

PUNYDUCK

UNE PLATEFORME DE DISTRIBUTION DE PROJETS EN LIGNE

RÉALISÉ PAR :



FVOSTUDIO

RAPPORT DE PROJET T.E.R. PROJET INFORMATIQUE - HLIN405

Étudiants :

Valentin FONTAINE
Paul BUNEL
Esteban BARON

Valentin PERON
Julien LEBARON

Encadrante : M^{me} Anne-Elisabeth BAERT

Année : 2019-2020



Table des matières

Introduction	2
1 Technologies utilisées	4
1.1 Langages	4
1.2 Outils	5
2 Conception de Punyduck	6
2.1 L'architecture Modèle-Vue-Contrôleur	6
2.2 Développement de Litequarks, le framework	7
2.3 Création de l'application	12
2.4 Communication réseau et base de données	13
2.5 Fonctionnalités de l'interface	14
2.6 Statistiques	15
3 Présentation d'algorithmes	16
3.1 Fonction de position du curseur	16
3.2 Fonction de gestion des requêtes SQL	18
4 Gestion du Projet	20
4.1 Organisation et planification	20
4.2 Changements majeurs	21
5 Bilan et Perspectives	23
Annexes	25
A Visuels de l'application	25
B Principaux algorithmes	28
B.1 Fonction cursor_position_callback	28
B.2 Fonction SQL	29

Introduction

Dans le cadre du TER de notre deuxième année à la faculté des sciences de Montpellier nous avons proposé un projet s'intitulant PunyDuck. Le but de ce projet est la réalisation d'une plate-forme de distribution des projets des étudiants du département informatique.

Le groupe de développement est composé de cinq personnes, Valentin FONTAINE, Paul BUNEL, Valentin PERON, Julien LEBARON et Esteban BARON. Nous sommes encadré par Mme Anne-Elisabeth BAERT.

Motivation

Le TER est un module qui apporte beaucoup aux étudiants en gestion de projet ainsi qu'en programmation. Seulement une fois terminés les projets ne sont pas valorisés et tombent dans l'oubli. Notre solution est de proposer une application permettant à chaque étudiants de déposer leurs projets pour les rendre visibles et téléchargeables par tous.

Approches

Notre objectif final étant de produire une application fonctionnelle pouvant être utilisée par tous, nous avons besoin : d'une interface graphique décente, totalement opérationnelle et facilement modulable, ainsi que d'un système de communication efficace entre l'application et la base de données. Nous avons donc décidé de diviser le projet en trois parties : tout d'abord la conception d'un framework permettant la réalisation d'interfaces graphiques, puis la réalisation de l'interface graphique elle-même à partir de ce framework, et enfin la partie réseau permettant à l'application de communiquer avec un serveur distant. Ces trois parties communiquent ensemble par le biais d'une architecture logicielle Modèle-Vue-Contrôleur (MVC).

Notre application est téléchargeable par tous depuis le site web <https://fvostudio.com/punyduck/>.

Cahier des charges

Le but de Punyduck est de proposer aux étudiants en informatique une plateforme de distribution de leurs différents projets. Cette plateforme doit être une application téléchargeable sur un site internet, et connectée à un serveur pour permettre l'échange de projets entre différents

clients.

Pour l'organisation du projet, nous l'avons tout d'abord divisé en plusieurs étapes :

- Mise en place d'un serveur qui servira d'intermédiaire entre les utilisateurs et la base de données.
- Création d'un framework pour faciliter la réalisation de l'application.
- Conception de l'application graphique à l'aide du framework.
- Connexion entre l'application graphique et le serveur.
- Création d'une base de données pour stocker les comptes des utilisateurs et les projets.
- Mise en service d'un site internet permettant le téléchargement de l'application.

Ensuite, une fois les différentes parties du projet dégagées, pour bien définir chacune d'entre elles nous avons décidé du cahier des charges suivant :

Le serveur : Le serveur devra fonctionner de manière asynchrone (nous définiront ce principe dans la partie 1.1). Il pourra héberger de manière sécurisée les données des utilisateurs (nom, mots de passes, ...) et devra être capable de gérer la plupart des erreurs de réseau, comme les coupures de connexion lors d'un téléchargement.

Le framework : Le framework définira un ensemble de classes et de fonctions qui serviront de base à la structure d'une nouvelle application. Son utilisation doit être facile avec assez de fonctionnalités déjà disponibles pour pouvoir réduire les appels de fonctions bas niveau. De plus, il devra également permettre la plasticité des interfaces.

L'application : Notre application présentera une interface graphique permettant de naviguer facilement entre différents onglets, de communiquer avec le serveur, et de personnaliser un minimum l'apparence : mode clair / sombre, position de certains éléments de la fenêtre, couleur de l'arrière plan.

La base de données : Elle stockera les données relatives aux comptes des utilisateurs ainsi que les projets. Les données confidentielles (mots de passe) seront cryptées.

Le site : Il sera constitué d'une unique page permettant de télécharger l'application. Il possèdera un design « responsive » et dynamique.

Partie 1

Technologies utilisées

1.1 Langages

Pour la programmation de notre application, nous avons choisis assez naturellement d'utiliser le langage C/C++. En effet, c'est le langage que nous avons le plus étudié à l'université, avec lequel 3 des 5 membres du groupe avaient codé leur projet de Licence 1 CMI, et qui nous offrait un support puissant et efficace pour notre interface graphique.

De plus le C++ nous a permis de coder en orienté objet, ce qui était une nécessité pour la création de notre application. En effet, notre programme utilise énormément ce type de programmation : nous développeront ce point plus tard dans la partie 2 sur la conception de l'application .

Pour l'interface graphique, nous avons utilisé la bibliothèque OpenGL : une interface regroupant environ 250 fonctions différentes qui peuvent être utilisées pour afficher des scènes tridimensionnelles complexes à partir de simples primitives géométriques. Du fait de son ouverture, de sa souplesse d'utilisation et de sa disponibilité sur toutes les plates-formes, OpenGL est une des bibliothèque graphiques les plus utilisée par la majorité des applications scientifiques, industrielles ou artistiques 3D.

L'utilisation de cette bibliothèque (associée à GLFW pour la gestion des fenêtres) était donc un choix assez basique puisqu'elle est très répandue, mais nous avons dû apprendre à l'utiliser : celle-ci est beaucoup plus complexe que les librairies que nous avons utilisées auparavant, comme SFML ou PyGame, de par son bas niveau.

Pour gérer la partie réseau de notre application, nous n'avions pas besoin d'utiliser un langage puissant comme le C/C++, nous pouvions donc nous rabattre sur un langage moins performant mais plus facile d'accès. De ce fait, nous nous sommes dirigés vers le langage python, qui est un langage très simple d'utilisation, pratique pour débiter dans le domaine complexe de la programmation que représente la programmation en réseau. De surcroît il est plutôt efficace pour gérer les entrées sorties, notamment pour la lecture et l'écriture dans des fichiers, qui est une notion centrale dans notre projet.

Cependant, le langage Python de base ne permet pas de faire de la programmation en réseau, nous avons donc du choisir une des nombreuses bibliothèques disponibles permettant de faire de la programmation réseau en Python. Après réflexion, nous avons opté pour le module « `asyncio` »

pour deux raisons : premièrement, ce module offrait une interface de programmation en réseau de haut niveau donc plus simple d'utilisation, ce qui nous arrangeait particulièrement étant donné que nous sommes parfaitement novices dans ce domaine ; Deuxièmement, le gros point fort de ce module est le fait qu'il implémente une nouvelle manière de programmer : la programmation asynchrone. La programmation asynchrone pour les entrées/sorties est une forme de programmation parallèle permettant d'exécuter d'autres parties d'un programme lorsque celui-ci est en attente d'une transmission de donnée, afin de grandement diminuer le temps d'exécution du programme.

1.2 Outils

Pour réaliser notre projet nous avons utilisé des outils différents et spécifiques à chaque tâche.

Nous avons utilisé le système de gestion PostgreSQL¹ pour gérer la base de données de notre serveur, ce système nous permettant d'utiliser des requêtes SQL en Python et ainsi de mettre en application les connaissances acquises cette année avec le module HLIN304.

Pour pouvoir exécuter des requêtes SQL depuis le Python, nous avons employé le module psycopg2²

Pour la partie modélisation de l'application nous avons opté pour le Langage de Modélisation Unifié (UML) vu en cours, dans le module HLIN406.

Toutes les communications du groupe se sont faites sur le logiciel Discord³, un logiciel facilitant grandement les communications en groupes avec par exemple le partage d'écran, les groupes vocaux et textuels, le fait de pouvoir partager des morceaux de code directement dans le canal de discussion textuel etc..

Pour ce qui est du partage et des sauvegardes du code, nous avons utilisé Git (un logiciel de gestion de versions décentralisé) via un serveur GitHub⁴ qui nous a permis de garder nos anciennes versions, de ne rien perdre en cours de route, et de pouvoir partager l'avancée du projet avec les autres étudiants du groupes et notre encadrante.

Pour ce qui est de l'éditeur de code utilisé, nous nous sommes tous penché sur Visual Studio Code⁵ pour sa fiabilité et sa mise en page agréable. De plus, nous avons pu, grâce à ce logiciel, coder à plusieurs en même temps avec sa fonctionnalité de partage en temps réel.

Enfin, nous avons utilisé le langage L^AT_EX via la plateforme Overleaf⁶ pour rédiger ce rapport.

1. PSQL : <https://www.postgresql.org/>

2. psycopg2 : <https://www.psycopg.org/>

3. Discord : <https://discordapp.com/>

4. GitHub : <https://github.com/>, notre projet : <https://github.com/valfvo/punydock>

5. VSCode : <https://code.visualstudio.com/>

6. Overleaf : <https://www.overleaf.com/>

Partie 2

Conception de Punyduck

2.1 L'architecture Modèle-Vue-Contrôleur

Nous avons commencé par séparer le projet en trois grandes parties avant d'entamer la conception :

- Le framework pour structurer l'application
- L'application utilisant le framework
- La communication réseau entre l'application et le serveur.

Par la suite nous avons opté pour une architecture *Modèle-Vue-Contrôleur* (ou simplement *MVC*) pour lier les différentes composantes du projet. En effet ce type d'architecture est plutôt populaire dans le développement d'interfaces graphiques car il permet une structure robuste et flexible avec une composition en trois modules :

- **Le modèle**, qui contient toutes les données et la logique pour manipuler ces dernières. Il représente une petite base de données et n'a pas à se préoccuper de savoir comment afficher ces données.
- **La vue**, qui contient la représentation graphique des données du modèle sans avoir besoin de les détenir. Elle est une sorte de schéma que l'on peut ressortir dès que l'on veut afficher une donnée.
- **Le contrôleur**, qui contient la logique pour interagir avec l'utilisateur et sert d'intermédiaire entre le modèle et la vue. Il constitue le « cerveau » de l'application.

Dans une architecture *MVC*, chaque module est indépendant et n'a qu'une fonction spécifique. Cela facilite énormément le développement car on peut se concentrer sur une seule tâche sans se soucier du reste. Pour implémenter le patron *MVC* à notre application nous avons décidé de faire du framework le contrôleur et le modèle. Le framework sert également de base à la vue, qui est naturellement l'interface graphique. La partie réseau en Python sert à communiquer avec le serveur pour mettre à jour le modèle.

2.2 Développement de Litequarks, le framework

2.2.1 Début de la conception

L'écrasante majorité du développement de notre application a été consacré à la réalisation du framework, qui représente environ 75 % du projet. La conception d'un tel framework a été particulièrement difficile. Effectivement, l'objectif est de créer un outil que nous allons nous-même utiliser pour faciliter la mise en place de l'interface graphique. Il ne doit donc pas seulement marcher à peu près, il doit marcher à tout les coups dans toute les situations, en plus d'être pratique et facile à prendre en main. La complexité de réalisation est à un tout autre niveau d'un simple exercice où l'on peut ne pas réussir, ici l'échec n'est pas une option.

La première étape fut logiquement la conception, qui précède l'implémentation. C'est l'étape la plus difficile et la plus délicate. En effet si la conception n'est pas bonne, c'est toute l'implémentation et l'utilisation du framework qui seront pénalisés. Pour être sûr d'avoir une conception correcte nous avons décidé de consacrer la moitié du temps qui nous a été donné pour la réalisation du projet à la conception du framework, à savoir un mois et demi sur un total de trois mois.

Avant même de parler d'interface graphique, nous avons tenté d'imaginer une structure adéquate et flexible qui sera facilement utilisable. Cependant « simple à utiliser » ne rime pas avec « simple à concevoir » car chacun a sa propre logique. Notre tentative nous a donc mené à l'élaboration des **LQuark**, qui permettent la création de *structures hiérarchisées hétérogènes*. Une très grosse partie du framework repose sur cette structure, et dont il tire son nom (Litequarks, ou le diminutif LQ).

2.2.2 Des LQuark à l'interface graphique

Les **LQuark** sont le coeur du framework. Ils sont extrêmement simples mais permettent la création de structures très organisées grâce à un système de hiérarchie. Chaque quark possède un parent et plusieurs enfants sans limite de nombre ou éventuellement aucun. Les enfants de mêmes parents ont également des liens vers l'enfant précédant et l'enfant suivant. Enfin chaque quark a la possibilité de rajouter ou supprimer des enfants. Cette structure est comparable à un arbre n-aire où chaque noeud a un lien vers 2 noeuds qui ont le même noeud parent.

Une fois la structure des quarks terminée, nous nous sommes attelés à l'apprentissage de la bibliothèque graphique que nous avons utilisé, OpenGL. La prise en main fut assez longue et complexe. Effectivement avec OpenGL nous ne pouvons dessiner que de simples primitives graphiques, comme des traits ou des triangles. Il faut par exemple 2 triangles pour dessiner un rectangle. Pour simplifier cette utilisation peu commode, nous avons créé 2 classes :

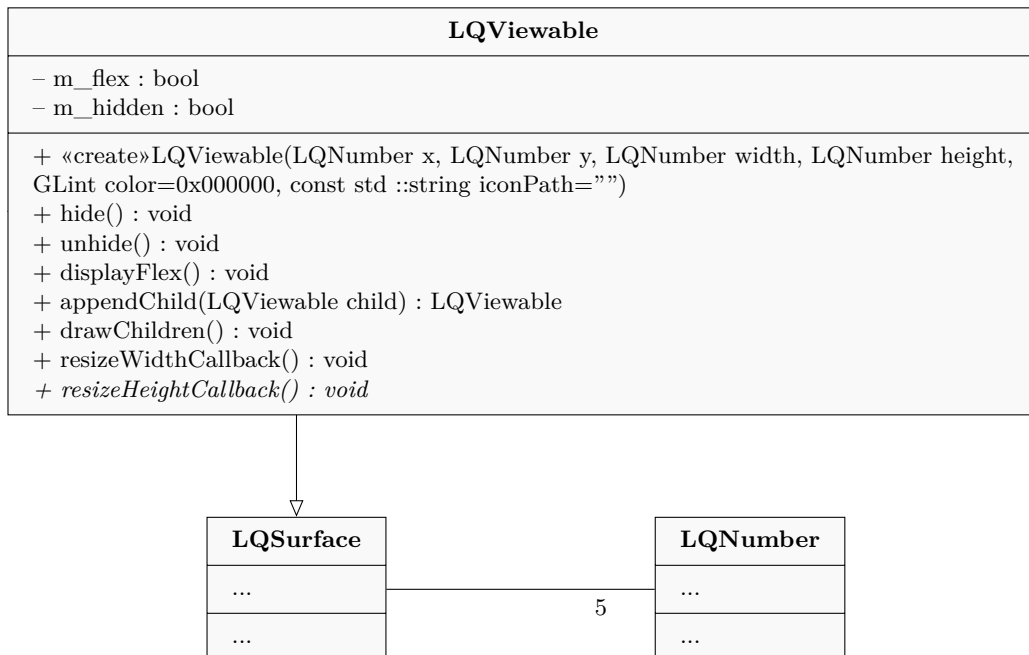
- **LQTexture**, qui représente les pixels d'un rectangle ou d'une image en mémoire. Elle encapsule également toutes les données nécessaire à OpenGL pour dessiner ces pixels à l'écran.
- **LQSurface**, qui représente une texture dans un espace en 2D. Une surface possède donc des coordonnées x et y , ainsi que différentes méthodes pour déplacer ces coordonnées et redimensionner la texture sous-jacente à la surface.

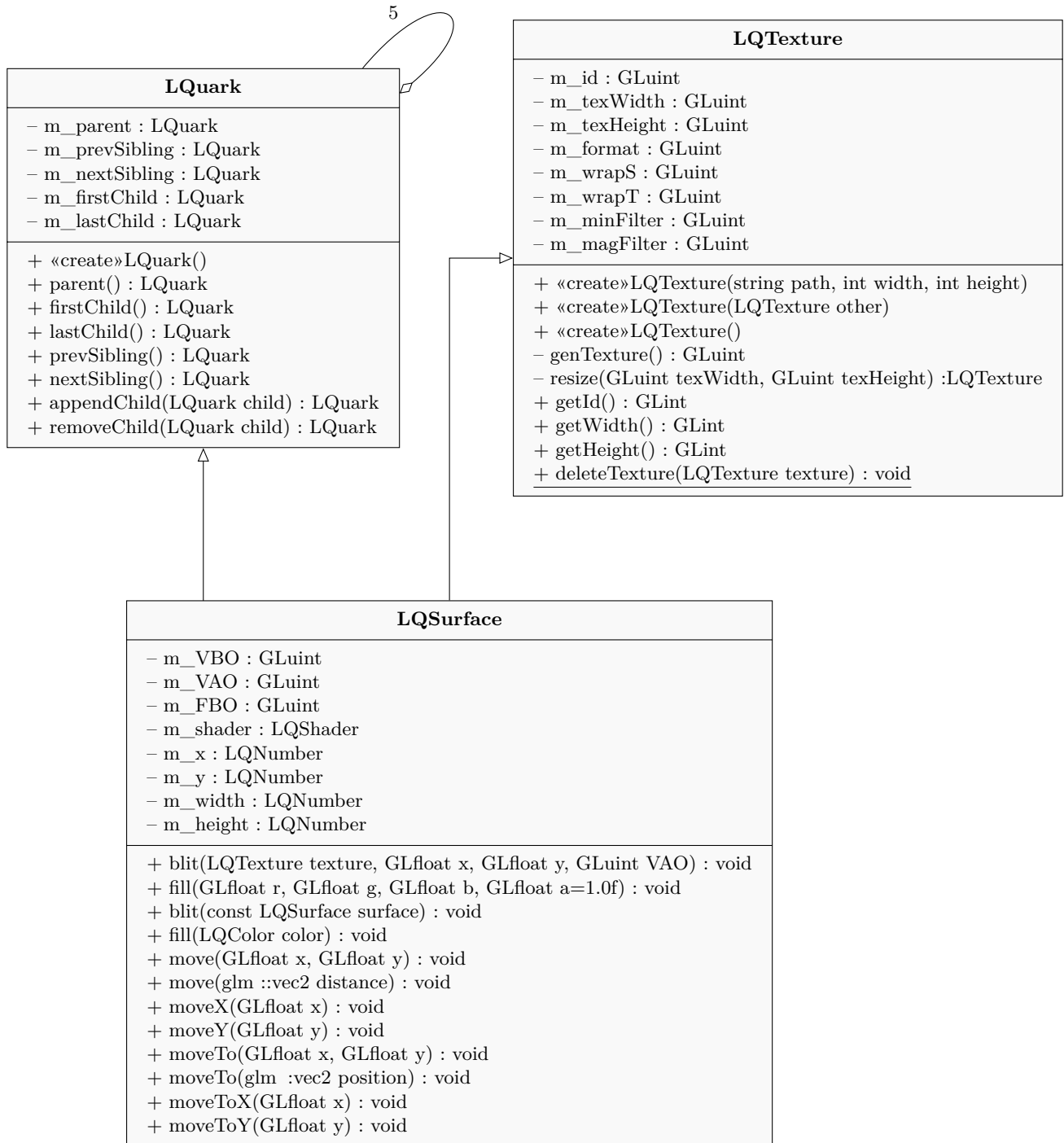
La classe **LQSurface** hérite de **LQTexture** mais aussi de **LQuark**. C'est donc un autre pivot du framework car elle regroupe la praticité des quarks et l'utilisation facilitée d'OpenGL par les textures. En combinant ainsi ces classes nous obtenons une structure particulièrement intéressante car nous pouvons désormais dessiner une surface sur autre surface très simplement. En effet grâce aux quarks chaque surface a des enfants et chaque enfant peut avoir des coordonnées relatives à son parent.

Par exemple, si une surface parent est en (100, 100) et qu'elle a un enfant dont les coordonnées sont (50, 50), l'enfant aura une position absolue de (150, 150). Pouvoir placer ainsi des surfaces relativement à leur parent rend le placement d'éléments graphiques extrêmement simple et intuitif.

Pour compléter les surfaces, nous avons également implémenté une autre classe, **LQViewable**. Assez légère, cette classe permet pourtant de faciliter davantage la création d'interfaces graphiques grâce à deux mécanismes. Le premier est la possibilité de « cacher » un **LQViewable**, c'est-à-dire qu'il est présent dans la structure des composants visuels mais n'est pas dessiné. Cela s'avère très pratique notamment pour afficher un message pour l'utilisateur, comme « votre projet est bien envoyé » ou encore « veuillez remplir tout les champs ». Le second mécanisme est la capacité d'un **LQViewable** à être flexible. Un élément flexible est un élément qui se redimensionne de lui-même pour toujours contenir tout ses fils. Ainsi suivant si un enfant est ajouté ou supprimé, il va s'agrandir ou se rétrécir.

Vous retrouverez ci-dessous les diagrammes UML des différentes classes précédemment introduites.





2.2.3 Les LQNumber, des nombres dépendants d'autres

Une autre clé de voûte du framework sont les **LQNumber**. Ils permettent grâce à un système de trois classes d'avoir des nombres qui dépendent d'autres. Un tel système est plutôt pratique pour les quarks, car nous pouvons désormais créer des éléments dont la taille dépend de ses parents ou de ses voisins. C'est avec les **LQNumber** que nous avons implémenté le *responsive design* de notre application, qui permet aux éléments graphiques de s'adapter à n'importe quelle taille de fenêtre. Vous pouvez retrouver en annexe A.2 et A.4 des visuels montrant ce système. Les trois classes le composant sont :

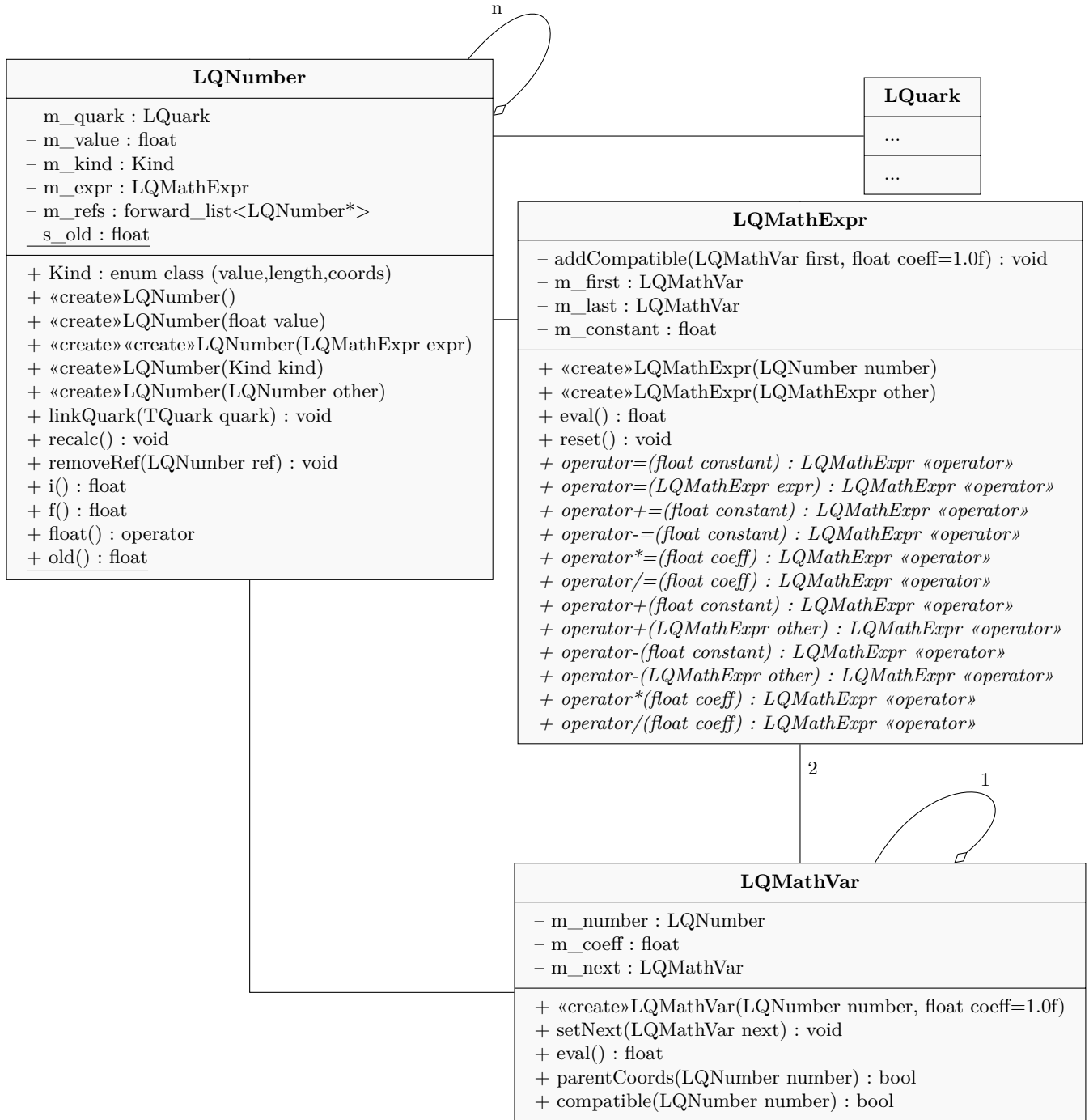
- **LQNumber**, qui est un nombre dépendant d'une expression. Il possède aussi une liste des nombres qui dépendent de lui. À chaque fois qu'un nombre est mis à jour, chaque nombre de sa liste est notifié et va alors recalculer sa nouvelle valeur en fonction de son expression.
- **LQMathExpr**, qui est une suite de variables avec une constante. Une expression possède par ailleurs des méthodes arithmétiques pour additionner, soustraire, multiplier ou diviser un expression par une autre expression ou une constante.
- **LQMathVar**, qui est simplement un **LQNumber** couplé à un coefficient positif ou négatif.

Les diagrammes UML des **LQNumber** sont présentés à la page suivante.

2.2.4 Fin du framework

Pour finaliser le framework nous avons créé plusieurs autres classes que nous avons utilisés pour composer l'interface graphique de notre application. Elles reposent toutes sur les **LQViewable** : **LQText** pour afficher du texte, **LQTextArea** pour entrer du texte dans un champ, **LQButton** pour créer des boutons cliquables, **LQViewport** et **LQDocument** pour mettre en place un système de barres de défilement et enfin **LQDrop** pour pouvoir glisser-déposer un fichier ou dossier dans l'application.

Nous n'entrerons pas dans les détails de ces classes pour garder un rapport d'une longueur acceptable.



2.3 Création de l'application

L'interface graphique a été créée en plusieurs temps. Dans un premier temps il a fallu trouver une représentation graphique validée par tous les membres du groupe. Pour cela, nous avons fait la liste des différentes pages qui seraient dans l'application :

- **Connexion** : une page permettant de s'inscrire ou se connecter à son compte
- **Projets** : une page affichant tous les projets stockés sur le serveur distant. C'est sur cette page que nous pouvons rechercher et télécharger un projet en particulier.
- **Dépôt** : une page pour pouvoir déposer ses projets sur l'application.

Par la suite nous les avons dessinés sur papier une par une, pour que chaque membre du groupe ait une vision très précise de l'aspect visuel de l'application. Enfin, nous avons utilisé le framework pour créer et agencer les composants spécifiques au visuel que nous voulions obtenir. Chacune des pages représente une « Vue » dans l'architecture MVC décrite précédemment.

Par exemple pour la page « Projets », Nous avons conçu (liste non-exhaustive) :

- Une liste de boutons permettant de choisir comment trier les projets ("mieux notés", "plus récents", "plus anciens", ...) ¹
- Une barre de recherche pour limiter la liste des résultats. Ainsi la recherche "ab" donne la liste de tout les projets dont le nom contient "ab"
- Des boutons qui permettent d'afficher les projets sous forme de liste ou sous forme de grille avec les images des projets¹
- Une liste où chaque ligne correspond à un projet avec son nom, son « tag » qui est sa catégorie (jeu, algorithme, logiciel...) et son image

Pour composer nos « Vues » encore plus rapidement, nous avons créé une classe `LQTreeCreator`. Cette classe permet de mettre à jour deux variables `parent` et `prev` à chaque nouvel élément ajouté à l'arbre des quarks pour pouvoir faire référence au parent et à l'élément précédent lors de la création d'un nouvel élément. De plus, les méthodes `sub()` et `super()` permettent de passer du quark parent au quark enfant ou inversement. Voici un exemple d'utilisation typique du framework pour créer un composant visuel :

```
1 LQViewable *parent, *prev;
2 createTree(*this, parent, prev)
3 .add<LQViewable>(parent->x(), parent->y(), parent->width(), parent->height()).sub()
4   .add<LQViewable>(10_px, 10_px, 50_px, 50_px)
5   .add<LQViewable>(0_px, 0_px, prev->width(), parent->height()).super()
6 .add<LQViewable>(25_px, 25_px, 100_px, 200_px);
```

1. Cette fonctionnalité n'est pas encore implémentée, c'est donc pour l'instant une simple image

2.4 Communication réseau et base de données

2.4.1 Réalisation d'un client et d'un serveur

La partie réseau de notre application s'est faite à part : en effet, elle utilisait un langage différent du reste (le Python) et devait s'exécuter dans un thread secondaire. Cette partie a donc commencé par l'apprentissage de la programmation en réseau via le module `asyncio` de Python. Ce module étant relativement simple d'utilisation, cette partie n'a pas posé beaucoup de problèmes.

Une fois que les bases étaient maîtrisées, nous avons dû créer deux programmes communicants : un client et un serveur. Le rôle du client était de transmettre les requêtes de l'application au serveur, qui lui devait les analyser, et envoyer une réponse adaptée au client, qui la transmettait à nouveau à l'application. Les requêtes que le serveur devait être capable de gérer étaient peu nombreuses : transfert de projet dans un sens ou dans l'autre (client vers serveur ou serveur vers client) avec ajout dans la base de données, inscription, connexion, et demande de données à la base de données. Ainsi, nous avons opté pour une solution extrêmement simple : pour chacune de ces cinq requêtes, nous associons un numéro (de 1 à 5), que nous insérons au début de la chaîne-requête au moment de sa création dans le C++. Une fois la chaîne envoyée au client python, celui-ci lit le premier octet (donc le numéro de la requête) et exécute la fonction associée. Dans le même temps, il envoie la chaîne au serveur qui fait exactement la même chose. Ainsi, chaque partie de notre système client-serveur sait ce qu'elle doit faire en fonction du besoin de l'application.

2.4.2 Interface Python/C++

Une fois le client et le serveur finis et leur fonctionnalités implémentées, nous devions faire communiquer le client avec l'application C++ afin qu'ils puissent se transmettre des informations. Pour cela, nous avons créé en C++ un module Python basique nommé « gateway » utilisant une classe `ClientGateway`. Cette classe contient deux attributs : deux `std::queue` de tableau de caractères, l'une stockant les requêtes à envoyer au serveur, l'autre stockant les réponses (`m_requests` et `m_responses`).

Ainsi le module possède deux fonctions : `poll_request` qui retourne le premier élément de `m_requests`, et `transmit_response` qui remplit `m_responses` avec la chaîne passée en paramètre. Le client peut donc communiquer avec l'application de la manière suivante : à chaque tour de boucle, il appelle `poll_request` pour voir s'il a reçu une action à exécuter, puis en cas de réponse du serveur, il transmet les informations nécessaires grâce à `transmit_response`. De la même manière, lorsque l'application veut communiquer avec le client, elle remplit `m_requests` ou regarde le contenu de `m_responses`.

Enfin, étant donné que l'application n'a pas accès directement à `m_requests` et `m_responses` puisque que le module s'exécute dans un second thread, nous déclarons une instance de `ClientGateway` dans le contrôleur, puis nous exécutons dans le second thread la fonction de la classe qui lance le programme Python du client. Cette opération nous donne donc accès aux attributs de la `ClientGateway` du second thread directement depuis le contrôleur, dans le premier thread.

2.5 Fonctionnalités de l'interface

Au lancement de l'application Punyduck, une page d'accueil s'affiche¹ pour se connecter ou s'inscrire, avec un identifiant et un mot de passe. Une fois connecté, l'interface principale s'affiche, nous amenant sur la page « Projets ».

L'interface graphique est composé de plusieurs fenêtres, chacune spécifique. Pour les deux fenêtres principales une barre de navigation s'affiche en haut, contenant des liens vers la page « Projet », où l'ensemble des projets mis en ligne seront affichés, et la page « Dépôt » qui servira à déposer un projet.

L'interface des projets¹ permet de naviguer à travers les différents projets mis en ligne sur la plateforme et de les rechercher par leur nom (via la barre de recherche). Pour télécharger un projet, cela se fait simplement via le bouton à droite du tag du projet.

La page dépôt¹, elle, permet à ceux qui souhaitent déposer leur propre projet sur la plateforme de le faire : il suffit de cliquer sur l'onglet "dépôt", glisser et déposer votre dossier ou fichier dans la zone correspondante, lui attribuer un nom puis un tag et enfin l'envoyer en ayant bien remplis tout les champs demandés.

1. Les visuels des différentes pages de l'application sont disponibles en annexe A

2.6 Statistiques

Nous allons maintenant regarder Punyduck d'un point de vue des statistiques :

Le code cumule plus de 5400 lignes de code, dont plus de 4000 concernant uniquement le framework. Ce framework est composé de 28 éléments et, on le voit sur la figure 2.1, il représente les trois quarts du projet. Le serveur est composé de 441 lignes de code, le client Python 310, et l'interface graphique de l'application (définition des « Vues ») 687. Il y a 12 classes utilisées dans ce projet sans compter celles du framework.

Le site web est composé d'un fichier html de 175 lignes de code, d'un fichier css de 702 lignes de code et 7 illustrations.

Nous avons également réalisé un module en python qui a aidé pour la réalisation de l'application, et plus précisément pour faire communiquer le Python avec le C++ .

Concernant le github, il y environ 140 commits en tout, les premiers datant du 19 janvier et les derniers du 10 mai.

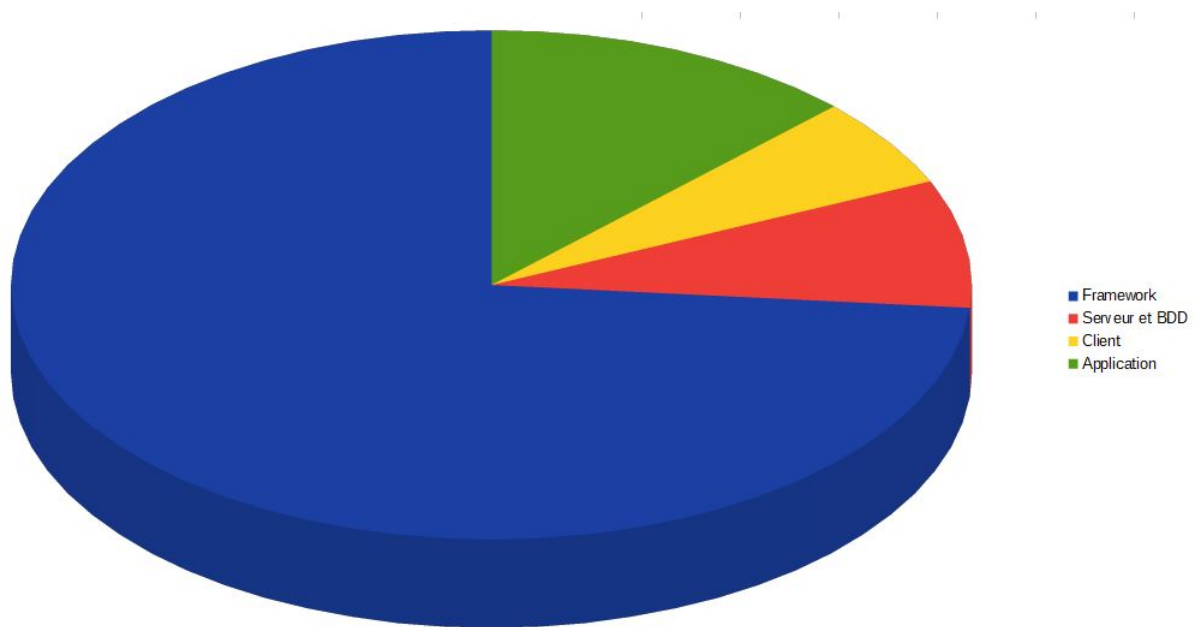


FIGURE 2.1 – Proportion des lignes de code par rapport au partie du projet

Partie 3

Présentation d’algorithmes

Nous allons vous présenter ici deux algorithmes que nous avons jugés intéressants : la fonction du contrôleur qui s’occupe de savoir, à chaque déplacement du curseur, sur quel élément (**LQViewable**) de la fenêtre il se trouve, et la fonction du serveur qui reçoit les requêtes SQL et renvoie une réponse adaptée.

3.1 Fonction de position du curseur

La classe **LQAppController** possède un attribut **s_hover_focus** qui est un pointeur vers le dernier **LQViewable** sur lequel s’est positionné le curseur (initialisé sur la fenêtre de l’application lors du lancement du programme). La méthode **cursor_position_callback** de cette classe est une callback appelée par le module GLFW à chaque déplacement du curseur. Elle prend en paramètre un pointeur vers un objet **GLFWwindow** représentant la fenêtre de l’application, ainsi que les coordonnées absolues du curseur m_x et m_y , en nombre réel. Cette méthode calcule sur quel **LQViewable** s’est déplacé le curseur, puis positionne l’attribut **s_hover_focus** du contrôleur sur ce viewable.

Étant donné qu’un **LQViewable** peut être situé dans un autre élément, et que sa position est relative par rapport à celle de son parent, et non absolue, il nous a fallu trouver une méthode efficace pour retrouver le **LQViewable** dans lequel se trouve le curseur.

Le fonctionnement de la fonction est donc le suivant² : à chaque appel de la fonction, on sauvegarde dans des attributs de la classe **LQAppController** les dernières positions absolues et relatives (par rapport à **s_hover_focus**) du curseur. Ensuite, nous calculons le déplacement Δ de la souris :

$$\Delta = m - prevAbs, \text{ avec } prevAbs \text{ la dernière position absolue du curseur}$$

Ensuite, nous ajoutons ce déplacement Δ sur nos anciennes coordonnées relatives à **s_hover_focus**, afin d’obtenir la nouvelle position relative du curseur par rapport au dernier **LQViewable**. Grâce à

2. Tout le long de l’explication de l’algorithme, nous considérerons les coordonnées en abscisse et en ordonnée comme une seule variable pour plus de simplicité, bien que dans notre programme elles soient séparées en deux variables x et y .

cela, nous pouvons définir un booléen important :

$$outCurrent = (prevRel < 0) \vee (prevRel > dim)$$

avec *prevRel* la position relative du curseur et *dim* les dimension d'un **LQViewable**, ici **s_hover_focus**.

Ce booléen nous permet de savoir, selon un **LQViewable**, si notre curseur se situe à l'intérieur de celui-ci ou à l'extérieur (en supposant avoir les bonnes coordonnées relatives). Une fois que ceci est fait, il ne nous reste plus qu'à vérifier si notre curseur est toujours dans **s_hover_focus** : si oui, alors nous parcourons tous ses fils récursivement pour savoir si l'on se trouve dans l'un d'eux ; si non, alors on remonte les parents de **s_hover_focus** jusqu'à trouver celui dans lequel nous sommes, puis nous parcourons également tous ses fils récursivement afin de trouver l'élément le plus précis dans lequel se trouve le curseur. À chaque déplacement dans l'arbre des éléments, nous devons également ajouter ou retirer à *prevRel* les coordonnées relatives du dernier **LQViewable** que l'on a testées, afin que notre booléen *outCurrent* soit bien calculé avec les bonnes données.

Cet algorithme est de complexité $O(n+m)$ dans le pire des cas, avec n le nombre de **LQViewable** dans l'arbre et m le nombre de fils du **LQViewable** dans lequel le curseur se trouve ; mais sa complexité « amortie » est de $O(p)$ avec p le nombre de fils de **s_hover_focus**. L'implémentation en C++ de cet algorithme est disponible en annexe B.1

3.2 Fonction de gestion des requêtes SQL

Quand le serveur reçoit une requête SQL de la part du client, il doit la traiter avant de l'envoyer à la base de données, puis traiter la réponse avant de la renvoyer au client. Tout ceci se fait dans une fonction, que nous allons décrire ici.

Tout d'abord, le but de cette fonction est le suivant : elle reçoit en paramètre une chaîne de caractères contenant une requête SQL, et elle envoie au client un `bytearray` contenant, dans l'ordre :

1. la chaîne "dataReceive"
2. suivie du nom du modèle qui recevra les données (afin que le client sache qu'il a reçu des données et à quel modèle elles sont destinées) ;
3. le nombre d'items reçus (par exemple, si le modèle à modifier est `Project`, un item correspond à un projet) ;
4. Le nombre total d'attributs de la base de données, ici toujours égal à 15 ;
5. le nombre d'attributs différents reçus (zéro s'ils sont tous envoyés) suivi de leur id ;
6. et enfin, les données binaires récupérées dans la base de données grâce à la requête SQL.

Pour cela, on crée en amont de la fonction un dictionnaire en variable globale, qui associe à chaque nom d'attribut de la base de donnée un id :

```
1 idColonnes = {
2     "idProjet": 0,
3     "valide": 2,
4     "nom": 3,
5     "tag": 4,
6     "pDescr": 5,
7     "pPathImage": 6,
8     "pIdLog": 7,
9     "idLog": 8,
10    "login": 9,
11    "password": 10,
12    "email": 11,
13    "admin": 12,
14    "uPathImage": 13,
15    "uDescr": 14
16 }
```

Listing 3.1 – serveur.py : dictionnaire idColonne

Ensuite, la première étape est de savoir si la requête est de type « `SELECT * FROM ...` » ou si elle demande des attributs précis de la base de données, par exemple « `SELECT idProjet, nom, tag, login FROM Projet, UserInfo WHERE idLog = pIdLog;` ».

Dans le premier cas, la fonction exécute la requête dans la base de données et reçoit un tableau de `tuples` en retour, chaque `tuple` correspondant à une ligne de la base de données. On parcourt donc chaque élément de ce tableau bidimensionnel, et en fonction de sa position dans le `tuple`, on agit différemment : en effet, on ajoute pas la même chose dans la chaîne finale en fonction de si on reçoit une chaîne de caractère, un entier ou un chemin vers une image. Une fois que tous les éléments sont ajoutés à la chaîne finale, il ne reste plus qu'à ajouter les données manquantes au

début de la chaîne, comme décrit juste avant ("dataReceive", nom du modèle, nombre d'items, etc.).

Dans le deuxième cas, c'est plus complexe : avant d'envoyer la requête à la base de données, il faut remettre les attributs dans le même ordre que dans le dictionnaire `idColonnes` (3.1) afin que le client puisse lire la réponse dans le bon ordre.

Pour cela, nous avons implémenté en Python l'algorithme `triFusion` étudié en cours d'algorithmique HLIN301 et HLIN401. En effet, l'objectif est de reconstruire la requête SQL dans le bon ordre avant de l'envoyer à la base de données : nous isolons donc les attributs de la requête dans un tableau grâce à des expressions régulières :

```
1      match = re.search(r'^SELECT (.*) FROM .*;', query)
2      if match != None:
3          # On remet les attributs dans l'ordre afin de les recevoir correctement, et que le
4          # client les recoive dans l'ordre
5          rows = match.group(1).split(', ')
```

Listing 3.2 – serveur.py : expressions régulières

puis on appelle notre fonction `triFusion` sur le tableau d'attributs (la fonction est adaptée pour pouvoir trier le tableau de chaîne de caractère par rapport au dictionnaire `idColonnes`). Le tableau d'attributs est donc dans le bon ordre après cette opération, nous n'avons plus qu'à reconstruire la requête à partir de ce tableau et à l'exécuter dans la base de données.

Une fois que ceci est fait, le reste est essentiellement la même chose que dans le premier cas.

Cette fonction s'exécute en temps $O(nm)$ avec n le nombre d'items renvoyé par la base de données et m le nombre d'attributs pour chaque item. Le code Python de cette fonction est disponible en annexe B.2

Partie 4

Gestion du Projet

4.1 Organisation et planification

Durant le développement de Punyduck, dès son début, nous avons décidé d'avancer le projet après les repas du midi, ou dès que l'occasion se présentait. Étant toujours ensemble la plupart du temps, c'était donc de manière quotidienne que se faisait l'avancée du projet. Quand le travail se faisait à distance via la plate-forme "Discord", nous faisions en sorte de garder toutes traces de ce qui avait été fait, et de ce qui restait à faire, ainsi que de bien penser à mettre le git à jour pour chaque membre du projet.

À chaque fin de mois, nous organisions une entrevue avec notre encadrante Mme Anne-Elisabeth BAERT afin de faire le point sur l'avancée du projet et de connaître ses priorités. Cela nous a permis de ne pas dériver et d'arriver jusqu'à l'aboutissement du projet.

Le développement du projet à été découpé en trois parties (hors site web et rapport) qui sont : la partie réseau en Python (le serveur/client et la base de données PostgreSQL), la partie framework et l'interface de notre application. Le groupe à été séparé au départ en deux groupes : un pour la partie réseau et l'autre pour la partie framework. La dernière partie a réuni les deux groupes afin de produire la liaison Python/C++ et la mise en place de l'interface graphique grâce au framework.

Une fois l'ensemble de l'application finie et fonctionnelle, nous avons mis le serveur en ligne afin que l'on puisse lancer l'application de n'importe où. Également, le site web se faisait en parallèle du projet par Valentin PERON durant cette phase.

4.2 Changements majeurs

L'organisation initiale du projet s'est vue considérablement changée tout au long du projet. En premier lieu, nous avons dû gérer la programmation de l'application avec seulement 3 membres du groupes sur les 5. Les tâches ont donc été au départ répartie de la manière suivante : Valentin FONTAINE s'occupait du framework, Esteban BARON des débuts de l'interface graphique utilisant le framework, et Paul BUNEL de la partie réseau en Python.

Puis, une fois ces parties bien avancées, nous avons commencé à mettre en place l'architecture MVC de notre application. À partir de là, nous n'étions plus que deux à nous occuper du code de l'application : ainsi, Valentin FONTAINE, aidé par Paul BUNEL, se sont donc occupés de l'implémentation de l'architecture MVC, ainsi que de la majorité de l'interface graphique de l'application.

Le diagramme de Gantt final du projet est disponible à la figure 4.1.

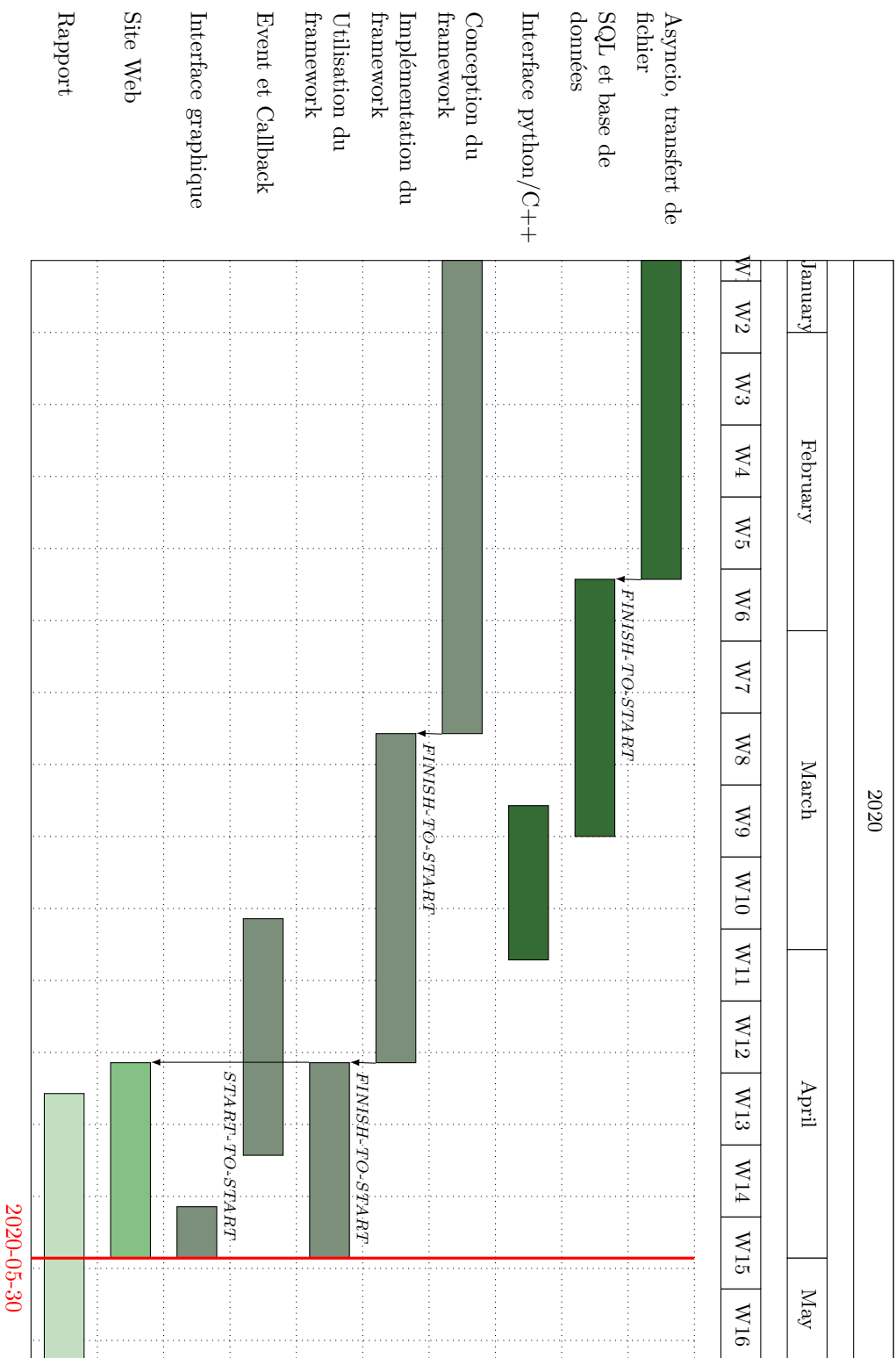


FIGURE 4.1 – Diagramme de Gantt du projet

Partie 5

Bilan et Perspectives

Le bilan pour la fin de projet est positive, bien que les objectifs du cahier des charges n'aient tous atteints par manque d'investissement de certains, nous avons quand même énormément appris grâce à ce projet, et avons produit une application dont nous sommes fiers même avec une équipe réduite pour le développement.

Grâce au développement de cette application, nous avons pu mettre en pratique les enseignements de plusieurs modules de licence : le module HLIN304 - Systèmes d'information et bases de données 1 pour les requêtes SQL, les modules de programmation impérative pour le développement en C++, le module HLIN406 - Modélisation et programmation par objet 1 pour la programmation orientée objet et le diagramme UML du framework, ainsi que les modules d'Algorithmique et de complexité pour l'étude de la complexité de nos principaux algorithmes.

De plus, nous avons réalisé cette application en partant du bas niveau pour le côté graphique, ce qui nous a permis de comprendre certaines mécaniques lors de la réalisation d'une telle application ; et nous avons pour la première fois fait de la programmation en réseau et en *multithread*.

Perspectives

Après la fin de ce projet, nous avons encore beaucoup de choses à améliorer pour l'application : plus de personnalisation de l'interface graphique, amélioration du framework, augmenter la sécurité du serveur, etc. Nous comptons donc continuer le développement du projet durant l'été qui suit, afin de produire une application plus aboutie, avec un maximum de fonctionnalités implémentées sur les 3 plans framework / application / réseau.

Annexes

Annexe A

Visuels de l'application

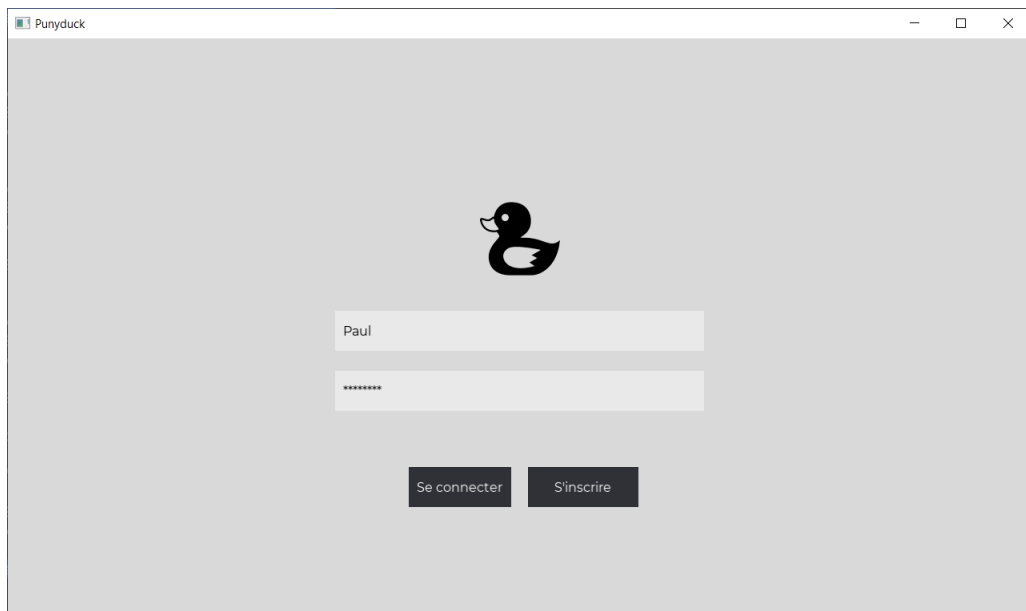


FIGURE A.1 – Capture d'écran de l'interface de connexion

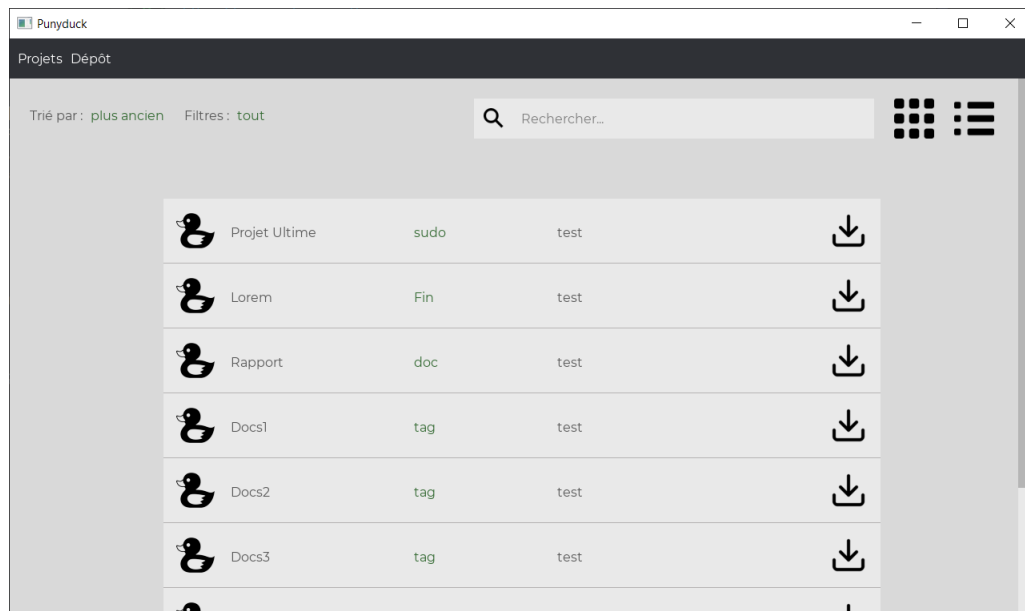


FIGURE A.2 – Capture d’écran de la page projets

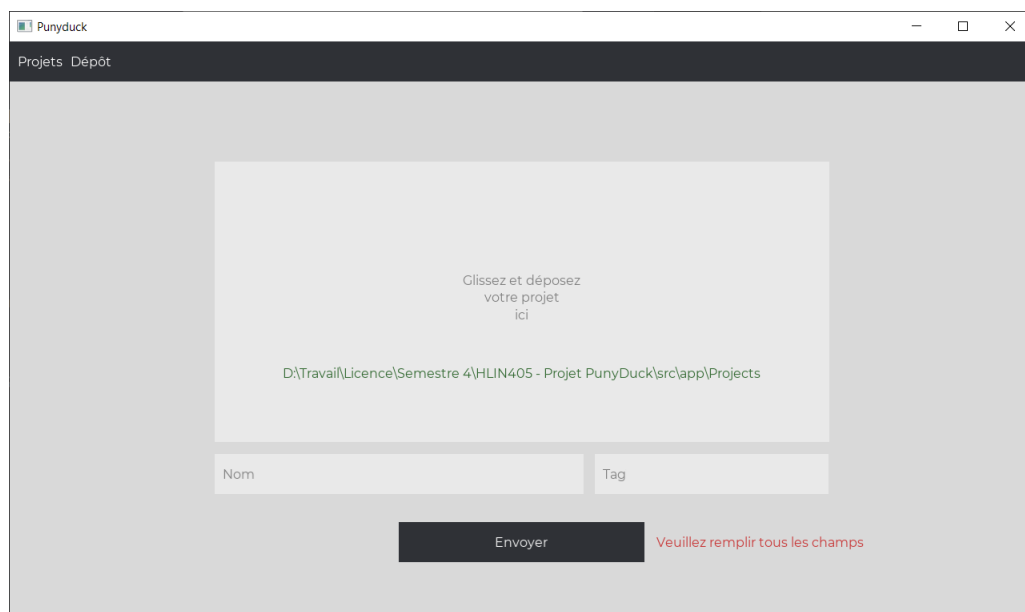
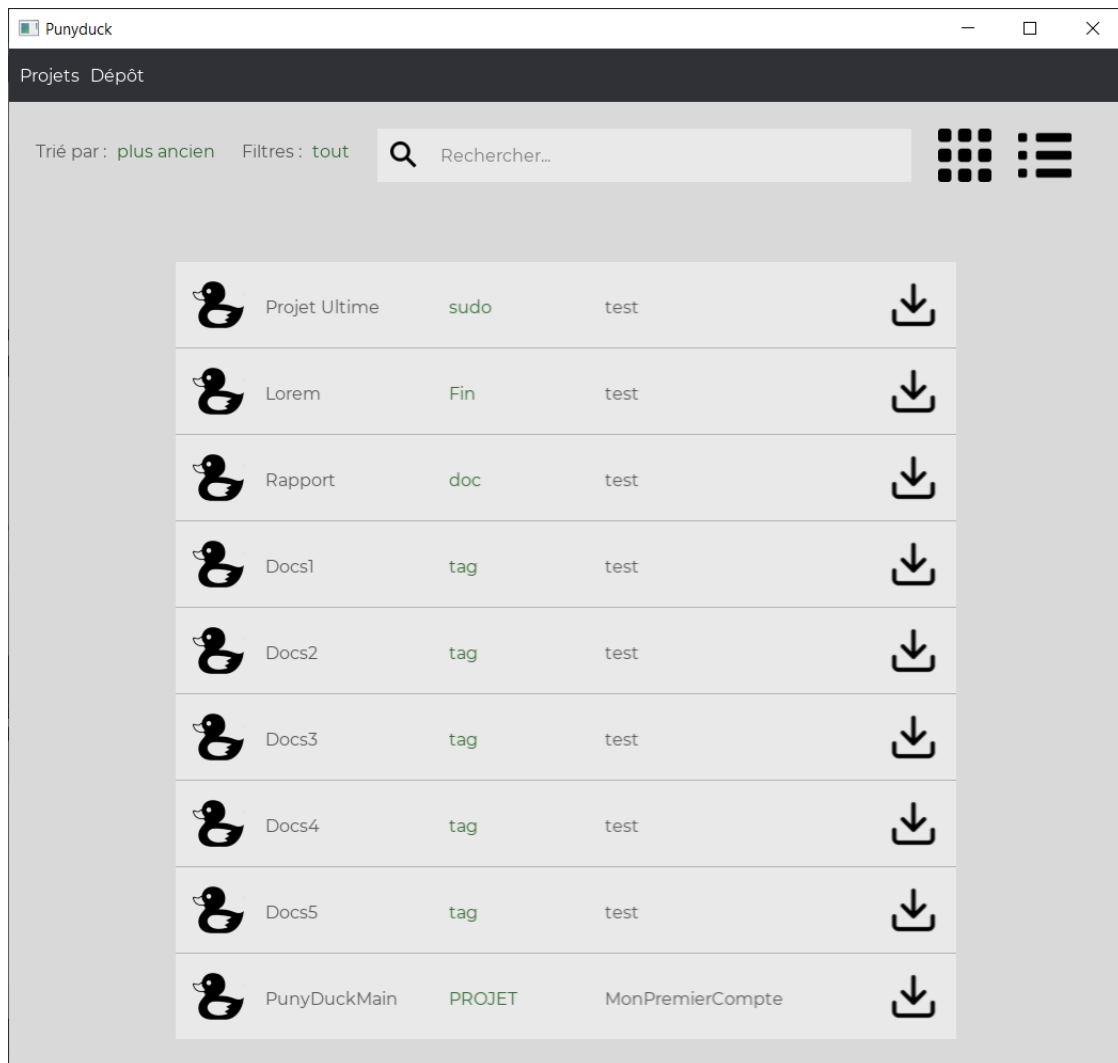


FIGURE A.3 – Capture d’écran de l’interface de dépôt

FIGURE A.4 – Exemple du *responsive design* de la page « Projets »

Annexe B

Principaux algorithmes

B.1 Fonction `cursor_position_callback`

Callback appelée lors d'un déplacement du curseur, calculant le `LQViewable` sur lequel le curseur se positionne.

```
1 void LQAppController::cursor_position_callback(GLFWwindow* window, double mx, double my) {
2     float deltaX = mx - prevAbsX;
3     float deltaY = my - prevAbsY;
4     prevRelX += deltaX;
5     prevRelY += deltaY;
6
7     LQViewable* current = s_hover_focus;
8     bool outCurrent = prevRelX < 0 || prevRelX > current->widthF() ||
9                     prevRelY < 0 || prevRelY > current->heightF();
10
11     // On cherche dans quel parent de s_hover_focus nous sommes
12     while (outCurrent && current != s_window) {
13         prevRelX += current->xF();
14         prevRelY += current->yF();
15         current = static_cast<LQViewable*>(current->parent());
16         outCurrent = prevRelX < 0 || prevRelX > current->widthF() ||
17                 prevRelY < 0 || prevRelY > current->heightF();
18     }
19
20     s_hover_focus = current;
21     current = static_cast<LQViewable*>(current->firstChild());
22     while (current) { // On recherche un fils correspondant a notre position
23         outCurrent =
24             prevRelX < current->xF() || prevRelY < current->yF() ||
25             prevRelX > current->xF() + current->widthF() ||
26             prevRelY > current->yF() + current->heightF();
27         if (outCurrent) {
28             current = static_cast<LQViewable*>(current->nextSibling());
29         }
30         else {
31             s_hover_focus = current;
32             prevRelX -= current->xF();
33             prevRelY -= current->yF();
34             current = static_cast<LQViewable*>(current->firstChild());
```

```

35     }
36 }
37
38 prevAbsX = mx;
39 prevAbsY = my;
40 }

```

Listing B.1 – LQAppController::cursor_position_callback

B.2 Fonction SQL

Fonction exécutée par le serveur lorsque le client envoie une requête SQL.

```

1  def fusion(T1, T2, T):
2      i1 = 0
3      i2 = 0
4      for i in range(len(T)):
5          if i1 >= len(T1):
6              T[i] = T2[i2]
7              i2 += 1
8          elif i2 >= len(T2):
9              T[i] = T1[i1]
10             i1 += 1
11          elif idColonnes[T1[i1]] < idColonnes[T2[i2]]:
12              T[i] = T1[i1]
13              i1 += 1
14          else:
15              T[i] = T2[i2]
16              i2 += 1
17
18  def triFusion(T): # Algorithme triFusion vu en cours HLIN301 et HLIN401
19      if len(T) > 1:
20          T1 = T[:int(len(T)/2)]
21          triFusion(T1)
22          T2 = T[int(len(T)/2):]
23          triFusion(T2)
24
25          fusion(T1, T2, T)
26
27  nAttributes = int(15).to_bytes(1, 'big')
28
29  async def SQL(writer, query):
30      # query = model + requete SQL
31      model = re.search(r'(.*)SELECT', query).group(1)
32      query = re.search(r'(SELECT.*)', query).group(1)
33
34      infos = b''
35      nItems = 0
36
37      if "*" in query: # Si la requete est du type "SELECT * FROM ..."
38          cur.execute(query)
39          datas = cur.fetchall()
40
41          ordreIndice = int(0).to_bytes(4, 'big')
42          if "Projet" in query:
43              for data in datas:

```

```
44         for row in range(len(data)): # On agit differemment en fonction du type de donnees
45             if row == 1 or row == 6:
46                 img = Image.open(data[row].replace("\\\\", "\\"))
47                 infos += img.width.to_bytes(4, 'big')
48                 infos += img.height.to_bytes(4, 'big')
49                 infos += GL_RGBA if img.mode == "RGBA" else GL_RGB
50                 infos += len(img.tobytes()).to_bytes(4, 'big')
51                 infos += img.tobytes()
52             elif row == 0 or row == 7 or row == 2:
53                 infos += data[row].to_bytes(4, 'big')
54             else:
55                 infos += data[row].encode() + b'\0'
56             nItems += 1
57
58     elif "UserInfo" in query:
59         for data in datas:
60             for row in range(len(data)): # On agit differemment en fonction du type de donnees
61                 if row == 5:
62                     img = Image.open(data[row])
63                     infos += img.width.to_bytes(4, 'big')
64                     infos += img.height.to_bytes(4, 'big')
65                     infos += GL_RGBA if img.mode == "RGBA" else GL_RGB
66                     infos += len(img.tobytes()).to_bytes(4, 'big')
67                     infos += img.tobytes()
68                 elif row == 0:
69                     infos += data[row].to_bytes(4, 'big')
70                 else:
71                     infos += data[row].encode() + b'\0'
72                 nItems += 1
73
74     else: # Si la requete est du style "SELECT ..., ... FROM ..."
75         match = re.search(r'^SELECT (.*) FROM .*;', query)
76         if match != None:
77             # On remet les attributs dans l'ordre afin de les recevoir correctement, et que le
78             # client les receive dans l'ordre
79             rows = match.group(1).split(', ')
80             endQuery = re.search(r'(FROM.*)', query).group(1)
81             triFusion(rows)
82             query = "SELECT "
83             for row in rows[:-1]:
84                 query += row + ", "
85             query += rows[-1] + " " + endQuery
86
87             # On sauvegarde l'ordre d'arrivee des attributs de la base de donnees
88             ordreIndice = len(rows).to_bytes(4, 'big')
89             for row in rows:
90                 ordreIndice += idColonnes[row].to_bytes(4, 'big')
91
92             cur.execute(query)
93             datas = cur.fetchall()
94
95         for data in datas:
96             for row in range(len(data)): # On agit differemment en fonction du type de donnees
97                 if idColonnes[rows[row]] == 0 or idColonnes[rows[row]] == 2 or
98                 idColonnes[rows[row]] == 7 or idColonnes[rows[row]] == 8:
99                     infos += data[row].to_bytes(4, 'big')
100             elif idColonnes[rows[row]] == 6 or idColonnes[rows[row]] == 13:
101                 fp = open(data[row], 'rb')
```

```
102         img = Image.open(fp)
103         infos += img.width.to_bytes(4, 'big')
104         infos += img.height.to_bytes(4, 'big')
105         infos += GL_RGBA if img.mode == "RGBA" else GL_RGB
106         infos += len(img.tobytes()).to_bytes(4, 'big')
107         infos += img.tobytes()
108     else:
109         infos += data[row].encode() + b'\0'
110     nItems += 1
111
112     # On construit la chaine a renvoyer au client
113     infos = "dataReceive".encode() + b'\0' + model.encode() + b'\0' + \
114           nItems.to_bytes(4, 'big') + nAttributes + ordreIndice + infos
115     # str + \0 + str + \0 + 4 bytes + 1 byte + (4 bytes + 4 * nbIndice bytes) + n bytes
116
117     await send_message(writer, infos)
```

Listing B.2 – serveur.py : SQL