutions technologiques systèmatiques, automatiques et non invasives à la collection de données usuelles déterminantes dans chacun des thèmes abordés. Les différentes travaux déjà effectués autours de ce sujet ne visaient qu'un seul type de problème à la fois, ne permettant pas une vision globale des difficultées rencontrées par les abeilles. Notre but est de réunir les différentes données qui peuvent être utilisés pour étudier l'influence des éléments et événements extérieurs sur leur santé et leur cadre de vie.

Concrétement, l'équipe de SuperBeeLive va concevoir une ruche plate afin d'y mettre en place plusieurs type de capteurs (hygrométrie, vibrations, température interne et externe, etc) ainsi que des caméras qui filmeront l'intérieur et l'extérieur de la ruche.

Cette instrumentation nous permettra dans un premier temps d'observer les abeilles afin de visualiser leurs comportements (danse, regroupement, protection, etc) ainsi que les parasites comme les frelons asiatiques ou les varroas. Une fois ces observations faites et documentées, il y aura assez de matière pour pouvoir créer des algorithmes qui reconnaitront automatiquement ces comportements pour en sortir des informations spécifiques.
Par exemple, un des buts et de repérer la danse d'une abeille qui permet aux autre d'indiquer où se trouve du polen et d'en extraire les informations qu'elle transmet (la direction à prendre, le temps à parcourir pour y arriver etc).
Avec ces recherches, il serait possible de mettre en place une vitrine web des caméras streamées et commentées automatiquement en temps réel, permettant à d'autres chercheurs d'avoir une ressource permanente pour travailler mais aussi au grand public d'avoir un accès plus restreint mais pédagogique sur le comportement des abeilles.

### 1.2.2 Mon rôle durant le stage

Pour ce stage, plusieurs missions m'ont été affectées, une qui est devenue ma principales et d'autres, plus courtes dans le temps.

Comme dit plus haut, le but premier de l'installation des caméras sur la ruche est de pouvoir observer et annoter des vidéos manuellement pour ensuite rendre cette tâche automatique. Seulement, ces annotations doivent être encadrées afin d'éviter que des données ne se perdent, et que les outils utilisés pour le faire ne soient pas les même d'une personne à une autre nous donnant alors des fichiers non uniformes et plus difficiles et long à traiter une fois réunis.
Ainsi, nous avons entreprit de créer un logiciel d'annotation, permettant de regarder en direct les caméras et de sauvegarder des morceaux de vidéos afin de pouvoir dessiner simplement dessus (encercler, mettre une flèche, encadrer, etc) et écrire quelques mots sur ce qu'on y observe. Ces vidéos seraient sauvegardées dans un fichier contenant la vidéo et les annotations et pourront être visualisés de nouveau dans ledit logiciel mais aussi dans un lecteur plus classique mais sans les annotations.
Au final, celui-ci alégera le travail des biologiste, qui auront un outils sur mesure pour annoter les vidéos, mais aussi de M. Druon, pour qui il sera plus simple de récupérer et traiter les données.

Il est évident que dans un tel projet, beaucoup de données seront transmises et stockées. Ainsi, toute une partie d'administration système est à gérer. Dans notre cas, nous avons deux tâches importantes.
D'abord, la question du stockage des données commençait à se poser lors de mon arrivée en stage. Il fallait choisir, acheter, installer puis configurer un serveur de stockage dans la salle serveur du CNRS.
Ensuite, le rucher du CNRS où notre ruche expérimentale est installé n'a pas de configuration réseau déjà établie. Une connexion par fibre optique est prévue, mais la suite de l'installation devra être gérée par nous même. Comme pour le serveur, il faudra choisir, installer et configurer un switch dans le rucher.
Pour ces deux tâches, la même problématique est soulevée : il faudra penser au grand nombre de données qui devront transiter sur le réseau et donc prévoir du matériel adapté.
D'autres réalisations auraient pû m'être affectées, comme le dimensionnement des caméras ou la construction des cartes éléctroniques qui seront installées au centre de la ruche. Seulement, ces sujets s'éloignant de mon DUT Réseaux et Télécommunications et mes compétences et connaissances dans ces deux domaines étant limitées, je n'ai pas eu à travailler dessus.
Cependant, la possibilité d'un apprentissage lors d'école d'ingénieurs en systèmes embarqués a été évoquée, ce qui correspondrait plus à ces tâches.

## 1.3 L'équipe

Brouillon organigrame

CNRS : Jean-Baptiste THIBAUD 50% IBMM : Matthieu ROUSSET 50% LMGC : Delphine JULIEN Capucine CARLIER LIRMM : Jean TRIBOULET Sébastien DRUON IUT de Béziers : Philiphe PUJAS -¿ stockage des données Equipe Bee@Wur Université Wegeningen (Néerlandais) -¿ ouverture internationnalle.

# Chapter 2

# Projet principal

## 2.1 Analyse du besoin et des fonctionnalités exigées

Dans le cadre de la récolte et de l'analyse des données, il était primmordial pour l'équipe d'avoir l'outils d'annotation décrit plus haut.
Celui-ci devait permettre de :

- Visualiser les caméras en direct
- Démarrer la capture vidéo à tout moment
- Ajouter un système de tag par mots clés afin de pouvoir trier facilement les vidéos
- Avoir plusieurs types d'annotations (entourer, marquer, avoir des mouvements etc)
- Retrouver toutes les mesures de la ruches (température, horaires, numéro de caméra, de ruche, de cadre etc )
- Avoir un principe d'auteur
- Pouvoir revisualiser les vidéos
- Pouvoir remodifier les vidéos

Afin de répondre à ces besoins, nous avons imaginé l'interface suivante : Bien sûr, celle-ci a connu beaucoup d'évolutions au cours de son développement : au fur et à mesure de l'avancement, nous nous rendions compte de certaines éventualités que nous n'avions pas imaginé et que nous avons intégré à la volée.

## 2.2 Contraintes

Le langage de départ pour coder l'interface et ses fonctionnalités, que nous pourrons familiairement nommer partie moteur et partie physique, m'a été imposé.
C'est donc en C que j'ai dû réflechir à comment préparer et lier ces deux parties. Ce langage a été choisit tout simplement parceque c'est le langage que M. DRUON a pour habitute d'utiliser.
La partie moteur peut être développée en C sans trop d'ajout de librairies annexes autres que celles dites basiques (stdlib, stdio). Cependant, nous avons dû utiliser une librairie pour développer la partie physique. Nous avions le choix entre GTK ou QT. QT devant être utilisé en C++, notre choix s'est naturellement dirigé vers GTK 3.0.
Comme dans tout projets collaboratifs, la convergence des données est importante et peut parfois se réveler difficile à mettre en place. Heureusement pour nous, en programmation l'outil GIT est excellent pour travailler à plusieurs. L'ayant déjà vu en cours cette année, j'ai pu l'utiliser concrétement lors de mon stage. Cependant, avant de commencer à utiliser concrètement l'outil, il a été préférable que je me remette à niveau sur celui-ci et ai donc entammé la lecture du livre Mastering Git qui m'a été d'une grande aide pour avoir des bases solides. Pour héberger notre code, nous avons choisi de le déposer sur GitHub : il est donc accessible au public sous le nom superbeelive.

Pour cela, le livre m'a servi de référence de base quant à la manière de manier les éléments. Je me suis également aidée de divers tutoriels sur internet, mais surtout de la document officielle de GTK.
Souvent, on a tendance à utiliser Glade, un logiciel permettant de créer une interface GTK avec des outils graphiques. Dans mon cas, je m'en suis surtout servie en tant que bibliothèque pour voir et tester certains Widgets ainsi que leur fonctionnement. Cet utilitaire, certes plus rapide à prendre en mains, est lié au fait qu'il utilise des fichiers XML pour décrire l'interface ce qui, au final, complique la portabilité de l'interface d'une version du logiciel à l'autre. De plus, j'ai préféré apprendre à utiliser GTK en ligne de code "brut". Plus fastidieu au départ, j'ai trouvé cela plus simple à la longue : je maîtrisais vraiment mes outils et apprenais plus rapidement à utiliser un nouvel élément de la librairie.

### 2.2.1 Principe de GTK

Apprendre à utiliser GTK sera plus long si la programmation objet nous est inconnu. En effet, même si le C n'est pas un langage objet, GTK lui, reprend les principes de la programation objet. Si vous êtes déjà familier avec ce type de langage, alors l'apprentissage sera bien plus rapide. L'idée principale de GTK est que l'on va ranger tous les éléments en fonction d'un autre élément, pour finir avec des sortes de poupées russes qui s'emboîtent et se rangent côte à côte pour donner un résultat final. Il exite une multitude d'éléments utilisables, appelés les Widgets. L'élément de base widgets contient des propriétés de base et, pour créer d'autre type d'élément, les développeurs de GTK y ont ajoutés d'autres propriétés à celles déjà existantes. Au fur et à mesure, ces ajouts de propriétés ont permit de créer toute sorte de Widget : des textes, des labels, des tableaux, des

séparateur, des listes, des bouttons... Ces héritage et cette hierarchie des objets va nous construire un arbre d'élements, nous permettant de savoir ce que l'on peut faire avec nos Widgets : En effet, si par exemple je veux changer un paramètre d'un élément "GtkSpinButton", et que je ne trouve pas de fonction affectant directement ce type d'élément, alors je vais regarder si chez ses parents je peux trouver le paramètre voulu, à savoir "GtkEntry", et "GtkWidget".

Chaque paramètre disponible pour les Widgets peut être changé directement à l'aide de fonctions retrouvable dans l'excelente documentation en ligne de GTK 3.0. Il est totalement déconseiller de changer directement la valeur des paramètre à la mains en utilisant, par exemple des pointeurs. Il faut partir du principe que chaque élément a son nombre de fonctions associées et que tout est prévu pour pouvoir faire ce que l'on veut sur notre interface.

### 2.2.2 Des boites, dans des boites...

Une fois la manière de créer les éléments comprise, il faut ensuite comprendre comment les ranger. D'abord, on va créer une fenêtre, soit une GtkWindows. Dans celle-ci, on ne peut y poser qu'un seul Widget. C'est pour ça que nous avons les éléments "GtkContainers" : c'est avec eux que nous allons pouvoir structurer et placer tous les autres éléments dans notre fenêtre. Le container de base est la box. A sa création, nous devons simplement indiquer si celle-ci va être horizontale ou verticale, c'est à dire si les éléments que nous allons mettre dedans vont être placé de haut en bas ou de droite à gauche. Une fois cette première GtkBox placée, le problème de limitation de place imposée par la GtkWindow n'en est plus un. Maintenant, nous pouvons ranger ce que nous voulons dans cette boite. C'est là que le système de poupée russe prend son sens : nous allons devoir jouer avec les différentes boîtes pour créer les espaces que nous désirons. Voici le schéma de construction que j'ai fais pour la première fenêtre de l'application : On peut voir que la GtkBox box_principale prend toute la place de ma fenêtre GtkWindow. Dans cette box_principale, j'y ai rangé verticalement box_up et box_down. Dans box_down j'ai integré box_left et box_right horizontalement, me permettant d'avoir quatre zones délimitées. Ensuite, j'ai créé des box pour mes éléments plus précis : dans box left j'ai directement intégré box_video dans laquelle se trouve la vidéo. Dans box right on va retrouvé box_info, box_meta, box_btn_cam, box_btn_video, box_info_time et box_file. Chacune de ces box m'ont permit de ranger à l'intérieur les éléments que je voulais retrouver : les boutons, les textes, les descriptions etc. A noter qu'en plus de ces boîtes j'ai également parfois ranger des séparateur, les barre horizontale et verticale, permettant d'ajouter un peu de structure visuelement.
Une fois que l'on sait comment ranger les éléments sur une fenêtre et quel est le principe de ces éléments dans l'idée, il ne manque plus qu'à voir comment concretement, dans le code, on applique ces principes.
GTK nécessite une organisation rigoureuse si on veut pouvoir s'y retrouver et imag-

iner facilement comment est construit la fenêtre décrite dans le code. C'est pourquoi j'ai préféré appliquer une manière de classer les éléments qui me permette de retrouver facilement ce avec quoi je veux travailler dans l'instant.
Ainsi, j'ai voulu identifier les différentes étapes de la création d'un Widget et les séparer en trois grandes parties pour m'y retrouver plus facilement.

### 2.2.3 Création et Définition

La première partie est la déclaration d'un Widget et sa définition. La création est simplement la déclaration de la variable avec son type. A noter que j'utilise des pointeurs pour toute mon application et que mes exemples viennent directement des fichiers win_main.c et win_main.h. La structure et l'explication de ces pointeurs est décrite dans la partie "Les structures" de "Définition des fonctionnalités".
Créons une fenêtre, une box et un label :
–¿ Dans win_main.h : GtkWidget* window ;
GtkWidget* box ;
GtkWidget* label ;
Pour ces trois exemples, ces éléments sont tous du type GtkWidget. Il arrive que, dans de rare cas, il soit nécessaire de déclarer un autre type d'objet.

Ces trois éléments existent désormais, maintenant il faut les définir, c'est à dire expliquer de quel type de Widget il seront et leur donner leur propriétés.
–¿ Dans win.c : tmp-¿window = gtk_window_new (GTK_WINDOW_TOPLEVEL) ; Ici, on utilise la fonction "gtk_window_new" pour indiquer que "window" est une fenêtre qui s'ouvre en premier plan avec l'argument "GTK_WINDOW_TOPLEVEL".

tmp-¿box_principal = gtk_box_new( GTK_ORIENTATION_VERTICAL, 0) ; Comme dit plus haut, à la création d'une box on indique d'abord si on veut que les éléments soient posés verticalement ou horizontalement. Ensuite, le second argument permet de définir le nombre de pixel d'écart par défaut entre les éléments, ici 0px, donc ils seront collés l'un à l'autre.
tmp-¿label = gtk_label_new("Je suis le texte du label") ;
Enfin, le label nécessite lui aussi un argument, ici ce sera le texte noté de base. On peut laisser ce texte vide si on ne rempli pas les "".

Les exemples ci-dessus ont tous eu besoin d'une définition précise du Widget, avec à chaque fois au moins un paramètre à régler. Il arrive parfoit que la création ne nécéssite aucun paramètre, et que des modification doivent être apportées ensuite. Par exemple, si je veux que mon label ai une écriture plus stylisée, je peux utiliser :
gtk_label_set_markup(GTK_LABEL(tmp-¿label), "¡span foreground=b̈lackf̈ont=ï0ï¿¡b¿Je suis le label stylisée avec du bold¡/b¿¡/span¿");
Ici, il faut préciser avant le nom de la variable à modifier quel type de widget on compte modifier. En effet, comme lors de la déclaration il est indiqué que label est de type

GtkWidget et que la fonction que l'on veut utiliser ne s'applique que sur un GtkLabel, il est donc nécessaire d'indiquer que cette fois-ci, c'est de type un GtkLabel depuis la fonction plus haut à l'aide de "GTK_LABEL". Cette fois ci, mon label ressemblera à ça :
Ainsi, il existe plein de fonctions différentes permettant de régler ce genre de paramètres sur les widgets, et parfois il est nécessaire de faire appel à elle pour avoir un Widget correct. Il est intéressant d'explorer les fonctions existantes afin de personnaliser au mieux notre interface.

Une fois les Widgets créés, il faut maintenant les placer et indiquer qu'il faut les afficher.

### 2.2.4 Placement, cosmétique et affichage

Maintenant, nous allons appliquer la théorie qui avait été développée au dessus : les poupées russes. On va commencer par dire où va se placer la box :
gtk_container_add (GTK_CONTAINER (tmp-¿window), tmp-¿box) ;
Window est un GtkContainer, nous utilisons donc la fonction qui permet d'affecter un élément à un container. Le premier argument indique le container où nous devons insérer un élément. Le second nous indique quel élément est à placer.

Ensuite, nous faisons de même pour la box :
gtk_box_pack_start(GTK_BOX(tmp-¿box_principal), tmp-¿label, FALSE, FALSE, 0) ;
Box étant de type GtkBox, nous utilisons cette fois la fonction qui lui est associée. Comme pour le Window, il faut indiquer la boîte concernée par l'élément à ajouter, mais cette fois-ci trois autres arguments sont à renseigner concernant la manière dont les Widgets vont se comporter à l'intérieur de la box. Mais d'abord, il faut savoir que par défaut (donc que tout est sur FALSE), les box allouent la même place à tous les Widgets qu'ils contiennent sans déborder sur le reste.

- Expand : peut être TRUE ou FALSE. Permet aux widgets de prendre plus de place si besoin si réglé sur TRUE.

- Fill : peut être TRUE ou FALSE. Ne peut être actif que si Expand est aussi TRUE. Alloue la hauteur ou la largeur totale de la box, celon si la box est une horizontale ou vertical.

- Padding : valeur numérique. Indique l'espacement en pixel (px) entre les widget que contient la box.

Les éléments sont maintenant placés. Afin d'avoir une relecture facile et pour savoir qui se trouve dans quelle box ou container, j'ai appliqué une indentation qui m'est propre mais que je trouve plus lisible, voici l'exemple pour win_main :
Ainsi, à chaque fois que je vais dans une sous box, j'indente avec une tabulation. Au final, le rendu me permet de voir qui est à quel niveau.

Une fois que les éléments sont placés dans les boîtes, ceux-ci ne sont par contre pas arrangés précisements. Il faut maintenant indiquer où il faut appliquer des marges, comment doivent se placer les éléments. Sachant que la première chose à faire pour cette étape est de jouer avec les paramètres EXPAND et FILL des box qui permettent, par exemple, d'autoriser ou non à un élément de s'agrandir en même temps que la fenêtre.
Voici un exemple de ce que donne une fenêtre sans la partie cosmétique et arrangement précis puis avec :
Comme pour la création et le placement, nous allons utiliser des fonctions pour indiquer ce que l'on veut faire avec le Widget. Celles-ci sont le plus souvent utilisées sur les éléments considérés en tant que Widget. Donc, comme plus haut, il faut préciser que nous l'appliquons sur un Widget. En voici quelques exemples :

gtk_widget_set_margin_top (GTK_WIDGET (tmp-¿label), 5 ) ;
Permet d'appliquer une marge sur le haut. On précise sur quel Widget l'appliquer puis la taille en px de la marge.
Cette fonction existe aussi pour bottom, end et start qui correspondent à la droite et la gauche du Widget.

gtk_widget_set_halign ( GTK_WIDGET (tmp-¿label), GTK_ALIGN_START ) ;
Les éléments se mettent par défaut au centre de leur place disponible. Cette fonction permet de forcer un Widtget à se placer autrement : ici, on force l'alignement horizontal (halign) à être en début de zone, soit à gauche (START).

On peut aussi retrouver des fonctions pour préciser le comportement de la fenêtre, comme :
gtk_window_set_title (GTK_WINDOW (tmp-¿window), "BEETERFACE") ;
qui permet de nommer la fenêtre "BEETERFACE".

gtk_window_set_default_size ( GTK_WINDOW (tmp-¿window), 30, 30 ) ;
Définit la taille minimale de la fenêtre, sachant qu'elle prend en compte la place nécessaire aux éléments pour pouvoir être placés et que si la taille indiquée est trop petite, alors la fenêtre prendra la place minimale possible pour afficher les éléments.

Avec ces outils et un peu de recherches sur la documentation en ligne, on peut maintenant créer n'importe qu'elle fenêtre comme on le souhaite.

Attention, jusqu'à nous avons juste définit l'interface. Il ne reste plus qu'une étape simple pour que celle-ci puisse être visible : l'affichage. Si on n'indique pas qu'il faut

afficher les éléments, alors notre interface sera vide.
Personnelement, j'aime la faire en deux étapes. D'abord l'affichage de la fenêtre avec :
gtk_widget_show (tmp-¿window) ;
Puis de la box principale avec tout ce qu'elle contient :
gtk_widget_show_all (tmp-¿box_principal) ;
Sachant que l'affichage se fera dans l'ordre annoncé, si vous avez peur que le logiciel apparaisse "étape par étape", c'est à dire avec les éléments apparaissant au fur et à mesure et une fenêtre vide restant affichées quelques secondes, préférez mettre l'affichage de la box principale avant l'affichage de la fenêtre : ainsi, la box et ses enfants sera déjà générée et n'aura besoin que de son support.

Il ne reste plus qu'à lier des fonctions aux différents éléments pour que notre application soit fonctionnelle.

### 2.2.5 Signaux et fonctions

Avoir des boutons, c'est bien. Pouvoir les utiliser, c'est encore mieux. Maintenant que tous éléments sont placés, il ne reste plus qu'à les lier aux fonctions qui permettrons de réaliser diverses tâches (mettre en pause la vidéo, se déplacer dans le temps de celle-ci, débuter la sauvegarde etc). C'est ce qui va établir le liens entre le moteur et la carcasse. Pour le moment, peu de fonctions ont été écrites pour le moteur : toute la partie graphique est prête à les acceuillir mais le développement des fonctionnalités est ce qui prend le plus de temps. En guise d'entraînement et pour pouvoir être prête le jour où il faudra faire tous les liens, j'ai déjà fait quelques tests pour comprendre les principes. Les liens entre l'interface et les fonctions vont se passer dans le main. Dans notre code, il est présent dans beeterface.c.
Avant tout, il faut noter que quelques fonctions de bases sont obligatoires pour pouvoir lancer notre interface, voici le modèle de base :

```
int main(int argc, char *argv[])
{
gtk_init(argc, argv);
    votre code

gtk_main();
return 0 ;
}
```

Les deux fonctions permettent d'initialiser l'interface et d'indiquer que celle-ci est en attente de signal de la part de l'utilisateur pour réagir.
Maintenant que nous avons la base, commençons à intégrer des évenements.
Le but va être d'appeler une fonction qui va lier un éléments à une fonction et d'indiquer en quel circonstance la fonction doit réagir. Voici ladie fonction :

```
g_signal_connect( queen-¿interface-¿win_main-¿btn
, "clicked",
```

G_CALLBACK(callback_btn)
, queen ) ;
Le premier argument correspond à l'élément qui est visé. Dans notre cas, beaucoup de pointeur sont en jeu, pour le moment nous allons passer outre ces pointeurs : ils seront expliqués dans la partie structure. Partons simplement du principe qu'il faut appeler le bouton concerné.
Le second argument sert à indiquer à quel évenement il faut réagir. Il en existe une liste prédéfinie : ici c'est l'événement "clicked". Lorsqu'on cliquera sur le bouton, la connection sera établie.
Le troisième argument correspond à la "callback" appelée, soit la fonction qui doit se déclancher. Nous allons y revenir.
Le dernier argument indique ce qu'on envoit au callback comme information. Ici, c'est "queen", une sorte de gros paquet. Comme pour les pointeurs, ce sera expliqué plus tard.

Allons vois la callbak qui est appellée :

```
void callback_quit_tag(GtkWidget* widget, gpointer data) {
queen_t* tmp ;
tmp = data ;
gtk_window_close (GTK_WINDOW(tmp-¿interface-¿win_main-¿window)) ;
}
```

Cette callback contient le strict minimum pour faire fonctionner simplement un bouton.
En arguments, elle reçoit d'abord le widget qui est à la source de son appel : dans notre cas, c'est "btn".
Ensuite, elle reçoit les data que "g_signal_connect" a envoyé, c'est à dire "queen".
Les deux premières lignes indiquent que la variable tmp reçoit les informations de data.
Enfin, la fonction utilisée permet d'indiquer qu'on va fermer une fenêtre, ici celle de win_main, donc là où on se trouve. Cet exemple est basique, on peut évidement faire bien plus et toucher à l'interface en elle même. A vous de voir ce que vous désirez faire.

Tous les outils principaux pour pouvoir créer une interface GTK ont été présentés. Avec ceux là, beaucoup de recherches dans la documentation et le "catalogue" de GTK, on a en mains une infinité de possibilité. Certaines actions graphiques nécessitent plus de reflexion que d'autre, mais au final ça n'est que de l'algorithmie ; toute la syntaxe est là.

## 2.3 Les fonctionnalités

### 2.3.1 Structuration du code

La première chose que j'ai eu à faire pour la création de l'application a été de réfléchir à comment celle-ci allait fonctionner. En étudiant la demande, une chose majeure en ressortait : il y allait avoir beaucoup de données à traiter et sauvegarder.
Si on lit la liste des besoins présentés dans la partie on peut voir qu'il y a une notion d'auteur, de caméra, de vidéo et de ruche qui sont présentées. Afin de pouvoir manipuler de telles informations, j'ai créé une structure pour chacun de ces éléments qui contiendront des informations primmoridales : dates, noms, numéro de téléphone, lieu, référence etc.
Pour chacune des structures, j'ai également préparé des fonctions permettant de manipuler les données. En général, on y retrouve une fonction pour créer un élément, une pour le supprimer et un jeu de fonction pour modifier ou lire toutes les données contenues dans l'élément indépendemment. Cette préparation permettra de pouvoir utiliser ces données sans avoir à se soucier des pointeurs : tout est prêt, il n'y a plus qu'à choisir comment on voudra modifier les données lors de l'utilisation du logiciel.
Cette préparation a aussi été faite pour toutes les annotations que nous pourrons intégrer sur les vidéos : les cercles, flèches, crois...
Pour le moment, toutes ces annotations ne sont pas prête. Mais les plus basiques, comme la croix, a déjà sa structure de préparée avec un x et un y permettant d'indiquer où elle devra se placer.
Ce travail de préparation m'aura prit environ une semaines. Il a été difficile seulement au début : il fallait réfléchir à comment créer une structure correctement, penser à bien utiliser les pointeurs et gérer leur allocation de mémoire. Une fois un premier modèle fait, les autres se faisaient de plus en plus vite. Utilisant l'outil VIM, j'ai énormément utilisé les "rechercher / remplacer" pour travailler.
Ce travail, bien que fastidieux, m'aura permit d'appliquer concrétement l'utilisation des pointeurs, ce qui me manquait lors de mon arrivée en stage.

### 2.3.2 Manière de coder

N'ayant pas d'habitudes particulières en code, j'ai été conseillée quant à la structure de mon code, afin que celui ci soit lisible et bien réparti. Ainsi, j'ai utilisé des couples de fichiers .c et .h me permettant de séparer la déclaration d'une structure et des fonctions ainsi que la définition de celles-ci.
De plus, nous avons rapidement intégré un Makefile afin de nous éviter de compiler manuellement à chaque fois : avec Git, ce sont des outils qui me reserviront régulièrement dans d'autres contextes désormais.
Une fois cette façon de faire mise en place pour la partie moteur, j'ai voulu l'appliquer à la partie graphique. Ainsi, un couple de .c et .h a aussi été créé pour chacunes des fenêtres, le .h contenant toutes les déclarations d'éléments sous forme de structure.

Dans le .c on retrouve la fonction de création de l'interface avec au début la définition de "tmp" qui correspond donc au "papier cadeau" contenant toutes les déclarations des éléments. C'est pourquoi nous avons vu sur les exemples plus haut que pour appeler un élément nous devions utiliser un pointeur comme "tmp-¿label".
Notre fichier .c contient alors de quoi créer l'interface, l'afficher et la supprimer, la suppression revenant à appliquer les Free nécessaires. C'est pour cela que dans notre main on retrouve ces fonctions et que celui-ci est allégé.

Jusque là, cette technique m'a permit d'avoir une structure de code agréable et ordonnée. Seulement, c'est arrivé au moment de faire les premier liens interface / fonctions qu'un autre problème s'est posé : comment pouvoir accéder à toutes les données dont j'ai besoin lorsque j'utilise une callback ? Revenons à la manière de

### 2.3.3 Représentation du projet

### 2.3.4 Traitement de la vidéo

# Chapter 3

# Acquis et compétences

## 3.1 Missions annexes

## 3.2 Compétences développées

# Chapter 4

# Conclusion

# Chapter 5

# Références

## 5.1 Annexes

## 5.2 Biblio

## 5.3 Lexique