D-BOOKER éditions

CRÉER DES APPLICATIONS GRAPHIQUES en Python avec PyQt5

Thibaut Cuvelier, Pierre Denis

Créer des applications graphiques

en Python avec PyQt5

par

Pierre Denis

Thibaut Cuvelier

Créer des applications graphiques - en Python avec PyQt5

par Pierre Denis, Thibaut Cuvelier

ISBN (EPUB) : 978-2-8227-0516-5

Publié par les Éditions D-BookeR, Parc des Rives créatives de l'Escaut, Nouvelle Forge, 80 avenue Roland Moreno, 59410 Anzin

www.d-booker.fr

contact@d-booker.fr

Couverture : d'après une création de Marie Van Der Marlière (www.marie-graphiste.com)

Mise en page : générée sous Calenco avec des XSLT développées par la société NeoDoc (www.neodoc.biz)

Date de publication : 20/03/2017

Édition : 1

Version : 1.0

À propos des auteurs

Pierre Denis

Ingénieur civil en informatique, j'ai travaillé plus de 15 ans chez Spacebel, une société belge de développement logiciel dans le domaine spatial. J'ai découvert Python en 1999 et j'ai été immédiatement séduit. Je l'ai d'abord utilisé dans mon travail pour des tâches de support (qualité, documentation, livraison…), puis, progressivement, pour des projets de plus en plus complexes. J'ai notamment participé au développement de plusieurs applications GUI avec vues cartographiques pour des centres de mission satellite ; partant d'ILOG Views, wxPython et Qt, l'environnement de développement a évolué vers PyQt, ce qui s'est révélé extrêmement productif. Dans mes temps libres, je crée des projets open-source en Python (Unum, Lea) et de petites expériences en PyQt, dont un jeu de combat spatial.

Thibaut Cuvelier

Je me suis d'abord formé à l'informatique en autodidacte, avant d'entreprendre des études d'ingénieur civil en informatique ; je poursuis mes études par un doctorat en mathématiques appliquées à l'université de Liège.

Conscient de l'importance du partage des acquis, je contribue activement depuis 2009 au site Developpez.com. De fil en aiguille, j'en suis arrivé à découvrir le langage Python, très pratique pour l'écriture de scripts, puis d'applications plus lourdes. En parallèle, j'ai aussi fait la connaissance de Qt pour le développement d'interfaces graphiques, plus productif encore en Python ou avec Qt Quick. J'apprécie particulièrement ce dernier dans le cas d'applications mobiles.

Avant-propos

1. Objectifs du livre

Ce livre a été pensé pour vous mettre le pied à l'étrier dans le développement d'interfaces graphiques, que ce soit de manière classique (Développement d'une application avec des widgets) ou déclarative (Développement d'une application avec Qt Quick). Ces modules présenteront des interfaces graphiques relativement simples, mais représentatives de la grande majorité des besoins, peu importe votre choix de paradigme. Deux autres modules, plus avancés, complètent ces introductions (Affichage 2D interactif avec les vues graphiques côté classique, Qt Quick avancé côté déclaratif) avec des aspects dynamiques.

Si vous avez besoin de dépasser ce stade, vous aurez suffisamment intégré les principes pour exploiter de manière efficace la documentation et le reste de PyQt. Par conséquent, cet ouvrage ne se conçoit pas comme une référence absolue à PyQt (qui serait de toute façon rapidement dépassée), mais bien comme un manuel d'apprentissage pour vous lancer dans le développement d'interfaces graphiques.

2. Public visé et prérequis

Ce livre s'adresse à toute personne souhaitant programmer des interfaces graphiques en Python, que ce soit pour définir une simple boîte de dialogue ou développer une application plus complexe avec menus, barre d'outils, accès à une base de données, etc.

D'une manière générale, nous supposons que vous maîtrisez les notions de programmation orientée objet en Python, notamment l'héritage. Autrement dit, vous débutez avec Qt, mais vous connaissez déjà Python.

Vous pouvez lire les chapitres dans l'ordre qui vous convient. Toutefois, certains sont plus complexes et requièrent des notions introduites dans d'autres ; ces prérequis et le niveau de difficulté sont précisés en début de chaque chapitre. Notamment, les deux modules plus avancés se basent sur le contenu des modules précédents : il vaut mieux avoir lu Développement d'une application avec des widgets avant Affichage 2D interactif avec les vues graphiques , avoir lu Développement d'une application avec Qt Quick avant Qt Quick avancé.

3. Organisation du livre

Les premiers chapitres de l'ouvrage (Préliminaires) portent sur des aspects généraux, non spécifiques à une manière ou l'autre de développer les interfaces graphiques. Il s'agit principalement de vous familiariser avec un environnement de développement intégré Python spécifiquement prévu pour PyQt — eric6.

Ensuite, vous pourrez choisir votre voie :

- une approche impérative par assemblage de composants, des *widgets* (Développement d'une application avec des widgets) ;
- une approche déclarative par énonciation de liens entre des composants (Développement d'une application avec Qt Quick).

Ces deux modules s'orienteront autour du développement d'une même application de gestion de bibliothèque. Vous pouvez les lire de manière totalement indépendante. La différence entre les deux réside réellement dans la philosophie : l'une ou l'autre approche est plus adaptée selon l'application à développer et la manière de penser des programmeurs.

Un autre point important qui les distingue est le langage de programmation : Python est un langage impératif, il est difficile de l'utiliser dans l'approche déclarative ; c'est pour cela que Qt Quick utilise un autre langage, développé spécifiquement pour ce cadre (QML), même s'il est très facile de faire communiquer les deux environnements.

Ensuite, les deux modules plus avancés proposent de construire sur ces acquis, afin de réaliser des applications plus dynamiques :

- soit de manière impérative avec les vues graphiques (Affichage 2D interactif avec les vues graphiques) : ce cadre permet de réaliser des applications de visualisation présentant un grand nombre d'éléments dynamiques. C'est notamment le cas d'applications d'ingénierie ou de jeux vidéo ;
- soit de manière déclarative (Qt Quick avancé), avec diverses fonctionnalités pour écrire plus facilement des applications dynamiques, y compris avec du dessin en 2D et des scènes 3D.

4. Codes sources des exemples

Vous trouverez sur la page du site des éditions D-BookeR consacrée au livre, onglet Compléments, une archive réunissant l'ensemble des codes sources des exemples du livre. Ceux-ci sont classés par projet et chapitre.

Toutefois pour vous simplifier l'accès à ces codes au fur et à mesure de votre lecture, nous les avons aussi hébergés sur GitHub dans un dossier dédié. Chaque fois que vous rencontrerez l'icône  , un simple clic dessus vous renverra vers le dossier contenant le code complet de l'exemple ou des exemples du chapitre.

5. Réglage de la largeur de l'écran

Vous trouverez de nombreux exemples de code, formatés dans une police à chasse fixe. Afin d'éviter des retours à la ligne inopportuns à l'intérieur d'une ligne de code, la longueur maximale des lignes de code a été fixée à 70 caractères, une valeur suffisamment basse pour être affichée sur la plupart des supports, tout en étant suffisante pour que le code puisse être correctement formaté.

Toutefois, il est possible que sur votre support la largeur maximale affichable soit inférieure à la limite fixée. Le paragraphe test ci-dessous permet de vérifier votre affichage. Il doit tenir sur deux lignes exactement :

```
000000000011111111112222222222333333333344444444445555555556666666666

01234567890123456789012345678901234567890123456789012345678901234567890
```

Si ce n'est pas le cas, regardez si vous pouvez agrandir la taille de la fenêtre, diminuer la taille des marges ou diminuer la taille de la police d'affichage. Sur un téléphone portable, placez-le plutôt en mode paysage. Si vous n'y arrivez pas, ne vous inquiétez pas pour autant, la plupart des lignes de code sont inférieures à 65 caractères.

6. Accès aux vidéos

La version numérique du livre contient quelques illustrations animées 🎞.

Si vous lisez ce livre en ligne, elles sont intégrées à votre page et votre navigateur ira chercher de lui-même le format de vidéo qu'il supporte.

Si vous lisez une version téléchargée, un clic sur l'image vous redirigera vers la vidéo en ligne au format MP4. Si votre navigateur par défaut ne supporte pas nativement le MP4, modifiez à la main l'extension du fichier dans l'url en remplaçant `.mp4` par `.webm`.

***Note** > Avant de cliquer, assurez-vous que le pointeur de votre souris s'est changé en main.*

Préliminaires

Présentation de PyQt

PyQt est la contraction de deux mots : d'un côté, Python (le langage de programmation utilisé) réputé fort simple d'apprentissage ; de l'autre, Qt, un cadriciel extrêmement complet (principalement pour des interfaces graphiques), mais écrit en C++. PyQt sert de couche de liaison entre ces deux mondes et apporte Qt à l'environnement Python.

Qt est une bibliothèque multiplateforme, reconnue avant tout pour ses fonctionnalités d'aide à la conception d'interfaces graphiques. Cependant, Qt peut faire beaucoup plus : cette bibliothèque vient avec des modules pour l'accès aux bases de données SQL, un navigateur web complet réutilisable, un système d'aide, des fonctionnalités multimédia. Depuis quelque temps, elle propose de nouvelles fonctionnalités plus intégrées et de plus haut niveau, comme l'accès à des outils de cartographie et de localisation, à la communication sans fil (NFC, Bluetooth), à des graphiques et de la visualisation de données, etc. Également, son environnement est très riche, avec de nombreuses autres bibliothèques d'extension disponibles (voir la Section 2, Environnement de PyQt).

Son principal point fort est de s'adapter aux nouvelles utilisations de l'informatique : il est l'une des très rares bibliothèques d'interfaces graphiques généralistes à s'être implantée dans le domaine des applications mobiles (il peut s'exécuter sur Android, iOS et Windows Phone) et à proposer une hybridation avec des applications web. Un autre point à souligner est Qt Quick, une technologie déclarative de développement d'interfaces : au lieu d'écrire du code pour effectuer le lien entre deux parties d'une interface graphique (un bouton et le texte affiché, par exemple), il suffit de *déclarer* qu'il existe une relation entre les deux ; ce paradigme est détaillé dans Développement d'une application avec Qt Quick.

***Note** > Qt dispose également d'une série d'autres modules, nettement plus utiles en C++ qu'en Python, comme des chaînes de caractères évoluées, avec des expressions régulières, ou encore l'accès au réseau et à Internet. Bon nombre de ces fonctionnalités sont disponibles de longue date en Python, mais elles ne sont arrivées que très récemment dans l'environnement standard C++.*

1. PyQt et les autres bibliothèques de développement d'interfaces graphiques

PyQt n'est pas la seule manière de réaliser des interfaces graphiques. En réalité, l'environnement Python ne manque pas de choix : notamment, Tkinter (construit par-dessus Tk), qui a l'avantage d'être livré par défaut avec Python ; cependant, il possède peu de composants graphiques de base (boutons, zones de texte, etc.). En outre, vous devrez recourir à des extensions pour disposer de composants graphiques très utilisés, comme des boîtes de dialogue (pour afficher une information à l'utilisateur ou lui demander de sélectionner un fichier), ainsi que des fonctionnalités comme le glisser-déposer. Beaucoup de développeurs lui reprochent une mauvaise intégration avec l'environnement de bureau, un souci qui a toutefois été corrigé avec les années.

L'autre grand concurrent est wxPython, qui à nouveau correspond à une couche de liaison vers une bibliothèque C++ (wxWidgets). Comme PyQt, il doit être installé séparément de Python et est livré avec quantité de composants graphiques (à la différence de Tkinter). Cependant, le développement de la branche actuelle est très lent (pas de nouvelle version depuis 2014) et wxPython n'est toujours pas compatible avec Python 3 (contrairement aux deux autres ; pourtant, Python 3.0 est sorti en 2008). Le projet Phoenix a un développement actif et remédie à ces problèmes, mais il n'est pas encore utilisable par le grand public.

Alors, quelle bibliothèque utiliser ? Toutes trois sont très matures, elles existent déjà depuis un certain temps et sont disponibles gratuitement ; chacune a sa propre communauté, qui propose son aide sur les forums et listes de diffusion. Tkinter est souvent dite pour être la moins intuitive des trois, PyQt est la seule à proposer une approche déclarative (détaillée dans Développement d'une application avec Qt Quick).

Ces aspects techniques ne sont pas les seuls importants, l'environnement de chaque bibliothèque importe au moins autant. PyQt dispose d'une documentation de qualité et celle de Qt reste exploitable en Python. Aussi bien wxPython que PyQt disposent d'un éditeur visuel d'interfaces (respectivement, wxGlade et Qt Designer).

Globalement, PyQt dispose d'un bon nombre d'avantages par rapport à ses concurrents, tant au niveau technique pur que dans l'environnement : après les avoir essayés, beaucoup de personnes (dont les auteurs) préfèrent généralement travailler avec PyQt plutôt que wxPython ou Tkinter.

En ce qui concerne les licences, Tkinter est distribué sous la même licence que Python, wxPython sous la licence wxWidgets (similaire à la LGPL) et PyQt sous la licence GPL ou une licence commerciale (payante). En pratique, pour distribuer une application utilisant l'une de ces bibliothèques, seul PyQt peut poser problème : il est nécessaire de la distribuer sous la licence GPL ou de payer pour une licence commerciale.

Note > *PySide est une autre couche de liaison Python pour Qt, la différence principale étant la licence : PySide est disponible sous la LGPL, très peu restrictive. Cependant, son développement a été longtemps à l'arrêt : il reprend maintenant sous l'égide du Qt Project.*

2. Environnement de PyQt

L'un des grands avantages de Qt et de PyQt, c'est qu'ils sont bien entourés. L'environnement de PyQt comporte notamment quelques outils incontournables qui vous simplifieront la vie dans un grand nombre de tâches ; bon nombre proviennent d'ailleurs de Qt en C++ et ont été légèrement adaptés pour PyQt.

2.1. Outils

Qt Designer est prévu pour la conception *graphique* d'interfaces, c'est-à-dire que vous les concevez à l'aide de la souris et de glisser-déposer d'éléments pour former la structure de la fenêtre. Une fois composée, la fenêtre peut être transformée en code (à l'aide de l'utilitaire `pyuic`).

Pour les traductions, Qt Linguist peut être utilisé par les traducteurs, en combinaison avec d'autres outils pour extraire le texte et le transmettre (`pylupdate`) et pour générer la version des traductions utilisable par PyQt (`lrelease`).

`pyqtdeploy` est un utilitaire qui facilite le déploiement d'applications PyQt, en prenant en compte ses spécificités (notamment la dépendance à des bibliothèques non écrites en Python, qui sont une source de problèmes avec d'autres outils de déploiement).

Tous ces outils sont généralement bien intégrés dans des environnements de développement intégrés (EDI), le principal pour PyQt étant eric. Qt Creator est l'environnement de référence pour Qt et s'ouvre petit à petit à Python ; il jouit également d'une très bonne intégration avec ces outils ; il est d'ailleurs le seul EDI gratuit à intégrer Qt Quick.

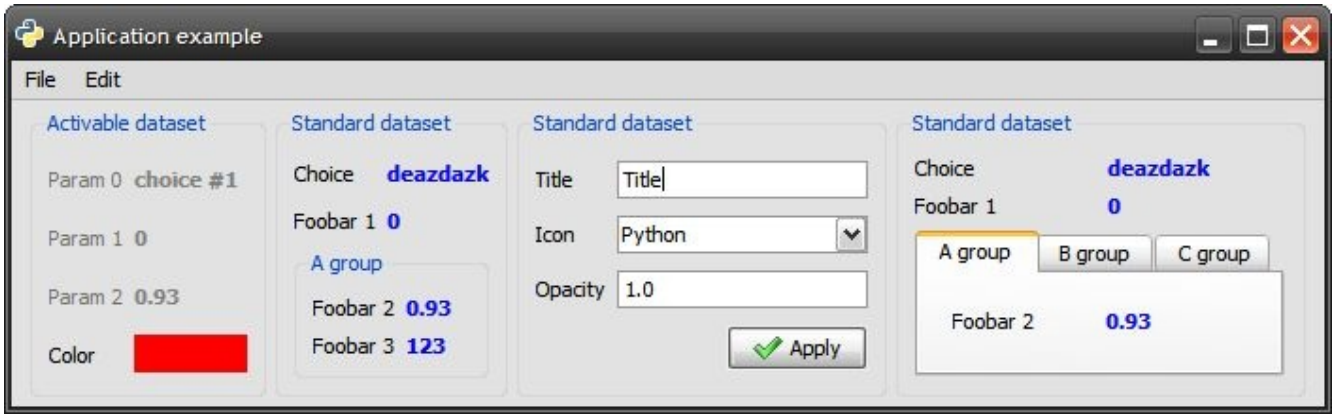
2.2. Bibliothèques

Outre ces quelques outils, livrés en standard avec toute distribution de PyQt, l'environnement contient un grand nombre de bibliothèques implémentant des fonctionnalités moins courantes, mais néanmoins utiles selon les cas.

Beaucoup d'applications exploitent une base de données (notamment les exemples

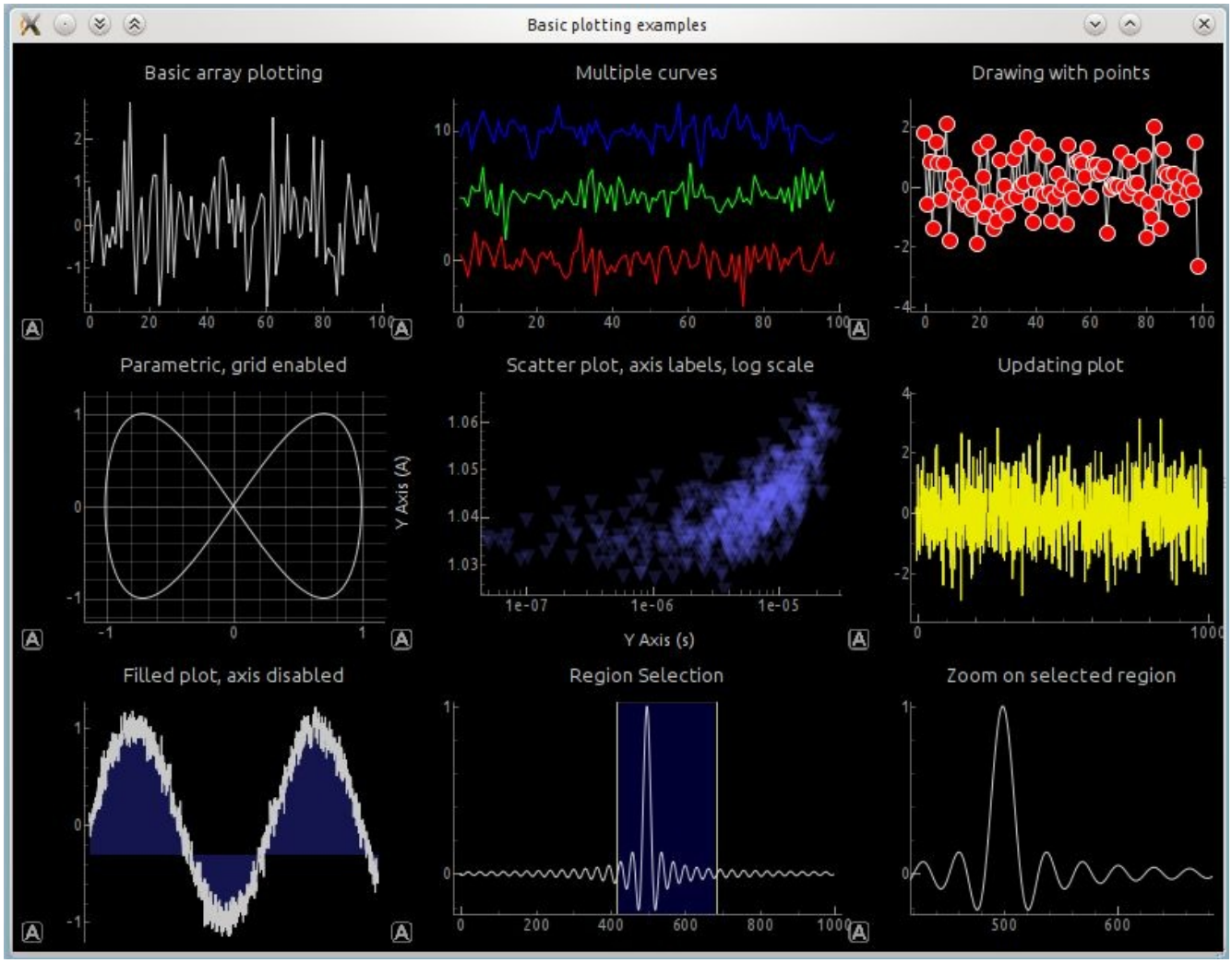
développés dans la suite de l'ouvrage), principalement relationnelles (SQL). Camelot permet d'utiliser une interface orientée objet pour ces dernières (à travers SQLAlchemy). GuiData remplit un objectif similaire ; il est mis à jour plus fréquemment, mais se limite à la partie d'édition des données.

Figure 1.1 : *Exemple d'interface réalisée avec GuiData*



Côté graphiques, PyQtGraph et GuiQwt proposent des fonctionnalités similaires pour la visualisation de données bi- et tridimensionnelles ou encore d'images avec éléments interactifs. Ces composants sont principalement prévus dans des applications scientifiques, notamment dans les domaines de l'ingénierie ou du médical.

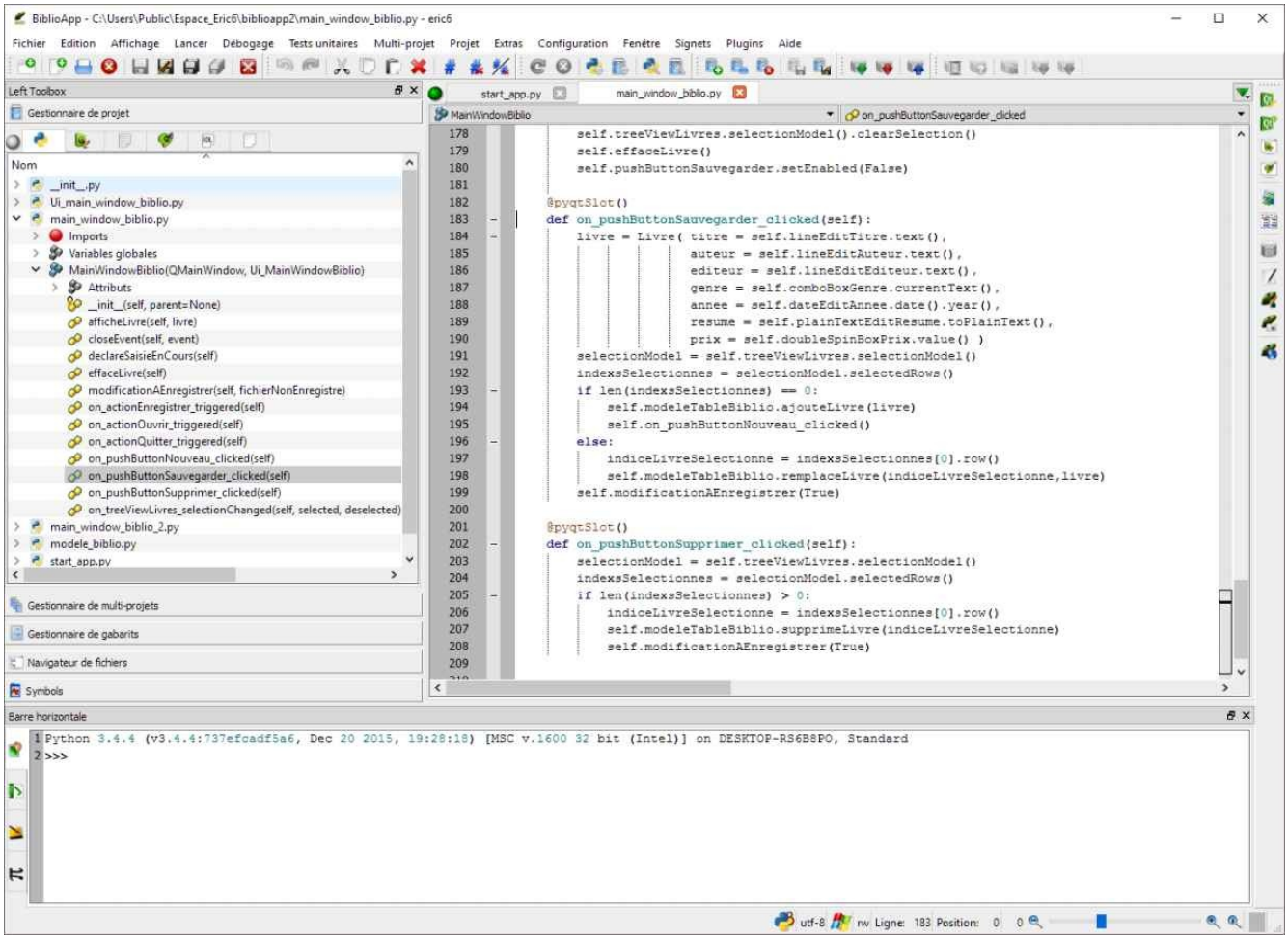
Figure 1.2 : *Quelques possibilités de PyQtGraph en 2D*



Note *> Depuis PyQt 5.7, des outils similaires sont fournis gratuitement avec PyQt, même s'ils doivent être téléchargés séparément. PyQtCharts affiche des graphiques bidimensionnels (comme PyQtGraph et GuiQwt). PyQtDataVisualization en est la déclinaison tridimensionnelle.*

QScintilla propose un composant d'édition de texte principalement prévu pour le code informatique, avec notamment la coloration syntaxique ou des facilités de présentation d'erreurs ou d'intégration d'autocomplétions. Il est notamment utilisé pour développer eric6.

Figure 1.3 : *Utilisation de QScintilla dans eric6*



Pour les plateformes mobiles (iOS et Android) ainsi que macOS, PyQtPurchasing propose d'effectuer des achats depuis l'intérieur d'une application tout en s'intégrant avec l'outil d'achat d'applications de la plateforme (Mac App Store, App Store, Android Market).

3. Choix d'une interface

Tout comme PyQt par rapport à ses concurrents, à l'intérieur même de la bibliothèque, vous devrez effectuer un choix pour la manière d'écrire votre application. Comment s'y retrouver entre Qt Widgets, Qt Quick, Graphics View ? Ces différentes approches ont peut-être l'air de s'opposer, mais elles peuvent se compléter mutuellement pour profiter des avantages de chacune dans une même application.

Outre une affaire de goût, ces méthodologies se distinguent sur le type d'applications visé. De manière générale :

- pour développer une interface graphique classique sur un ordinateur avec clavier et souris, **Qt Widgets** est bien adapté. Cette possibilité, qui utilise uniquement le langage Python, est détaillée dans Développement d'une application avec des widgets. Si cette interface nécessite par ailleurs des graphiques complexes présentant beaucoup d'interactions avec l'utilisateur (vue cartographique, CAO, jeux, etc.), l'approche Qt Widgets peut être complétée avec le framework **Graphics View**, qui est couvert dans Affichage 2D interactif avec les vues graphiques ;

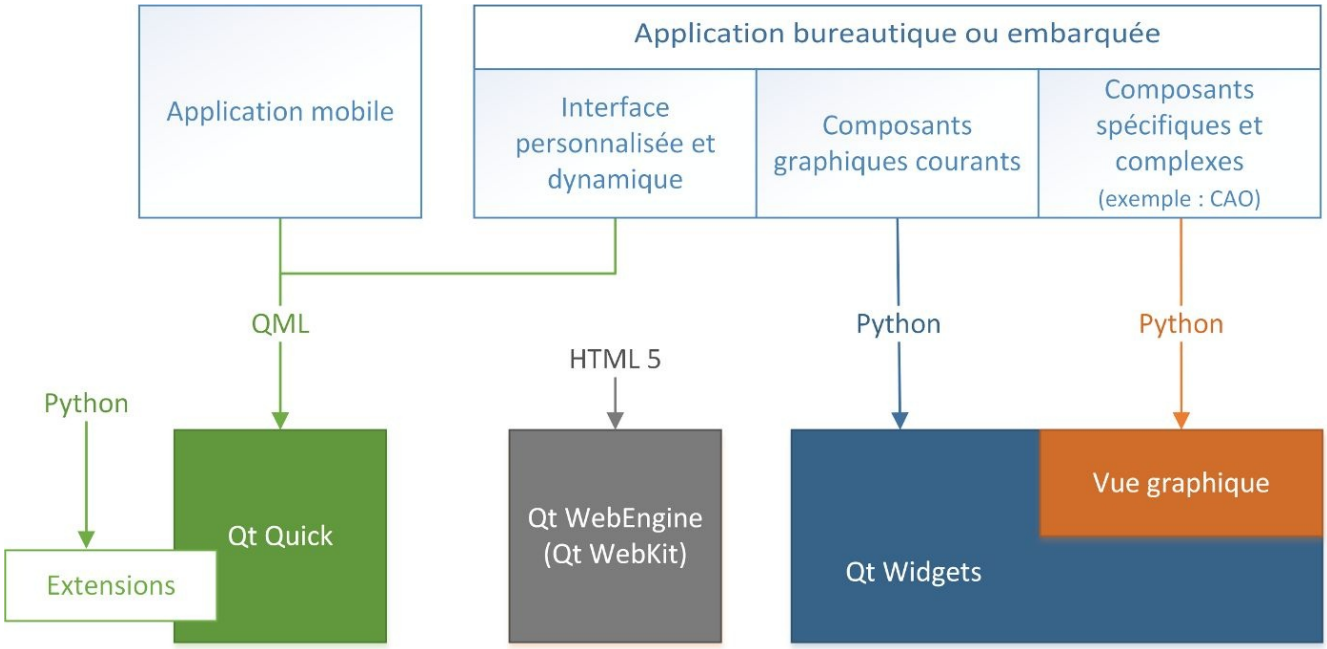
- pour développer une interface pour un téléphone portable, une tablette ou un écran tactile en général, le meilleur choix est probablement **Qt Quick**, puisqu'il a été prévu pour ces utilisations, mais il reste parfaitement adapté à bon nombre d'applications plus traditionnelles. Il nécessite cependant l'apprentissage d'un autre langage de programmation, QML. De manière générale, les nouveautés de Qt seront développées pour Qt Quick (bien que généralement disponibles en Python). Cette possibilité est détaillée dans Développement d'une application avec Qt Quick ;

- pour développer une interface avec des technologies web (par exemple, pour utiliser le même code sur un site web ou pour une vraie application), c'est-à-dire une application hybride, votre choix devra se porter sur **Qt WebEngine**. Ces applications doivent exploiter une base avec Qt Widgets ou Qt Quick. Vous trouverez régulièrement des références à Qt WebKit, mais ce dernier n'est plus maintenu officiellement (même si la communauté a remis le projet au goût du jour).

Ces premiers éléments sont lapidaires et permettent d'orienter rapidement un choix vers une technologie qui sera probablement adaptée, bien que la situation varie énormément,

notamment selon les habitudes des développeurs.

Figure 1.4 : *Choix d'une technologie de création d'interfaces graphiques*



Vue graphique ou Qt Quick ?

Lorsque la vue graphique (Graphics View) de Qt est apparue avec Qt 4.2, en 2006, il s'agissait du summum du modernisme pour afficher de grands nombres d'éléments à l'écran… puis Qt Quick est arrivé.

Ce nouveau venu fut, en réalité, une opportunité pour la vue graphique : la première version de Qt Quick l'utilisait pour son implémentation, guidait la danse pour les prochaines fonctionnalités et améliorations de performance. Depuis PyQt 5.0, cependant, Qt Quick n'utilise plus ce cadriciel pour son implémentation, mais bien un graphe de scène et un rendu effectué directement avec OpenGL. Ce n'était pas un aveu de faiblesse de la vue graphique : ce cadre a été très utile pour proposer une version de Qt Quick performante, avec un temps de développement très réduit. Néanmoins, il s'agissait d'une couche d'abstraction pas forcément utile pour l'utilisateur final de Qt Quick et elle ne permettait pas toutes les optimisations nécessaires pour les ambitions que nourrissaient les développeurs envers Qt Quick. Depuis ce changement radical, la vue graphique n'a pas beaucoup évolué, comme le montre l'historique Git.

Alors, que faut-il choisir : vue graphique ou Qt Quick ? Ce dernier n'est pas

forcément un remplaçant pour tous les emplois de la vue graphique : au contraire, il lui manque un certain nombre de fonctionnalités, plus ou moins importantes selon les cas d'utilisation.

- La vue graphique distingue clairement la vue (QGraphicsView) de la scène (QGraphicsScene) : une vue n'est qu'une manière de voir une scène (voir la Section 2, Scènes, éléments et vues). On peut ainsi disposer d'un grand nombre de vues d'une même scène, avec une très bonne performance.

  Au contraire, avec Qt Quick, cette fonctionnalité (pas si souvent utile, voire inutilisable dans bon nombre d'applications mobiles et embarquées) n'a pas été répliquée : pour y arriver, il faut impérativement créer plusieurs scènes, à garder synchronisées — ou alors utiliser Qt 3D (voir chapitre Affichage 3D avec Qt 3D) et plusieurs caméras. Ce faisant, le code de rendu de Qt Quick a pu être grandement simplifié et optimisé.

- Les QGraphicsItem disposent d'une solution de détection de collisions efficace, totalement absente de Qt Quick. C'est un réel handicap pour réaliser des jeux, vu que ces algorithmes doivent être implémentés pour chaque application — tout en faisant attention à la performance, souvent critique pour des jeux mobiles. L'infrastructure nécessaire est cependant arrivée avec Qt 3D (voir chapitre Affichage 3D avec Qt 3D).

- Qt Quick ne dispose, pour le moment, que d'une seule forme de base : le rectangle. Certes, il est très configurable (avec des bords arrondis, on peut dessiner des cercles, des ellipses), mais le niveau de fonctionnalité est très loin de celui proposé par la vue graphique et ses formes personnalisables à l'infini.

  On peut obtenir un résultat similaire avec le composant Canvas de Qt Quick émulant l'API HTML5 du même nom (c'est-à-dire que le dessin est piloté en JavaScript, avec les problèmes de performance qui l'accompagnent — voir le chapitre Affichage 2D avec Canvas), mais sans réelle intégration à l'environnement. On peut aussi créer ses propres composants Qt Quick en Python, voire créer ses propres nœuds pour le rendu dans le graphe de scène (ce qui apportera une performance maximale, mais avec un complexité d'implémentation très élevée).

- La vue graphique est entièrement pensée pour Python, il est très facile de créer ses propres classes. Au contraire, l'interface Python de Qt Quick n'est pas pensée pour l'extension : un très grand nombre de composants n'a qu'une

interface privée. Tout ce travail d'extension est censé être fait en QML (il est d'ailleurs encore plus aisé qu'en Python), mais la performance peut en pâtir.

En d'autres termes, il n'est pas facile d'y accéder depuis son propre code et, de toute façon, les développeurs de Qt ne garantissent aucunement sa stabilité : tout code qui utilise directement les composants Qt Quick en Python (par héritage, par exemple) n'a aucune garantie de continuer à fonctionner avec la prochaine version de Qt.

Cette décision a beaucoup de sens : l'utilisateur peut toujours faire ce qui lui plaît en QML (le code ne sera pas complètement cassé par une mise à jour), tout en laissant aux développeurs de Qt la liberté de faire évoluer l'API interne, notamment à cause du jeune âge de Qt Quick.

Quelles conclusions peut-on en tirer ? La vue graphique n'est plus l'objet de grandes attentions, mais est extrêmement stable et performante : elle est d'ailleurs utilisée dans un très grand nombre d'applications. Il n'est pas envisagé, pour le moment, de les retirer de Qt. Néanmoins, la politique actuelle de Qt indique clairement que Qt Quick est l'avenir pour la création d'interfaces graphiques : les nouvelles fonctionnalités de Qt sont d'ailleurs généralement utilisables aussi bien en Python qu'en QML (comme Qt 3D).

Ce livre ne pose pas de choix particulier : pour le moment, les deux approches ont du sens, selon le type d'application à développer. Ainsi, nous présentons tant la vue graphique (Affichage 2D interactif avec les vues graphiques) que Qt Quick (Développement d'une application avec Qt Quick). Vous pourrez ainsi vous faire votre propre avis sur chacune de ces deux solutions et voir laquelle est la plus appropriée à vos besoins.

***Note** > À titre de comparaison, PyQt propose la classe QPainter pour le dessin à l'écran (tout ce qui n'est pas possible avec les classes de base de Qt, comme des courbes, des graphiques scientifiques, etc.). Elle est déjà relativement ancienne (première moitié des années 2000 !), mais reste là : malgré l'apparition de solutions nettement plus évoluées, plus complexes (les vues graphiques et Qt Quick en particulier), les fonctionnalités de QPainter sont toujours là et elles ne devraient pas disparaître avant longtemps. Les changements sont rares, mais pas inexistants : courant 2017, elle sera adaptée aux technologies de rendu les plus modernes (une dépendance à OpenGL 2 limitée).*

4. Installation

Avant d'aller plus loin, il sera nécessaire d'installer Python et PyQt. L'installation de PyQt est plus compliquée que celle d'un module Python traditionnel à cause de la dépendance envers du code C++. Le plus simple est de recourir à une version déjà compilée de PyQt, sans passer par des étapes de compilation nombreuses et alambiquées.

***Note** > Ce livre utilisera PyQt5, non PyQt4 : notamment, toutes les distributions Linux ne proposent pas automatiquement la dernière version.*

Pour Windows et macOS, il est recommandé d'utiliser le gestionnaire de paquets de Python `pip`. Dans un terminal, il suffit de saisir la commande suivante :

```
pip3 install pyqt5
```

Parfois, `pip` n'est disponible que sous la commande `pip`, pas `pip3`.

Côté Linux, bon nombre de gestionnaires de paquets proposent directement PyQt5, dans un paquet souvent nommé `python3-pyqt5`. Par exemple, pour Ubuntu, dans une invite de commande :

```
$ sudo apt-get install python3-pyqt5
```

Pour Fedora :

```
$ sudo yum install python3-qt5
```

***Note** > Pour toutes les plateformes, l'installation d'extensions à PyQt se fait avec pip ; le nom de la bibliothèque pour pip n'est cependant pas toujours celui de la bibliothèque. Par exemple, pour toutes celles citées dans la Section 2.2, Bibliothèques, dans l'ordre, les commandes correspondantes sont :*

```
pip3 install Camelot
pip3 install guidata
pip3 install pyqtgraph
pip3 install guiqwt
pip3 install PyQtCharts
pip3 install PyQtDataVisualization
pip3 install QScintilla
pip3 install PyQtPurchasing
```

Sinon, une méthode universelle mais bien plus compliquée consiste à compiler soi-

même PyQt pour Python. Ceci n'a pas vraiment de sens quand des paquets précompilés existent, ce qui sera très probablement le cas. La documentation précise néanmoins la procédure à suivre, qui commence par l'installation des dépendances.

Environnement de développement

Niveau : débutant

Objectifs : utiliser eric6 pour développer des applications PyQt

Les applications Python peuvent être développées à l'aide d'un simple éditeur de texte. Ceci inclut les applications PyQt. Nous verrons dans ce livre qu'une application PyQt est créée en général à partir de plusieurs fichiers sources Python, de bibliothèques et de ressources (icônes, images, sons, textes traduits, etc.). Selon la complexité, le nombre de ces fichiers peut passer de un à plusieurs centaines. Il est dès lors utile d'avoir un environnement de développement qui facilite la structuration, l'édition, l'intégration et l'utilisation de ces fichiers. Il est bien entendu possible d'utiliser différents outils (éditeur, compilateur, débogueur, documentation, gestionnaire de versions, etc.) et de passer de l'un à l'autre au gré des besoins : c'est ce que faisaient les programmeurs avant (et ce qu'on peut faire encore, si l'on en a le courage). Il existe cependant aujourd'hui — depuis les années 80, en fait — des applications appelées *environnement de développement intégré* — EDI (ou, en anglais, *Integrated Development Environment* — IDE), qui ont pour vocation d'intégrer tous les outils de développement dans une même application.

Qu'est-ce qu'un EDI ?

Un EDI est une application qui intègre différents outils de développement et les rend accessibles depuis une même interface utilisateur, souvent graphique (GUI). Voici quelques-unes des fonctionnalités proposées par un EDI.

- *Structuration en projets* : tous les fichiers de l'application en développement sont rassemblés en un projet qui porte un nom et peut posséder des attributs tels que numéro de version, auteur(s), etc. L'EDI permet de créer un nouveau projet, de l'ouvrir, le fermer, etc. Il ne montre que les fichiers appartenant au projet courant ouvert. Il montre une vue arborescente et favorise la structuration par types de fichiers : fichiers sources (`*.py`), fichiers écran Qt Designer (`*.ui`), fichiers ressources — images, icônes (`*.png`, `*.gif`, `*.xml`, etc.).

- *Édition intelligente* : l'EDI permet d'éditer les fichiers sources pour un

langage de programmation donné en reconnaissant la syntaxe de ce langage : couplage des parenthèses, vérification syntaxique, coloration syntaxique (mise en gras de mots clés, couleurs identifiant les commentaires, noms de variables, méthodes, etc.). Certains éditeurs proposent aussi l'*autocomplétion* qui permet (notamment) de retrouver facilement le nom et les arguments d'une méthode dans un contexte donné.

- *Analyse statique du code* : des outils permettent de diagnostiquer dans un programme, avant même qu'il soit exécuté, certains problèmes comme l'importation d'un module non utilisé, une variable assignée mais jamais lue, une faille de sécurité, etc.

- *Exécution et débogage* : l'EDI permet de lancer facilement l'application en cours de développement via un bouton ou un raccourci clavier. En cas de problème lors de l'exécution (résultats erronés, exception inopinée, blocage, etc.), un débogueur permet de voir la ligne du programme en cours d'exécution, le contexte d'appel ainsi que le contenu des variables, et d'exécuter pas à pas les instructions.

- *Contrôle des versions* : la gestion des versions est un aspect important dans le développement moderne. Il s'agit non seulement de pouvoir gérer les différentes versions de l'application développée mais aussi de ses dépendances, c'est-à-dire les différents composants logiciels qu'elle utilise et leur version. Un autre aspect très important est le développement distribué, où une équipe de programmeurs travaille sur le même projet de manière coordonnée et décentralisée, ceci via un dépôt accessible sur le Web ; dans les cas extrêmes, notamment les gros projets open-source, la plupart des programmeurs ne se connaissent pas entre eux. Il existe plusieurs outils qui permettent de gérer tout cela. Les plus connus ont pour nom Subversion (SVN), Mercurial (Hg), Git. Ces outils s'utilisent à la base en ligne de commande ; l'EDI propose pour un ou plusieurs d'entre eux une interface graphique qui facilite l'accès à leurs fonctions : affichage des fichiers et lignes de code modifiés entre telle et telle version, contrôle et historique des changements faits sur le dépôt, assignation des versions, contrôle d'accès des développeurs, gestion des branches, etc.

  ***Note** > L'acronyme anglais VCS, pour Version Control System, est fréquemment rencontré dans les EDI.*

- *Conception GUI* : beaucoup de bibliothèques graphiques proposent un éditeur permettant de dessiner l'interface graphique, c'est-à-dire des fenêtres avec

leurs menus, boutons, listes, cases à cocher, etc. Ceci permet en général de soulager le programmeur d'une tâche laborieuse qui nécessite, sauf pour les plus chevronnés, une approche par essais et erreurs. De plus, un tel éditeur graphique permet de faire du prototypage rapide pour discuter avec les utilisateurs de la future application ; on peut ainsi, sans écrire une seule ligne de code, montrer une maquette réaliste de l'application et faire, au besoin, les adaptations requises. L'intégration d'un tel outil dans un EDI ne se limite pas au dessin proprement dit ; il faut que les fenêtres dessinées puissent être facilement exploitables lors du codage de l'application, ce qui peut nécessiter une phase de génération automatique de code.

- … et encore beaucoup d'autres fonctions : factorisation du code, tests, couverture du code, profilage, gestionnaire de tâches, gestion des traductions, etc.

Il existe actuellement une pléthore d'EDI, chacun dédié, en général, à un langage de programmation particulier ; chacun couvre à sa manière, et avec plus ou moins de bonheur, les différentes fonctions évoquées précédemment. Python n'échappe pas à la règle et, depuis la création du langage fin des années 1980, beaucoup d'EDI ont fleuri : IDLE, eric, Wing IDE, PyCharm, PyDev, Ninja-IDE, Spyder, pour n'en citer que quelques-uns. Il est difficile d'élire un meilleur EDI, car il y a évidemment plusieurs critères de sélection, critères dont l'importance varie d'un type d'utilisation à l'autre ; par ailleurs, il y a bien entendu la question des licences et du prix. Nous n'allons pas, dans cet ouvrage, analyser ces EDI ni vanter leurs mérites respectifs. Plutôt que de rester dans une neutralité prudente, il nous a semblé plus intéressant de choisir un EDI particulier comme support aux exemples.

Notre choix s'est porté sur… eric6 !

1. Pourquoi eric6 ?

Le choix d'eric6 comme EDI est justifié essentiellement par le fait qu'il est gratuit et offre une bonne intégration avec PyQt. Ceci nous a paru être deux critères importants dans le contexte du présent livre qui vise l'apprentissage de PyQt. De plus, eric6 a des caractéristiques intéressantes qu'on ne trouve pas forcément dans tous les EDI :

- il est relativement riche au niveau de ses fonctionnalités, avec un système d'extensions à la carte ;
- il supporte Python 2 et 3 ;
- et... il est écrit en Python avec PyQt !

Le fait qu'eric6 soit écrit à l'aide de PyQt n'est pas un point essentiel en soi, mais il démontre que PyQt est un outil puissant qui permet de développer des applications complexes, comme le sont les EDI.

2. Installation et configuration d'eric6

Les prérequis pour installer eric6 sont l'installation de Python, QScintilla, Qt et PyQt (voir les versions sur la page de téléchargement). Les utilisateurs sous Windows noteront que l'installation de PyQt sous forme de binaires (*Binary Packages*) inclut déjà Qt et QScintilla : tout est prêt pour installer eric6.

La page de téléchargement d'eric6 contient un lien vers une page SourceForge sur laquelle on peut choisir un dossier avec la dernière version d'eric6 (disons `x.y.z`); téléchargez le fichier d'installation compressé `eric6-x.y.z.tar.gz` et, si vous voulez l'interface en français, le fichier `eric6-i18n-fr-x.y.z.tar.gz` (pour les utilisateurs de Windows, ces fichiers existent aussi au format `.zip`).

Après décompression, on obtient un répertoire `eric6-x.y.z/`. Pour disposer d'une interface en français, il faut avant toute chose placer les traductions françaises dans ce répertoire : copiez le contenu décompressé de `eric6-i18n-fr-x.y.z/eric6-x.y.z/eric/i18n/` (deux fichiers) dans `eric6-x.y.z/eric/i18n/`.

L'installation proprement dite d'eric6 est faite en lançant simplement, sur la ligne de commande :

```
python install.py
```

Attention *> eric6 s'installe dans le sous-répertoire `lib/site-packages/` du répertoire racine Python, à côté du répertoire `PyQt/`. Si vous avez plusieurs versions de Python installées, il est impératif d'appeler le bon interpréteur, c'est-à-dire celui pour lequel PyQt a été installé. On peut vérifier la version en exécutant au préalable la commande `python -V`. Si la version n'est pas celle attendue, il y a lieu, par exemple, d'appeler `python3` ou de changer la variable d'environnement `PATH` en indiquant le répertoire principal de la version requise de Python.*

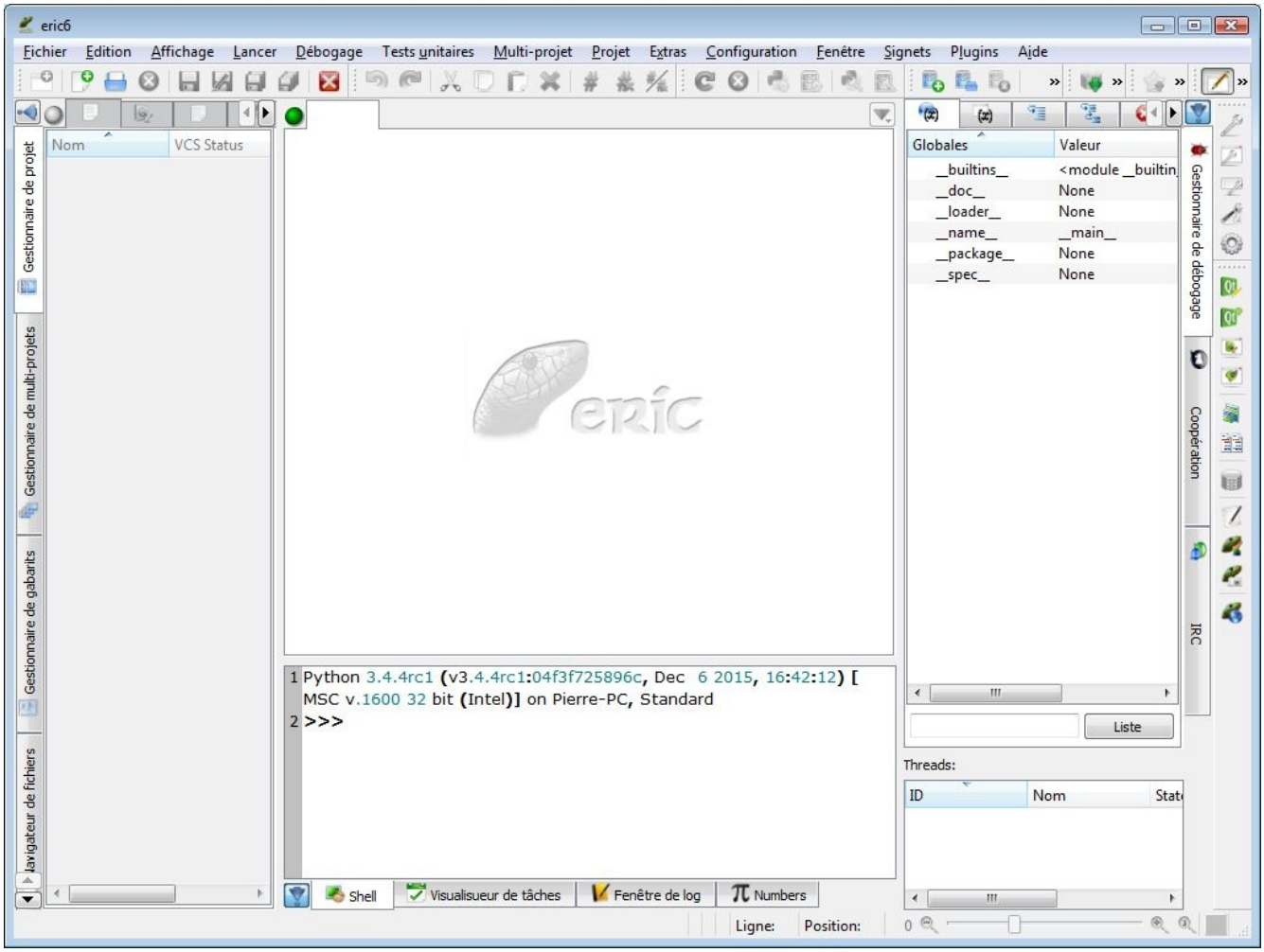
Après installation, pour lancer eric6 il suffit d'appeler :

```
eric6
```

Pour information, cette commande lance en fait un script `eric6` (ou `eric6.bat` sous Windows) situé dans `Python/Scripts/`. Ce chemin est supposé être dans `PATH`. Ceci est utile à savoir pour créer un raccourci sur le bureau ou... si la commande `eric6` ne fonctionne pas.

Au démarrage, la fenêtre principale d'eric6 devrait ressembler à ceci.

Figure 2.1 : *Fenêtre principale d'eric6*



Note *> Si l'application s'affiche en anglais, il est possible de changer la configuration pour passer en français : activez le menu Plugins > Configure... allez dans la rubrique Interface/Interface et choisissez Français (fr) dans la boîte de sélection Language ; redémarrez ensuite eric6.*

Une caractéristique commune à beaucoup d'EDI est que l'interface utilisateur est très chargée, avec une multitude de menus, d'onglets, de boutons aux noms parfois ésotériques. Ceci est souvent déconcertant pour l'utilisateur débutant. eric6 n'échappe pas à la règle. Heureusement, cette interface est configurable et il est possible de cacher pas mal de choses qu'on n'utilise pas (ou pas encore). Voici quelques options pratiques et faciles à appliquer :

- les éléments latéraux et en bas possèdent un bouton avec une flèche et un point

(infobulle : Décocher pour activer le repliement automatique). Ce bouton peut être coché (flèche vers le bas) ou décoché (flèche vers la gauche) : si le bouton est décoché, l'élément graphique correspondant n'apparaît que si le curseur souris passe sur la zone en question ;

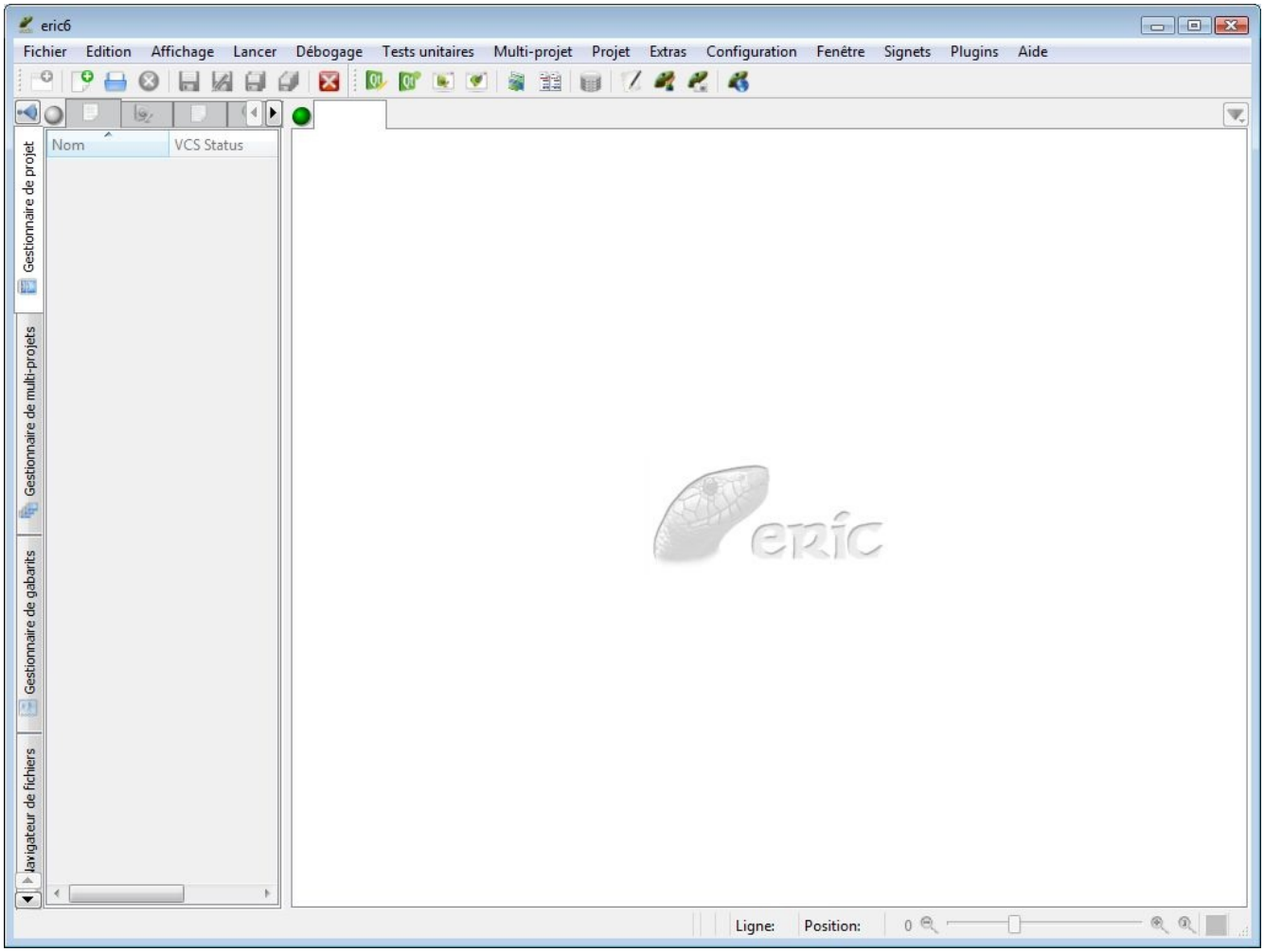
- la boîte de dialogue Profils de visualisation accessible par un bouton sur la barre de droite ou dans le menu Configuration permet de décocher les éléments qu'on veut cacher ;

- les barres d'outils peuvent être déplacées par groupe en faisant glisser les poignées, reconnaissables par six petits points alignés horizontalement. On peut aussi changer l'orientation d'une barre verticale en la faisant glisser de la gauche vers le haut de la fenêtre, et inversement ;

- les barres d'outils peuvent être affichées/cachées par groupe en cliquant-droit sur n'importe quelle barre d'outils : un menu apparaît qui permet de cocher/décocher les barres.

Par exemple, voici une configuration eric6 allégée, obtenue en décochant les deux dernières cases du Mode Edition dans Profils de Visualisation, en décochant toutes les barres d'outils à part Fichier et Outils et en déplaçant Outils sur la barre horizontale.

Figure 2.2 : *Interface eric6 après configuration*



Cette configuration n'est bien entendu qu'une suggestion : vous ferez votre choix selon vos goûts et en fonction des outils que vous utilisez le plus couramment.

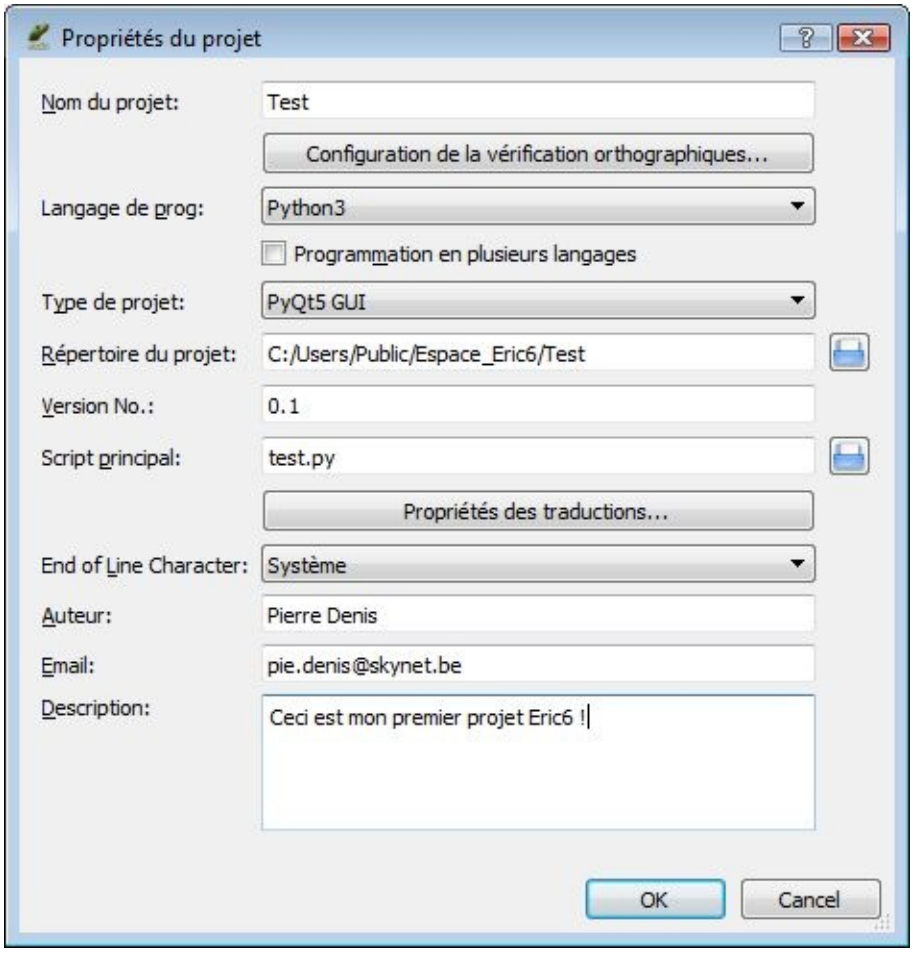
Nous allons à présent créer notre premier projet eric6, qui sera une application minimale PyQt. Il permettra de se familiariser avec l'EDI et de s'assurer que l'installation de PyQt est correcte.

3. Premier projet

Un projet eric6 est un ensemble de fichiers qui permet de créer une application : fichiers sources Python, fichiers écrans, images, données de configuration, etc. Un projet possède différents attributs (nom, auteur, version, etc.). Il contient au minimum un fichier Python appelé *script principal* ; il s'agit du programme qui doit être exécuté au démarrage de l'application ; c'est en quelque sorte le point d'entrée. Dans une application bien construite, ce programme est assez court : il consiste à initialiser l'application et faire appel à des modules Python écrits dans d'autres fichiers (par exemple, un fichier pour la fenêtre principale qui va éventuellement lui-même utiliser d'autres fichiers pour définir ses composants). Une approche modulaire est essentielle pour tout développement d'application conséquente.

Pour créer un nouveau projet dans eric6, il faut activer le menu Projet > Nouveau.... Une boîte de dialogue Propriétés du projet est alors affichée. Nous allons appeler ce projet `Test` et le définir de façon similaire à ce qui suit :

Figure 2.3 : *Boîte de dialogue de création d'un projet eric6*



Note *> Le champ `Répertoire du projet` peut différer de celui qui est indiqué, pourvu qu'il désigne un répertoire accessible en écriture. Notons qu'il FAUT saisir un sous-répertoire dans le répertoire d'espace de travail d'eric6 (`/Test` dans l'exemple), sans quoi le bouton OK n'est pas accessible.*

Après avoir cliqué sur le bouton OK, le projet `Test` est sauvegardé ; deux fichiers Python sont créés et affichés dans la zone Gestionnaire de projet à gauche :

- `test.py` : le script principal, initialement vide ;
- `__init__.py` : un script technique spécifique à Python (donc, non lié à eric6 ni à PyQt) qui permet de *marquer* le répertoire du projet comme un *package* au sens Python ; ceci permet d'organiser les fichiers de manière hiérarchique et d'importer des modules avec la syntaxe `mon_projet.mon_package.mon_sous_package.mon_module`. Dans la suite, nous ignorerons ce fichier.

À présent, vous pouvez double-cliquer sur `test.py` pour ouvrir le fichier en édition

sur la partie centrale et saisir les lignes qui suivent :

```
import sys
from PyQt5.QtWidgets import *

app = QApplication(sys.argv)
mainWindow = QMainWindow()
QLabel("Bonjour le monde !",mainWindow)
mainWindow.show()
app.exec_()
```

Pour sauvegarder, on utilise les moyens habituels : activation du menu Fichier > Enregistrer, le bouton Disquette ou le raccourci clavier Ctrl+S.

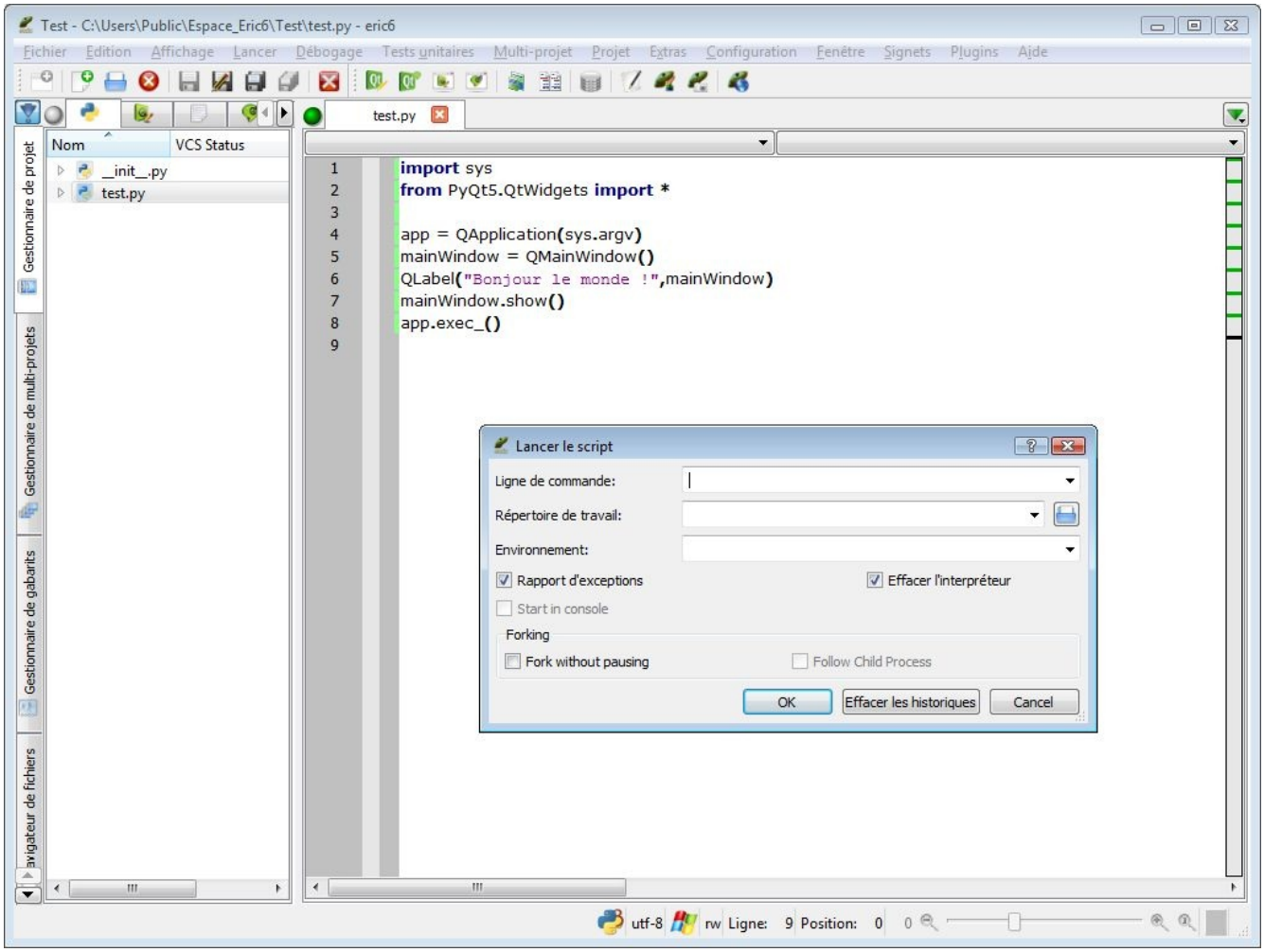
Attention *> N'oublions pas que nous sommes en Python : l'indentation et la casse (majuscule/minuscule) doivent être respectées strictement. Notons que, lors de la sauvegarde, l'éditeur eric6 détecte les erreurs de syntaxe les plus flagrantes et, le cas échéant, les signale en affichant un insecte peu sympathique, à gauche de la ligne erronée.*

Nous avons à présent un projet eric6 qui contient une application PyQt minimale. Ce projet pourra être rouvert plus tard, en activant le menu Projet > Ouvrir... ou Projet > Ouvrir un projet récent.

4. Première exécution

Pour tester l'application, il existe plusieurs méthodes. La plus simple à ce niveau consiste à activer le menu Lancer > Lancer projet... (Maj+F2). Ceci fait apparaître une boîte de dialogue (voir Figure 2.4) qui permet de configurer les paramètres d'exécution.

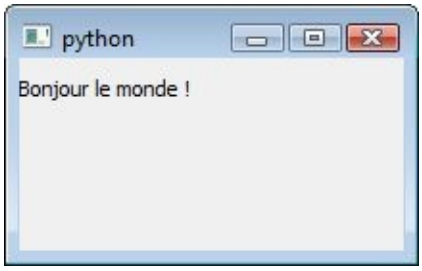
Figure 2.4 : *Lancement de l'application*



Cette boîte de dialogue permet notamment de simuler des arguments en ligne de commande qui pourront, si l'application le requiert, être récupérés via la variable Python `sys.argv`. Dans le cas présent, nous n'allons rien saisir et cliquer simplement sur OK. eric6 va alors lancer le script `test.py`, sachant que c'est lui qui a été configuré comme script principal du projet `Test`. Si tout se passe bien, la fenêtre

principale de l'application devrait apparaître :

Figure 2.5 : *Application PyQt minimale*



Si cette fenêtre n'apparaît pas ou si un message d'erreur est signalé, vérifiez votre configuration. La ligne la plus critique est la ligne 2, qui est censée importer des classes de PyQt. Si une erreur telle que *No module named 'PyQt5'* apparaît, cela signifie que l'installation de PyQt5 est incorrecte ou mal configurée. Vérifiez également la configuration du projet eric6 ; cette configuration peut être éditée via le menu Projet > Propriétés....

Astuce *> Après une première exécution, les paramètres de lancement éventuellement saisis dans la boîte de dialogue Lancer le script sont mémorisés ; il est alors possible de lancer l'application directement, en sautant cette étape : il suffit d'activer le menu Lancer > Restart (raccourci clavier F4). Ce sera le plus rapide en phase de test.*

Comme on peut s'y attendre, notre programme ne permet pas de faire grand-chose à part les opérations standards comme déplacer la fenêtre, la redimensionner et la fermer. Ceci dit, l'effort pour y arriver est minimal (sept courtes lignes de code), ce qui est bien en deçà de ce qui aurait été nécessaire avec bon nombre d'autres bibliothèques graphiques. Notons aussi que les étapes de configuration ne concernent en définitive qu'eric6, qui est dans le cas présent un luxe inutile, utilisé seulement pour l'apprentissage de l'EDI. Il est d'ailleurs tout à fait possible d'exécuter l'application sans eric6 : il suffit d'ouvrir un terminal, de se placer dans le répertoire du projet `Test` et de lancer la commande :

```
python test.py
```

La fenêtre de l'application s'ouvre comme avant. Cette simplicité offerte nativement par PyQt contraste avec l'approche d'autres bibliothèques pour lesquelles on a besoin d'un EDI ou d'un outil annexe pour obtenir le même niveau de simplicité (configuration des `Makefile`, variables d'environnement, etc.). Ceci s'inscrit dans la philosophie de Python à laquelle, nous le verrons dans cet ouvrage, PyQt souscrit pleinement.

Distribuer une application PyQt

Niveau : avancé

Objectifs : être capable de distribuer une application PyQt

Prérequis : connaissances de base en administration système

Une fois PyQt installé, lancer une application est très simple pour le développeur, indépendamment du système d'exploitation : il suffit de lancer l'interpréteur Python sur le programme principal. Ceci contraste avec les environnements basés sur des langages compilés (C, C++, Java, etc.), qui requièrent une configuration plus ou moins complexe pour compiler les différents fichiers sources et construire un exécutable.
L'environnement tel que Python/PyQt a donc clairement l'avantage quand il s'agit du développement pur, en particulier celui de prototypes : le programmeur peut laisser de côté les aspects techniques de bas niveau et se concentrer sur les fonctionnalités et l'interface utilisateur. Cependant, la vocation d'une application est de pouvoir être distribuée, installée et exécutée dans des environnements qui sont différents de celui utilisé pour le développement.

Machine hôte, machine cible et croisement de plateforme

Quand on parle de distribution d'application, il est important de bien comprendre les rôles des machines et systèmes d'exploitation impliqués dans le processus :

- la machine *hôte* (en anglais, *host*) est celle où tourne l'outil de préparation de la distribution ;
- la machine *cible* (en anglais, *target*) est celle destinée à faire tourner l'application distribuée.

Par extension, on parlera de plateformes hôte/cible pour désigner à la fois le type de machine (processeur, 32/64 bits) et le système d'exploitation.

Il est bien entendu possible que les plateformes hôte et cible soient les mêmes ; c'est même imposé par certains outils (voir Section 3.1, PyInstaller). Quand les plateformes hôte et cible diffèrent, on parle de *croisement de plateformes* (en

anglais, *cross-platform*).

1. Que recouvre la distribution d'une application ?

Plusieurs questions se posent pour la distribution d'une application PyQt sur une ou plusieurs machines cibles.

- Installation de Python : la machine cible a-t-elle une (ou plusieurs) versions de Python installées ? Si oui, y a-t-il une version qui est compatible avec l'application ? Si non, va-t-on installer Python soi-même ?

- Installation de PyQt : la machine cible a-t-elle une (ou plusieurs) versions de PyQt installées ? Si oui, y a-t-il une version qui est compatible avec l'application ? Si non, va-t-on l'installer soi-même ?

- Multiplateformes : compte-t-on installer l'application sur plusieurs systèmes d'exploitation (Windows, Linux, macOS, Android, etc.) ? Y a-t-il des dépendances liées à une plateforme particulière ? [1]

- Droits : quels sont les droits dont on dispose sur la machine cible ? Dispose-t-on des droits administrateur pour *installer* l'application ? Dispose-t-on des droits administrateur pour *exécuter* l'application ?

- Audience : qui procède à l'installation ? L'application est-elle sur mesure et destinée à quelques utilisateurs ou est-elle destinée à être téléchargée par tout un chacun ?

Attention *> Il est souvent important d'isoler la version de Python utilisée par l'application : certains systèmes d'exploitation ont une version de Python installée nativement et certains outils système pourraient ne plus fonctionner après installation d'une autre version de Python (ceci a été malheureusement expérimenté par les auteurs !).*

---

[1] Python et PyQt sont en principe indépendants de la plateforme. Il existe cependant certains modules ou bibliothèques Python qui ne tournent que sur une plateforme particulière. Par exemple, les modules standards Python winreg et winsound ne sont

supportés que sur Windows. Quand vous utilisez un module ou une bibliothèque, vous devez être conscient de ces dépendances, indiquées normalement dans la documentation — et, autant que possible, les éviter !

2. Distribution par copie des sources

L'idée la plus immédiate pour déployer une application PyQt est d'installer le même environnement que celui utilisé pour le développement, sans un EDI tel qu'eric6. Il faut donc :

1. installer une version de Python adéquate, si elle n'est pas déjà installée ;
2. installer la version adéquate de PyQt, si elle n'est pas déjà installée ;
3. copier les fichiers de l'application nécessaires : sources Python, images, etc.

La distribution consiste donc à fournir une procédure d'installation, avec les identificateurs de version et points d'entrée nécessaires pour l'installation de Python et PyQt. Ce mode de distribution peut convenir dans le cadre d'application sur mesure, où le développeur (qui connaît bien l'environnement parce qu'il l'a lui-même installé) peut lui-même procéder à l'installation sur la ou les machine(s) cible(s). Il est en revanche peu attirant pour une distribution massive car il nécessite des compétences que ne possèdent pas forcément les utilisateurs potentiels.

Quels sont le pour et le contre de la distribution par copie des sources ?

Le POUR :

- pas besoin d'apprendre un nouvel outil !
- même distribution pour toutes les plateformes ;
- paquet de distribution limité à l'application seule : peut donc être très léger ;
- maintenance facilitée : on peut mettre à jour l'application sans toucher au reste.

Le CONTRE :

- pas de paquet tout-en-un ;
- complexité d'installation : nécessite de faire une installation de Python et PyQt, qui est dépendante de la plateforme ;

- gaspillage d'espace disque : installation de beaucoup plus de modules que nécessaire pour une stricte exécution ;

- risque de casser des applications existantes sur la machine cible (ex : changement de la version de Python par défaut).

En résumé, on peut dire que la distribution par copie des sources facilite l'étape de distribution proprement dite … au détriment de l'étape d'installation.

Note *> Une alternative pour limiter les inconvénients de cette approche consiste à utiliser la virtualisation : une machine virtuelle (VM) destinée à exécuter l'application peut être configurée une fois pour toutes avec, typiquement, le système d'exploitation que le développeur connaît bien. Il suffit alors de distribuer l'image de cette VM, qui pourra tourner sur une machine hôte ayant éventuellement un système d'exploitation différent. Cette approche nécessite bien entendu une connaissance des techniques de virtualisation et l'installation d'un produit ad-hoc (VMware, VirtualBox, VirtualPC, KVM, QEMU, Xen, etc.) sur la machine hôte et sur la machine cible. Il faut aussi être attentif à la taille des images VM qui peut atteindre, si on n'y prend pas garde, plusieurs dizaines de gigaoctets. Pour ces raisons, la distribution par VM reste en général limitée à des applications sur-mesure, à déployer en petit nombre.*

3. Distribution par outils

Il est heureusement possible de distribuer une application Python comme un paquet autonome à destination d'une machine cible qui ne contient pas (forcément) Python. Le terme consacré est le *freezing* : on va figer et empaqueter tout ce qui est nécessaire à l'exécution, y compris l'environnement Python et PyQt. À l'inverse de l'approche vue précédemment, on n'a plus besoin d'avoir Python/PyQt installés sur la machine cible ; de plus, on est sûr d'avoir les bonnes versions, quelles que soient celles éventuellement déjà installées sur la machine cible ; on est sûr aussi de ne pas interférer avec celles-ci. À partir du paquet d'installation, qu'il soit copié ou téléchargé, l'effort d'installation pour l'utilisateur est minimal et surtout… standard !

Dans l'écosystème Python, il existe plusieurs outils pour faire cela : cx_Freeze, bbFreeze, py2exe, PyInstaller et py2app. Chacun présente ses particularités (support Python 3, multiplateformes, licence, etc.). Parmi ces outils, nous avons choisi de présenter en détail PyInstaller qui a le mérite d'être multiplateforme, de supporter PyQt sans effort et d'être très simple d'utilisation.

Par ailleurs, Riverbank Computing Limited (la société qui développe PyQt) propose aussi un outil spécifique pour PyQt : pyqtdeploy. Beaucoup plus complexe d'utilisation, il vous sera présenté succinctement. Précisons tout de go que pyqtdeploy a une philosophie très différente des outils classiques de freezing évoqués plus haut ; il est plus difficile à mettre en œuvre que PyInstaller, mais il permet notamment le croisement des plateformes comme nous le verrons plus loin.

3.1. PyInstaller

PyInstaller est un outil de freezing Python qui a les caractéristiques suivantes :

- il supporte Python 2.7 et 3.3+ ;
- il découvre automatiquement les modules à inclure à partir du programme principal Python ;
- il supporte nativement nombre de bibliothèques externes, telles que Django, maplotlib, numpy, pandas et, surtout, PyQt (4 et 5) ;

- il est multiplateforme : PyInstaller peut s'utiliser sur Windows, Linux, macOS, FreeBSD, Solaris et AIX ;

  Attention *> Le mot multiplateforme n'est pas synonyme de croisement de plateformes : pour une application donnée, il n'y a qu'un seul paquet généré, qui n'est valide que sur le système d'exploitation où tourne PyInstaller. On verra que ceci n'est pas le cas avec pyqtdeploy, ce qui est donc un des atouts de ce dernier.*

- il est tout à fait libre quant à son utilisation, même pour une application commerciale (licence GPL avec exceptions).

L'installation de PyInstaller est aisée. Parmi les méthodes possibles, vous pouvez utiliser la commande `pip3` en ligne de commande :

```
pip3 install pyinstaller
```

Note *> L'installation avec `pip3` est notamment recommandée pour Windows, car elle installe automatiquement la bibliothèque `PyWin32`, qui est un prérequis pour cette plateforme.*

Un des mérites de PyInstaller est sa simplicité d'utilisation. Voyons comment on peut distribuer l'application BiblioApp développée dans le cadre de ce livre (voir Développement d'une application avec des widgets ou Développement d'une application avec Qt Quick).

PyInstaller propose deux options principales de distribution :

- distribution en un fichier exécutable (*one-file*) ;

- distribution en un répertoire (*one-folder*).

Dans les deux cas, comme PyInstaller fonctionne en ligne de commande, vous aurez besoin d'une console. Ensuite, comme il faudra spécifier quel est le programme principal de l'application, le plus simple est de se placer dans le répertoire qui contient le programme principal de l'application, par exemple :

```
cd espace_eric_6/biblio_app
```

Enfin, la commande à lancer dépend de l'option choisie, comme expliqué dans les deux sections qui suivent.

Attention *> Les commandes qui suivent créent les répertoires `build/` et `dist/` dans*

le répertoire courant. En cas de collision de noms, il suffit de spécifier d'autres noms ou chemins dans les options respectives `--workpath` *et* `--distpath` *de PyInstaller.*

Distribution en un fichier exécutable

Pour distribuer l'application BiblioApp en un seul fichier exécutable, du nom `biblioapp` par exemple, saisissez la commande :

```
pyinstaller -F -n biblioapp start_biblio_app.py
```

Après quelques messages un peu ésotériques, un fichier exécutable `biblioapp` (ou `biblioapp.exe` pour Windows) devrait être disponible dans le sous-répertoire `dist/` créé par PyInstaller. Notons que cet exécutable a une taille conséquente (23 Mo dans notre test sous Windows), de loin supérieure à la taille de l'application. C'est assez logique : il inclut toutes les bibliothèques requises pour faire tourner Python et PyQt. Si cette taille est jugée prohibitive, vous pouvez faire appel à UPX, un utilitaire de compression d'exécutables ; référez-vous à la documentation de PyInstaller qui explique comment mettre cet utilitaire en œuvre.

Nous pouvons à présent lancer cet exécutable et constater que la fenêtre principale apparaît et que tout fonctionne comme dans l'environnement de développement. Notons toutefois qu'une console est lancée en sus de la fenêtre principale ; elle peut être très utile si l'application écrit des messages sur la sortie standard, via l'instruction `print` par exemple. Il faut avertir l'utilisateur de ne pas fermer cette console (qui est en fait le processus parent de l'application PyQt), auquel cas la fenêtre principale serait elle-même fermée brutalement. Si on ne veut pas faire apparaître cette console, il faut l'indiquer à PyInstaller en passant l'option `-w` ou `--noconsole` sur la ligne de commande : l'exécutable alors généré n'affichera que la fenêtre principale au démarrage.

Attention *> L'option d'exécutable sans console est fortement déconseillée dans le cas d'une application qui écrit sur la sortie standard. Les messages, qui n'ont aucun moyen de s'afficher, sont mis dans une zone tampon qui n'est jamais purgée. Au mieux, on gaspille de la mémoire ; au pire, l'application peut planter lorsque cette zone tampon est remplie.*

L'exécutable qui vient d'être créé est prêt à être distribué pour la plateforme hôte. Comme signalé plus haut, si on veut des exécutables pour d'autres plateformes, il faut avoir des machines (éventuellement virtuelles) avec les systèmes d'exploitation cibles ; on doit alors y avoir installé Python, PyQt et PyInstaller et, bien sûr, avoir accès au

répertoire de l'application (par copie ou, mieux, par partage du système de fichiers). Pour plus de détails, vous pouvez vous référer à la documentation de PyInstaller pour le croisement de plateformes.

La technique de création d'un fichier exécutable qu'on vient de voir est évidemment la plus simple pour la distribution et l'utilisation. En revanche, on obtient un gros fichier monolithique. Ceci présente deux désavantages :

- à chaque exécution, tout l'exécutable doit être (re-)lu et les bibliothèques qu'il contient doivent être (re-)chargées : on perd tout l'intérêt des bibliothèques dynamiques par rapport aux bibliothèques statiques ;

- l'application est une boîte noire, qu'on peut difficilement ouvrir ; on ne peut pas avoir un fichier de configuration éditable à la main, qui serait bien utile ; de plus, à chaque nouvelle version, il faut redistribuer tout l'exécutable alors qu'on sait pertinemment que seul le code qu'on a écrit a changé, ce qui représente une fraction minime du fichier.

La technique de distribution en un répertoire que nous allons voir à présent pallie ces défauts.

Distribution en un répertoire

Pour distribuer l'application BiblioApp en un répertoire, saisissez la commande :

```
pyinstaller -D -n biblioapp start_biblio_app.py
```

Après exécution, vous aurez un nouveau répertoire `dist/biblioapp/` qui contient un exécutable `biblioapp` (ou `biblioapp.exe` pour Windows) ainsi que d'autres fichiers ou sous-répertoires. On peut vérifier que l'exécutable tourne comme avant. Le répertoire `dist/biblioapp/` contient tout ce qui est nécessaire à l'exécution ; ce répertoire peut donc être compressé, empaqueté et distribué selon les modalités du système d'exploitation (fichier `zip`, `tar.gz`, etc.). La discussion vue à la section précédente sur la console ainsi que l'option pour la désactiver restent applicables ici.

On peut remarquer que la taille du fichier exécutable, qui ne contient plus que le code développé, est considérablement réduite par rapport à celui généré avec l'option `-F` (1,2 Mo dans notre test sous Windows). Il est donc possible, une fois que le répertoire complet a été déployé sur la machine cible, de faire des maintenances en ne livrant *que* le petit exécutable de l'application proprement dite, qu'il suffira de substituer à

l'exécutable précédent.

3.2. pyqtdeploy

Passons à présent à pyqtdeploy, l'outil proposé par les créateurs de PyQt. L'approche est très différente de celle de PyInstaller, qui est un outil généraliste pour toute application Python. pyqtdeploy nécessite une phase de configuration préalable pour fonctionner, ce qui le rend plus compliqué d'emploi que PyInstaller. L'atout de pyqtdeploy est de permettre le croisement de plateformes : il peut, à partir d'un seul environnement, préparer des exécutables pour Windows, Linux, macOS et même les plateformes mobiles Android et iOS.

Pour utiliser pyqtdeploy, vous aurez besoin :

- d'un compilateur C++ : par exemple, gcc/g++ (Linux), Xcode (macOS), Visual Studio Express ou Community (Windows) ;
- de Python 3.2 minimum, avec ses sources C ;
- de PyQt5 ;
- de Qt 5.3 minimum.

Pour installer pyqtdeploy, ouvrez une console et lancez la commande :

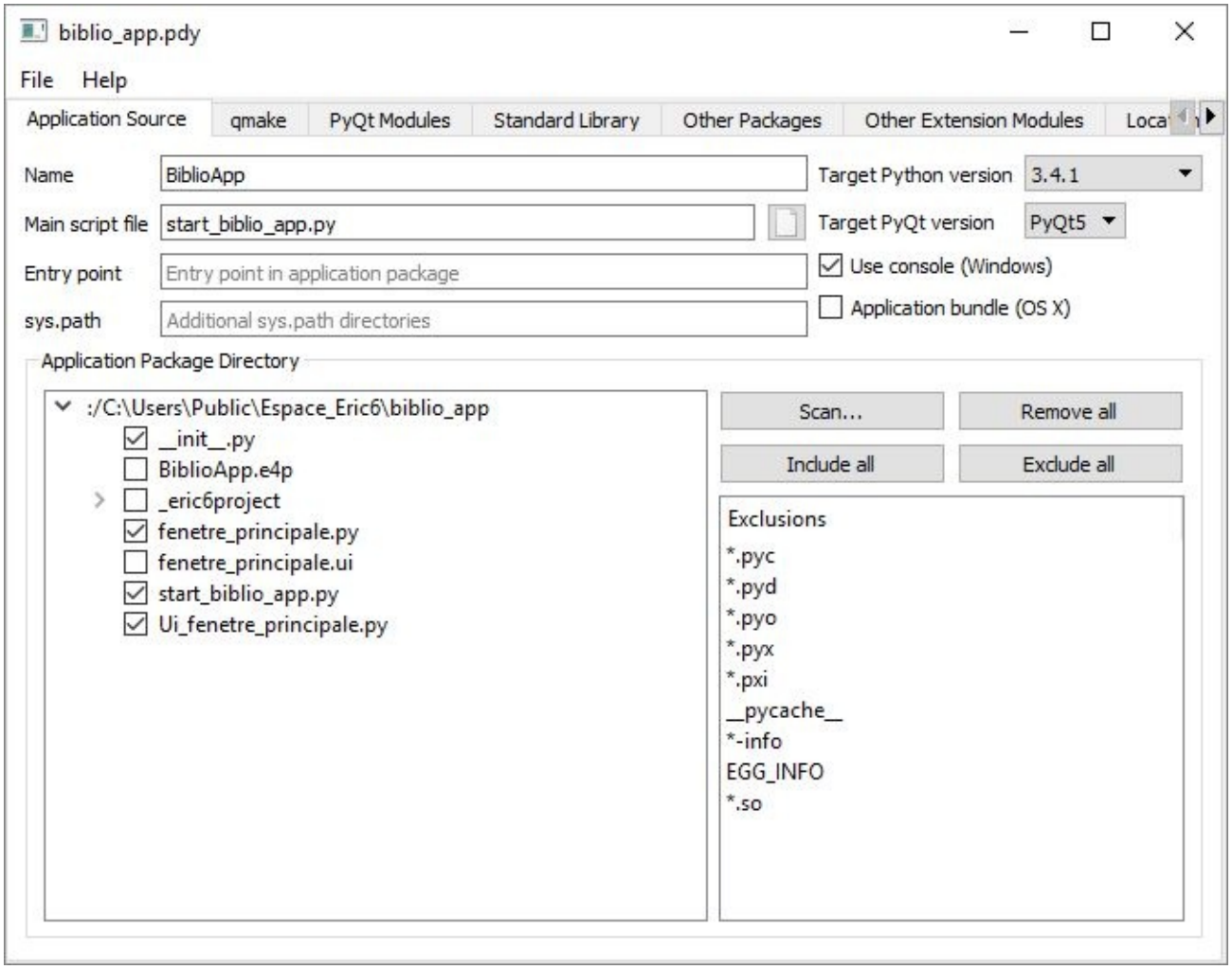
```
pip3 install pyqtdeploy
```

La configuration de pyqtdeploy pour un projet donné est stockée dans un fichier d'extension `.pdy`. Pour le créer ou l'éditer, lancez l'interface utilisateur de pyqtdeploy en lui donnant le nom du fichier `.pdy` en argument :

```
pyqtdeploy biblio_app.pdy
```

L'interface est composée de plusieurs onglets qui permettent de spécifier tout ce qui est nécessaire à pyqtdeploy pour travailler.

Figure 3.1 : *pyqtdeploy — interface utilisateur*



Le menu File permet de sauvegarder le fichier `.pdy` courant. Référez-vous à la documentation de pyqtdeploy pour définir les données adéquates. Pour définir cette configuration, prenez garde à bien distinguer dans la documentation machine hôte et machine cible, tel qu'expliqué plus haut.

Le dernier onglet, nommé Build, contient un bouton du même nom qui permet de générer le fichier `.pro`. Les trois cases à cocher sous Additional Build Steps permettent d'enchaîner les autres actions requises, à savoir : invoquer qmake pour générer le fichier `Makefile`, invoquer make pour créer l'exécutable de l'application et, si on ne fait pas de croisement de plateforme, lancer l'exécutable pour vérifier que l'application fonctionne bien. En cas de problème pour ces étapes, il est conseillé de passer sur la console et de lancer ces outils à la main : on peut alors mieux voir où se situent les problèmes et corriger ce qu'il faut (typiquement, les variables d'environnement).

Comment ça marche ?

Plutôt que de réinventer la roue, pyqtdeploy crée un pont qui permet d'utiliser l'infrastructure offerte par Qt, la bibliothèque C++ sur laquelle est construite PyQt. Concrètement, pyqtdeploy génère des fichiers C++ qui encapsulent le code Python, aussi bien les modules standards que l'application développée en PyQt ; ensuite, il crée un fichier `.pro` , à partir duquel l'outil qmake de Qt peut générer un fichier `Makefile` pour une plateforme donnée (ceci est une étape connue des lecteurs familiers de Qt) ; enfin, ce dernier fichier est lui-même utilisé par l'outil général make pour créer l'exécutable final. Par expérience, il faut noter que ces étapes nécessitent une certaine connaissance de la méthode de déploiement de Qt et de la compilation C++ en général. Fort heureusement, pyqtdeploy propose une interface utilisateur qui facilite (un peu) la configuration.

Développement d'une application avec des widgets

Tour d'horizon

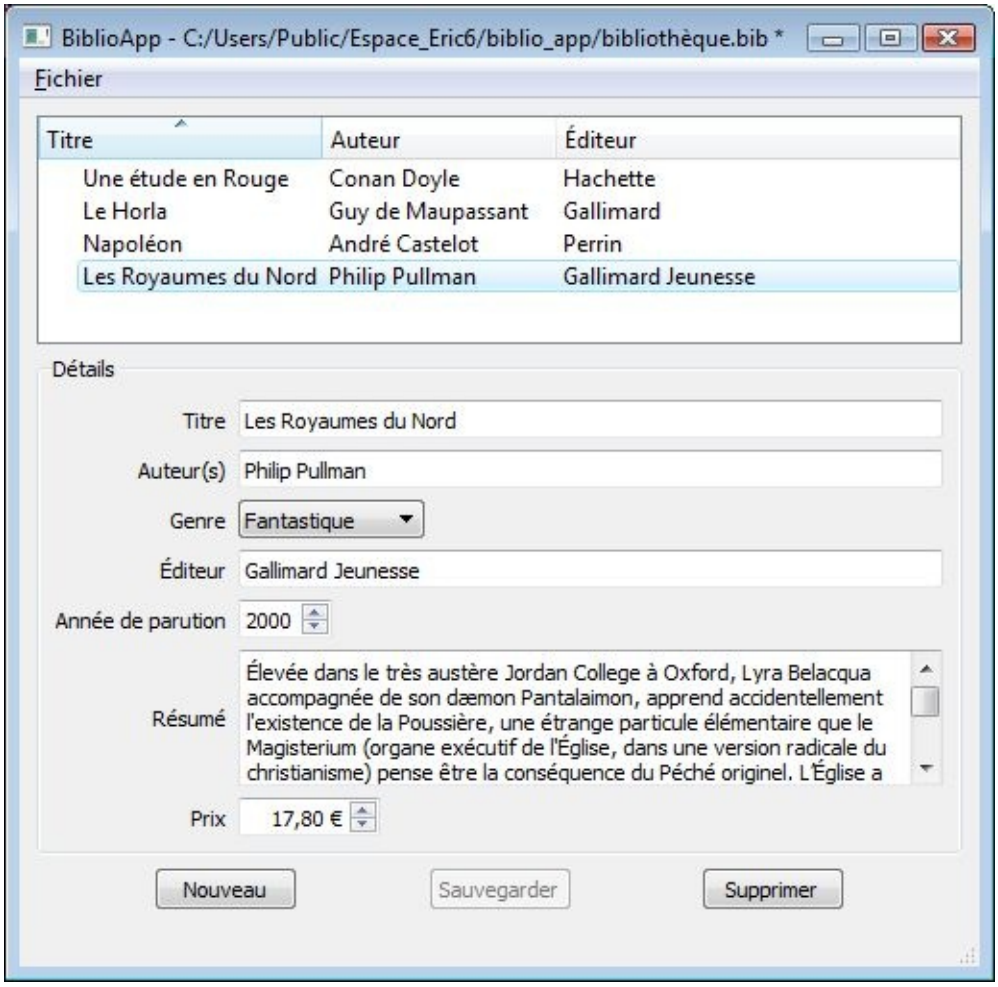
Notre environnement de développement mis en place, passons au développement proprement dit avec PyQt. Comme décrit à la Section 3, Choix d'une interface, il existe plusieurs manières de développer une interface graphique dans PyQt. Dans ce module, nous allons expliquer le développement avec Qt Widgets qui est la technique traditionnelle proposée par PyQt (Qt Quick n'est apparu qu'avec Qt 4.7, en 2013). Nous allons voir :

- les éléments de base d'une interface graphique basée sur Qt Widgets ;
- la gestion des événements avec les signaux et slots ;
- la conception des fenêtres à la souris (sans rien n'avoir à coder) avec Qt Designer ;
- l'internationalisation d'une application ;
- l'accès à une base de données.

Pour illustrer ces concepts, nous allons construire pas à pas une application de gestion de livres dans une bibliothèque. Nous appellerons cette application *BiblioApp*, à défaut d'un nom plus original !

L'interface utilisateur de BiblioApp se présentera sous la forme d'une fenêtre telle que présentée ci-après.

Figure .1 : *L'application BiblioApp, programmée avec Qt Widgets*



Cette application, simple et classique, ne devrait dérouter personne. On y trouve :

- une fenêtre principale, décorée des éléments standards (barre de titre, menu système, boutons de minimisation, maximisation et clôture) ; on verra qu'une application PyQt s'affiche d'une manière spécifique à l'OS sur laquelle elle s'exécute (Windows Vista, sur l'image) ; on parle dans ce cas d'*aspect et convivialité natifs* (en anglais, *native look & feel*) ;

- une barre de menus permettant, par exemple, d'ouvrir le menu Fichier et de sélectionner l'entrée Ouvrir... ;

- une liste, montrant les livres avec leurs attributs principaux ;

- des libellés et des champs de saisie pour saisir les données d'un nouveau livre ou éditer le livre sélectionné dans la liste ;

- des boutons pour créer, éditer ou supprimer un livre.

Cette application, quoique simple d'utilisation, permet d'illustrer les concepts les plus importants d'une application PyQt basée sur Qt Widgets tant au niveau de la conception que de la programmation. Notons que le développement d'une application similaire en Qt Quick est présentée dans Développement d'une application avec Qt Quick.

Au chapitre Créer une première application, nous initierons le projet et poserons les concepts de base de l'application ; au chapitre Développer avec Qt Designer, nous ferons le développement complet avec Qt Designer par glisser-déposer des composants et, au chapitre Programmer par modèle-vue, nous l'optimiserons à l'aide du patron de conception MVD ; enfin, nous présenterons des évolutions possibles de l'application aux deux derniers chapitres (Internationaliser son application et Accès à une base de données).

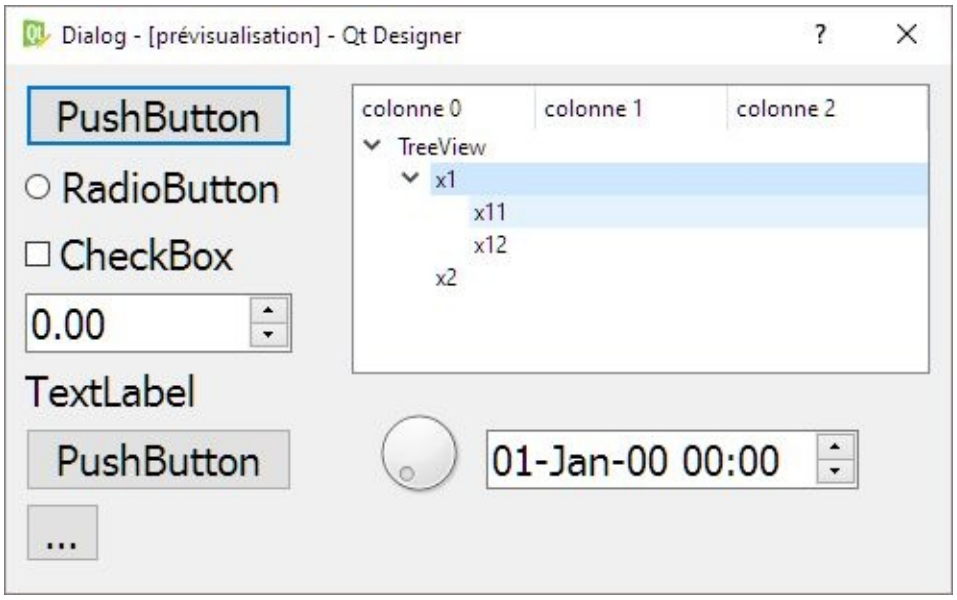
Anatomie d'une GUI

Avant de commencer le développement proprement dit, il est nécessaire d'introduire quelques concepts et un peu de vocabulaire. À l'exception de la Section 4, Signaux et slots de PyQt qui couvre des concepts spécifiques à Qt/PyQt, ce chapitre s'adresse principalement aux lecteurs peu familiers avec le développement de GUI (*Graphical User Interface*).

1. Widgets

Une application GUI est composée de plusieurs composants qui communiquent entre eux. Ils peuvent être graphiques, c'est-à-dire visibles à l'écran, ou non graphiques. Parmi les composants graphiques d'une application donnée, on aura en général une fenêtre principale et plusieurs fenêtres secondaires (typiquement, des fenêtres de dialogue) qui apparaîtront au gré des besoins. Le programmeur a pour rôle de créer ces fenêtres, selon un assemblage de divers composants graphiques appelés *widgets*[2]. Ces widgets sont fournis par la bibliothèque graphique et PyQt n'en est pas avare, avec une large palette, bien conçue et une riche documentation pour l'accompagner. D'ailleurs, si ces composants de base n'étaient pas suffisants, vous pourriez toujours définir les vôtres.

Figure 4.1 : *Quelques widgets PyQt...*



Certaines fenêtres standards sont prêtes à l'emploi : par exemple, une fenêtre de dialogue pour sélectionner un fichier ou pour inviter l'utilisateur à répondre à une question Oui/Non. Les fenêtres plus spécifiques devront quant à elles être construites par le programmeur, en positionnant les éléments un à un. Comme déjà signalé, Qt met à la disposition des développeurs une application puissante pour dessiner les fenêtres sans avoir à saisir une ligne de code : Qt Designer.

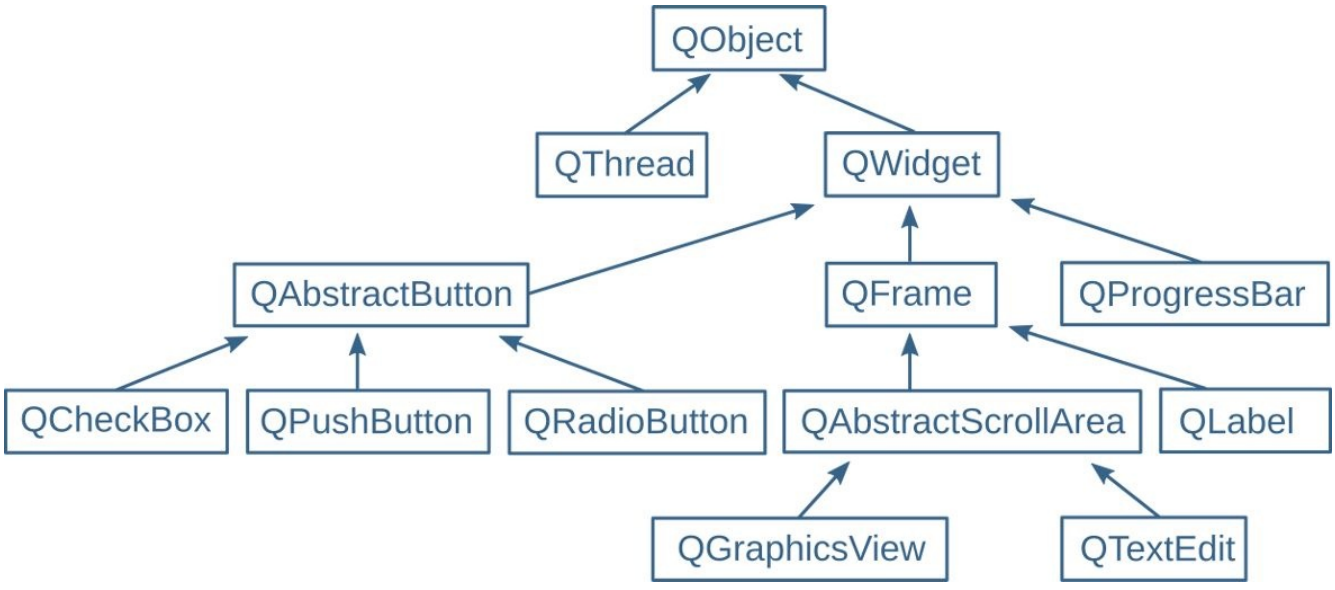
[2] Le mot widget est souvent présenté comme la contraction de *windows gadgets*. Toutefois on le trouve déjà dans la pièce *Beggar on Horseback* de George S. Kaufman… en 1924 !

2. Orientation objet

Comment concrètement cet assemblage de composants doit-il se programmer ? La programmation d'une interface graphique se base sur le paradigme de l'*orientation objet* (OO). C'est un trait commun à la grande majorité des bibliothèques graphiques, en incluant Qt et PyQt : chaque élément de l'interface utilisateur, du plus simple au plus compliqué, est un objet : il a ses attributs et ses méthodes, définis par la classe dont il est l'instance. Un prérequis pour comprendre les chapitres qui suivent est donc de maîtriser les concepts de base de l'OO et de connaître comment ils sont incarnés en Python (rappelons que Python est multiparadigme : on peut donc programmer en Python sans faire de l'OO).

PyQt fournit les composants sous forme d'un ensemble de classes ; ces classes devront être instanciées par le programmeur en fournissant les arguments requis par les constructeurs. Vu la complexité interne des composants, PyQt utilise la technique de l'héritage de façon poussée : les classes à utiliser par le programmeur d'application font partie d'une hiérarchie de classes parfois imposante ; aussi, les méthodes accessibles dans une classe donnée peuvent être documentées dans une classe parente ou ancêtre.

Figure 4.2 : *Hiérarchie de classes Qt (extrait)*



Par exemple, la plupart des éléments affichés (champs de texte, boutons, etc.) héritent d'une classe abstraite appelée QWidget qui intègre notamment toute la stratégie de

positionnement et de dimensionnement d'un élément. La classe QWidget sait juste qu'elle doit dessiner *quelque chose* dans une zone rectangulaire sur l'écran ; ce qu'il faut dessiner exactement est défini dans les classes enfants de QWidget.

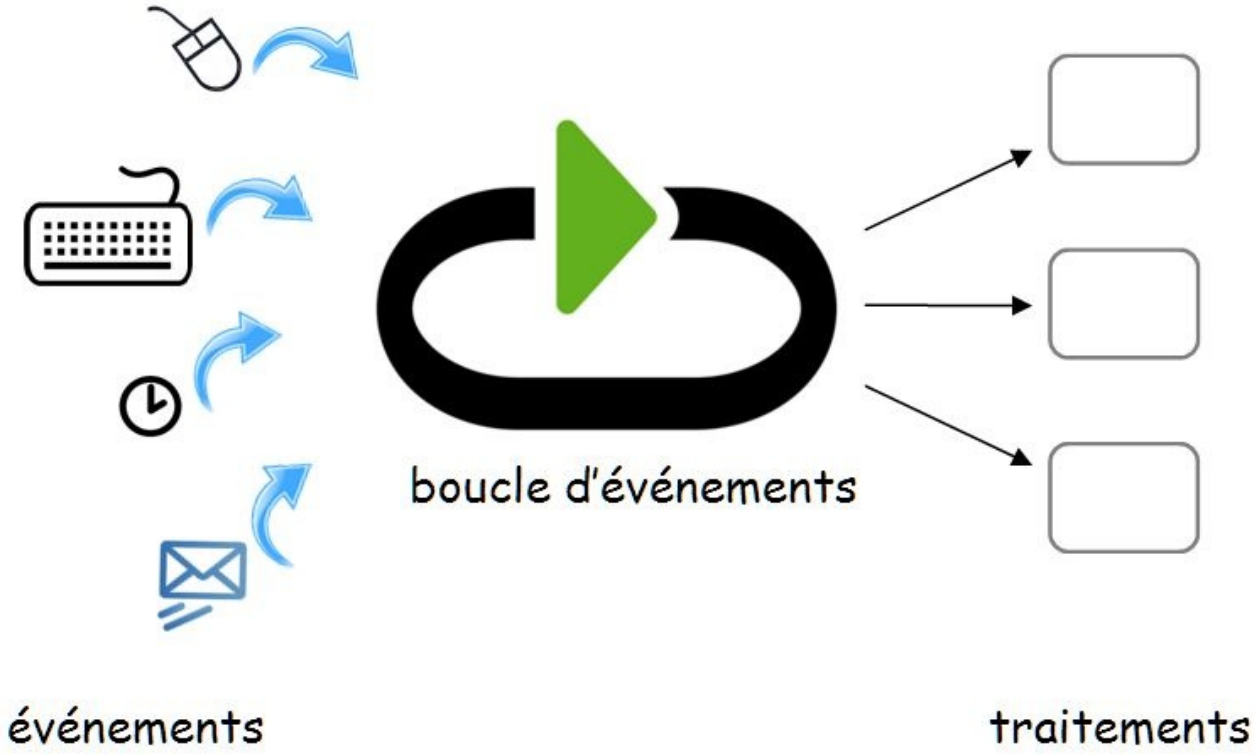
3. Programmation événementielle

Bien entendu, une application est beaucoup plus qu'un assemblage de composants graphiques sans interactivité, sinon ce ne serait qu'une "nature morte" ! Selon les actions effectuées par l'utilisateur, il faudra exécuter certains traitements qui peuvent influer sur les composants affichés. On parle de *programmation événementielle* pour ce type d'application (*event-driven programming*). Typiquement, un programme classique de type séquentiel lit des données en entrée, exécute un traitement, produit des données résultats et s'arrête. À l'inverse, une application de type interface utilisateur, après initialisation, se met en attente et réagit aux événements qui se présentent. On parle de *boucle d'événements* pour désigner la boucle principale qui réceptionne les événements et les dirige vers les fonctions de traitement adéquates.

Quels sont ces événements ? En fait, il y en a beaucoup : clics et déplacements de la souris, enfoncement/relâchement d'une touche au clavier, etc. Outre ces événements liés aux actions de l'utilisateur, d'autres sont d'ordre programmatique : fin d'une tâche, réception d'un message, échéance d'un certain intervalle de temps, etc. C'est par leur biais que l'affichage peut se mettre à jour, même lorsque l'utilisateur ne fait rien : par exemple, une barre d'avancement qui évolue, une petite enveloppe qui apparaît lorsqu'un nouvel e-mail est arrivé ou même une balle qui rebondit dans un jeu.

La Figure 4.3 illustre le principe de la programmation événementielle : la boucle d'événements vérifie si de nouveaux événements sont apparus ; le cas échéant, elle appelle la fonction de traitement correspondante.

Figure 4.3 : *Programmation événementielle*

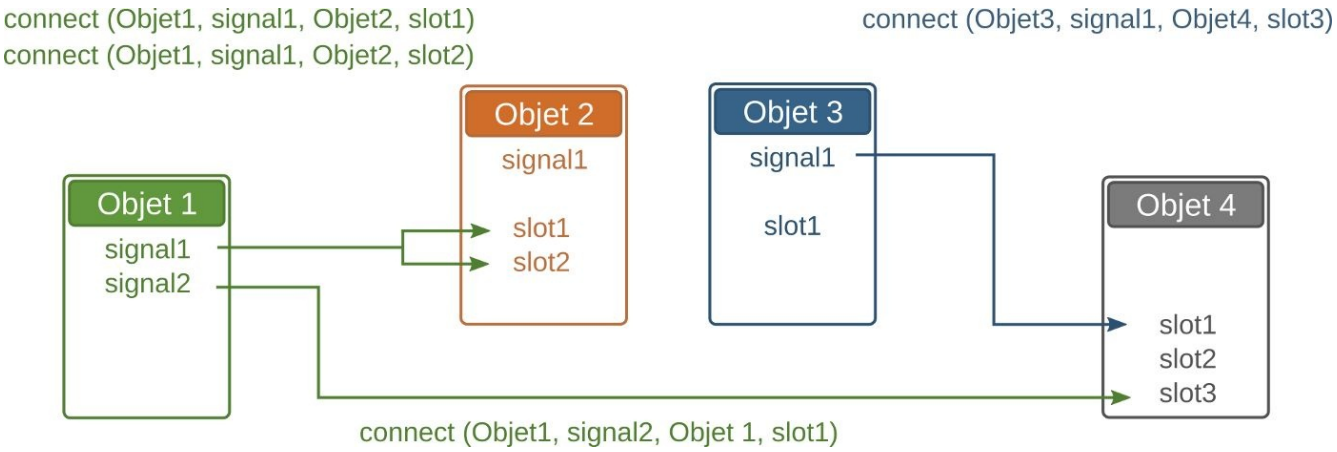


Fort heureusement, les composants de PyQt gèrent par eux-mêmes la majorité des événements et réagissent de manière adaptée, soulageant le programmeur d'une multitude d'activités de bas niveau : redimensionnement d'une fenêtre, affichage et parcours de menus, changement de couleur quand le curseur passe au-dessus d'un élément, infobulles, etc. Le programmeur peut dès lors se concentrer sur les événement spécifiques à l'application et les traitements qui leur sont associés.

4. Signaux et slots de PyQt

Comment concrètement faut-il s'y prendre pour réagir aux événements ? PyQt définit un mécanisme original et puissant : les *signaux* et les *slots*. Lorsqu'un événement se produit un signal est émis, identifiant l'événement, le composant où il se produit et d'éventuels paramètres associés. Le programmeur peut *connecter* ce signal à une méthode spéciale appelée slot. Cette méthode s'exécutera alors à chaque fois que le signal est émis. Un signal ne doit pas forcément être connecté à un slot (typiquement, si le traitement de PyQt est suffisant). Si nécessaire, le même signal peut être connecté à plusieurs slots, qui s'exécuteront tous à l'émission du signal.

Figure 4.4 : *Signaux et slots en PyQt*



Le grand avantage du mécanisme signaux/slots est qu'il permet une indépendance entre les composants. Le composant qui émet le signal n'a pas besoin de connaître le ou les composant(s) qui lui sont connectés : il émet son signal et ne se soucie pas de ce qu'il en advient.

Tout ce qui vient d'être présenté peut paraître un peu abstrait, mais ces concepts vont bien sûr être détaillés pas à pas dans les chapitres qui suivent et… concrétisés dans du code Python !

Créer une première application

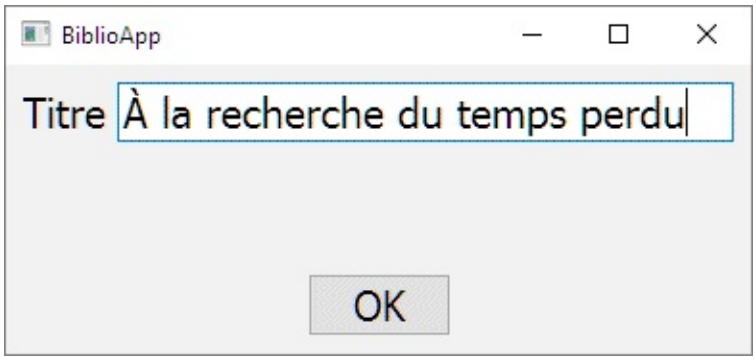
Niveau : débutant

Objectifs : comprendre les concepts généraux de Qt Widgets : fenêtres, widgets, signaux/slots et layout

Prérequis : Anatomie d'une GUI

Pour introduire les concepts de base de PyQt, nous allons commencer par construire une application minimale : une fenêtre montrant un libellé fixe, un champ de saisie textuel et un bouton.

Figure 5.1 : *BiblioApp version 0*

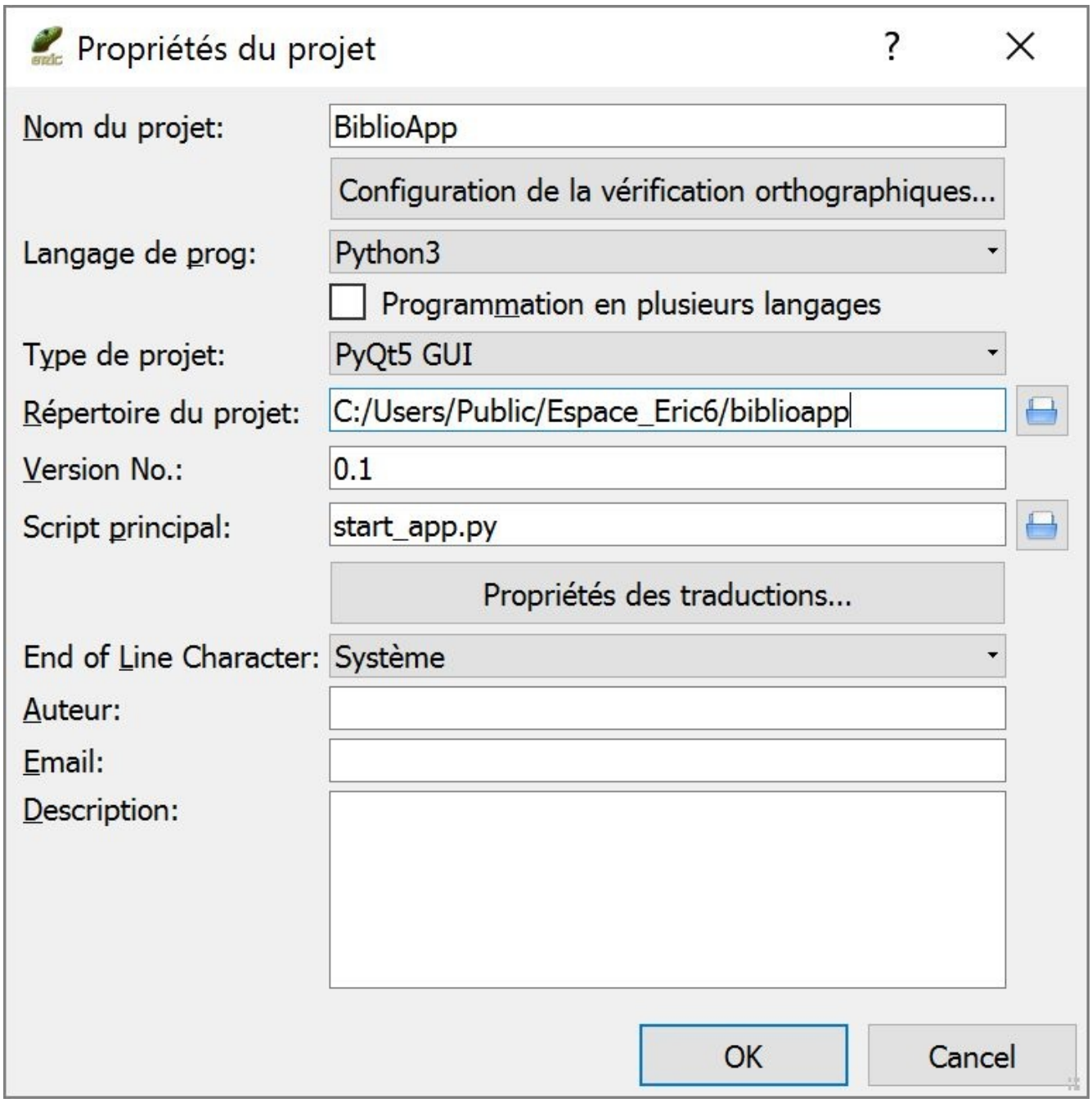


Nous réaliserons cette application en nous passant, dans un premier temps, de l'outil Qt Designer. Cet outil sera expliqué au chapitre suivant, où nous développerons l'application BiblioApp complète. Le but recherché ici est de bien comprendre le fonctionnement d'une application PyQt, avant de s'armer d'outils plus sophistiqués.

1. Création du projet

Pour créer le projet, on procède comme expliqué au chapitre d'introduction à eric6. Dans le présent chapitre, on utilisera comme nom du projet *BiblioApp*, comme répertoire `biblioapp` et comme script principal `start_app.py` (voir Figure 5.2). Vous pouvez bien sûr changer ces données comme bon vous semble, en adaptant les instructions quand nécessaire.

Figure 5.2 : *Création du projet BiblioApp*


Propriétés du projet
Nom du projet:
BiblioApp
Configuration de la vérification orthographiques...
Langage de prog:
Python3
Programmation en plusieurs langages
Type de projet:
PyQt5 GUI
Répertoire du projet:
C:/Users/Public/Espace_Eric6/biblioapp
Version No.:
0.1
Script principal:
start_app.py
Propriétés des traductions...
End of Line Character:
Système
Auteur:
Email:
Description:
OK
Cancel

2. Fenêtre principale : la classe QMainWindow

La majorité des applications PyQt contiennent au minimum une fenêtre, appelée *fenêtre principale*. Elle apparaît au démarrage et constitue le point d'entrée qui donne accès à toutes les fonctions de l'application. Dans le monde PyQt, le contenu de cette fenêtre et son comportement seront définis par le programmeur dans une classe Python qui hérite de la classe QMainWindow.

***Note** > Pour la documentation des classes (telle que QMainWindow), le site de PyQt renvoie au site de Qt, donc aux classes C++. Pour éviter l'indirection, les liens que nous fournissons pour les classes réfèrent directement à celles de Qt. A priori, ceci peut paraître un inconvénient pour le programmeur Python mais, en fait, les signatures de méthodes C++ apportent des informations très intéressantes sur les types attendus pour les arguments et le type de l'objet renvoyé. Si vous n'avez aucune connaissance du C++, l'encadré suivant est fait pour vous !*

Comment lire la documentation Qt C + + ?

Quand on programme en PyQt, il n'est pas nécessaire de bien connaître le C++ (un langage bien plus compliqué que Python) pour comprendre la documentation de Qt. Voici un petit guide pour vous aider à exploiter les informations données par les signatures de méthodes dans cette dernière.

- À l'inverse de Python, C++ est un langage typé statiquement : chaque argument est précédé du type attendu pour cet argument et chaque nom de méthode est précédé du type qu'elle renvoie. Quand le type de retour est `void`, cela indique que la fonction ne renvoie rien.

- Vous pouvez allégrement ignorer le mot clé `const`, les symboles `*` (pointeur) et `&` (référence) : il n'y a pas ces raffinements en Python (toute variable/argument est implicitement une référence).

- Le mot clé `static` indique une méthode qui ne s'applique pas à une instance particulière de la classe. C'est exactement comme une méthode décorée par `@staticmethod` en Python (pas d'argument `self`).

- Le mot clé `protected` associé à une méthode *m* d'une classe *QCcc* indique que *m* ne peut être appelée que par une méthode de *QCcc* ou d'une de ses classes filles.

- Le mot clé `virtual` associé à une méthode *m* d'une classe *QCcc* indique que vous pouvez surcharger la méthode *m* dans une classe fille de *QCcc*. En Python, qui ne s'embarrasse pas de telles formalités, *toutes* les méthodes peuvent être surchargées. Le mot clé `virtual` vous signale donc quelles méthodes sont destinées à être (éventuellement) réimplémentées dans vos classes qui héritent de *QCcc*. Ces méthodes sont appelées quand il le faut par la classe *QCcc* elle-même : c'est ce qu'on appelle le *polymorphisme*, dans le monde orienté objet. À noter : le mot clé `virtual` est souvent accompagné du mot clé `protected`.

Créer une fenêtre principale en PyQt est une tâche relativement systématique. Pour une application simple, telle que BiblioApp, les grandes étapes sont, dans l'ordre :

1. créer une classe qui hérite de QMainWindow : nous l'appellerons MainWindowBiblio ;

2. créer dans le constructeur de MainWindowBiblio les divers éléments graphiques constitutifs de la fenêtre (libellés, champs de texte et boutons) et les stocker en tant qu'attributs ;

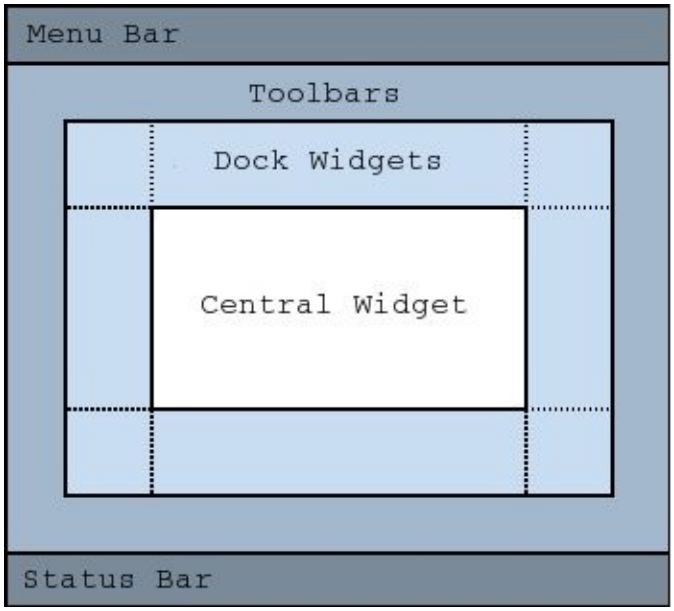
3. définir dans MainWindowBiblio les méthodes de réaction aux événements et les associer à ces événements (par exemple, clic sur bouton) ;

4. créer un programme principal qui instancie MainWindowBiblio, l'affiche et démarre l'application.

Nous allons tout d'abord nous attacher aux points 1, 2 et 4 cités et laisser la gestion du clic sur bouton pour plus tard.

Avant de commencer à programmer, il est utile de comprendre la structure d'une fenêtre principale telle que définie par PyQt. QMainWindow définit un agencement standard pouvant accueillir une barre de menus, des barres d'outils, des *dock widgets* et une zone centrale avec les éléments spécifiques à l'application. Tous ces éléments périphériques sont optionnels, mais, s'ils sont définis, ils seront placés par PyQt selon le format standard qui suit :

Figure 5.3 : *Structure d'une QMainWindow*



Pour notre application, la première chose à faire sera de créer un *widget central*, qui est en général un objet conteneur passif destiné à accueillir les éléments spécifiques à l'application (libellés, champs de saisie texte, listes, boutons, etc.). Ce widget central est en quelque sorte la grande plaque Lego rectangulaire sur laquelle nous allons placer nos briques. En termes PyQt, ce widget sera le parent de tous les éléments visibles de la fenêtre, hors éléments standards mentionnés (barre de menus et consorts).

Après ce préambule, nous pouvons définir la classe MainWindowBiblio, que nous allons écrire dans le fichier `main_window_biblio.py`. Pour créer ce fichier dans le projet BiblioApp, en supposant que ce dernier est ouvert, on active le menu Fichier > Nouveau et on le sauvegarde avec Fichier > Enregistrer sous....

```
# main_window_biblio.py

from PyQt5.QtWidgets import QMainWindow, QWidget, QLabel, QLineEdit,
\
```

❶ QPushButton class MainWindowBiblio(QMainWindow): ❷ def __init__(self): super(MainWindowBiblio,self).__init__() ❸ self.resize(300,150) ❹ self.setWindowTitle("BiblioApp") ❺ self.centralWidget = QWidget(self) ❻ self.setCentralWidget(self.centralWidget) self.label = QLabel("Titre",self.centralWidget) ❼ self.lineEditTitre = QLineEdit(self.centralWidget) self.lineEditTitre.move(80,0) self.pushButtonOK = QPushButton("OK",self.centralWidget) self.pushButtonOK.move(20,40)

On voit dans ce fichier :

❶ l'import des classes PyQt qu'on utilise, à partir du module `PyQt5.QtWidgets`. On les a nommées ici pour vous indiquer dans quel module PyQt ces classes sont définies ; on aurait tout aussi bien pu faire `from PyQt5.QtWidgets import *` ;

❷ la définition de la classe QMainWindow comme classe fille de QMainWindow ;

❸ l'appel au constructeur parent, via la fonction `super` (chose habituelle – et essentielle ! – en OO) ;

❹ le dimensionnement de la fenêtre : 300 pixels (horizontalement) × 150 pixels (verticalement) ;

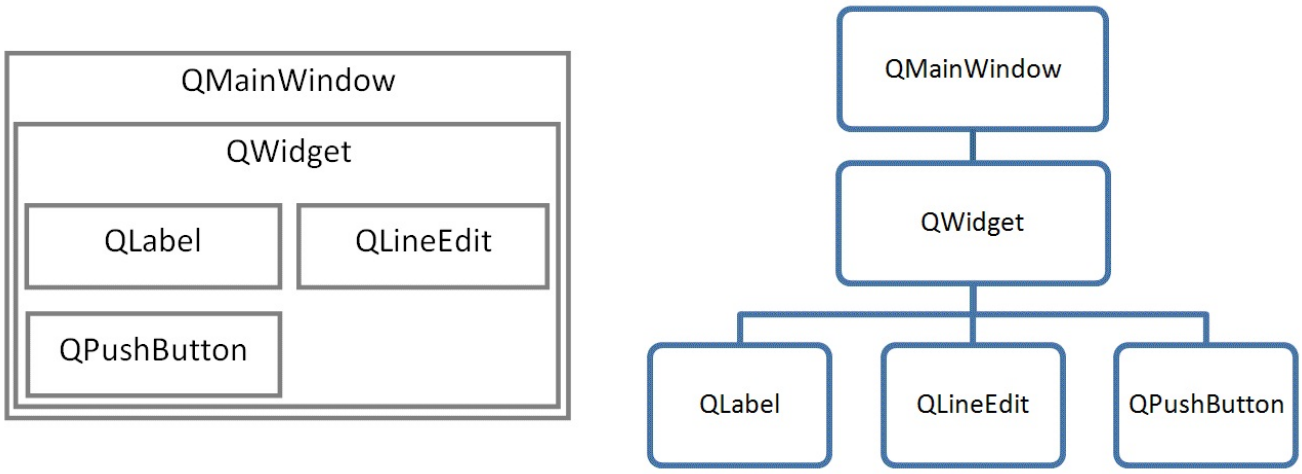
❺ la définition du titre à afficher pour la fenêtre, via `setWindowTitle` ;

❻ la création d'un widget central, instance de QWidget ;

❼ la création d'un libellé statique affichant `"Titre"` (instance de QLabel), d'un champ de saisie (instance de QLineEdit) et d'un bouton OK (instance de QPushButton).

La création des divers éléments mérite quelques précisions. Chaque constructeur attend un argument indiquant un objet parent. Ceci permet à PyQt d'enregistrer les relations de parenté entre composants et définir ainsi une arborescence. Nous voyons ainsi que, par construction, le widget central (`self.centralwidget`) est enfant de la fenêtre principale (`self`) et les trois autres éléments sont enfants de ce widget central. La hiérarchie des composants de notre fenêtre MainWindowBiblio est illustrée sur la Figure 5.4 (à gauche, la version visuelle et, à droite, la version arborescente, telle que stockée en mémoire par PyQt).

Figure 5.4 : *Hiérarchie des composants dans MainWindowBiblio*



Ces relations sont importantes pour plusieurs raisons, notamment pour le positionnement des composants. Ainsi, les coordonnées de n'importe quel composant sont relatives au parent de ce composant, avec (0,0) correspondant au coin supérieur gauche du parent. Par ailleurs, chaque instance a été assignée à un attribut de la fenêtre principale, de sorte que toute méthode puisse y accéder, en écrivant par exemple `self.lineEditTitre` pour l'élément ligne de saisie (`QLineEdit`). Il est important de noter toutefois que l'assignation à des attributs de MainWindowBiblio (comme `self.lineEditTitre = ...`) ne remet pas en cause la relation de parenté évoquée plus haut : le widget parent de l'objet QLineEdit est et reste `self.centralWidget` (instance de QWidget) et *pas* `self` (instance de MainWindowBiblio) ; les attributs tels que `self.lineEditTitre` servent essentiellement à pouvoir référencer simplement chaque composant, quel que soit le niveau qu'il occupe dans la hiérarchie.

Notons au passage un aspect qu'on va se dépêcher d'oublier dans les sections qui suivent, car laborieux et archaïque : les instructions `move(x,y)`, qui sont destinées à positionner les divers composants. Sauf avis contraire, PyQt place tous les éléments créés aux coordonnées relatives (0,0), ce qui correspond dans notre cas au coin supérieur gauche du widget central : les éléments s'empilent les uns sur les autres de sorte qu'on ne voit finalement que le dernier créé. Le positionnement via `move(x,y)` est une solution rapide pour se tirer d'affaire. Cette solution n'est toutefois pas recommandée ; nous verrons bientôt une technique beaucoup plus puissante.

3. Programme principal

Il reste à écrire le programme principal et à le placer dans `start_app.py` (le script principal défini pour notre projet eric6).

Outre la fenêtre principale définie plus haut, ce programme utilise une instance de QApplication, un objet discret mais essentiel de PyQt : il gère notamment l'initialisation du moteur graphique et la gestion des événements. Le programmeur n'a en général pas à se soucier de cet objet si ce n'est l'instancier et l'exécuter. Voyons concrètement comment on fait cela dans le listing ci-dessous.

```
# start_app.py

import sys
from PyQt5.QtWidgets import QApplication
from main_window_biblio import MainWindowBiblio

❶ app = QApplication(sys.argv) ❷ mainWindowBiblio = MainWindowBiblio() ❸ mainWindowBiblio.show() ❹ rc = app.exec_() ❺ sys.exit(rc)
```

Comme on peut s'y attendre, on commence par importer la classe MainWindowBiblio ❶ et l'instancier ❸. Notons que la création d'une fenêtre n'implique pas son affichage : elle est simplement chargée en mémoire et initialisée ; pour la faire apparaître à l'écran, il faut invoquer la méthode `show()` ❹. On crée une instance de QApplication ❷ : on passe au constructeur QApplication les arguments éventuels fournis au démarrage (`sys.argv`). Ensuite, on démarre l'application en invoquant la méthode `exec_()` ❺ ; l'appel à cette méthode est essentiel : il donne vie à l'application en lançant la boucle d'événements qui va réagir à toutes les actions de l'utilisateur. Cette méthode ne rend pas directement la main à l'appelant : elle n'est interrompue qu'à la clôture de l'application, quand l'utilisateur ferme la fenêtre principale.

***Note** > À la fermeture de la fenêtre, la méthode `exec_` renvoie un nombre appelé code de retour (variable `rc` dans le listing) : il vaut zéro si l'application s'est terminée de façon normale ou un nombre négatif identifiant le type d'erreur dans le cas contraire. Le programme principal peut alors se terminer ; la bonne pratique est de transmettre ce code à l'environnement d'exécution via `sys.exit(rc)`.*

Dans une application PyQt bien conçue, le programme principal est très court et semblable à celui que nous venons d'écrire. Il est recommandé de placer la logique de l'application dans un ou plusieurs modules, tels que `main_window_biblio.py`, qu'on

importera dans le programme principal.

Pour lancer l'application la première fois, on active le menu Lancer > Lancer projet... (raccourci Maj+F2) et on clique directement sur OK dans la boîte de dialogue. Ceci va exécuter le programme principal `start_app_py`. Pour les exécutions ultérieures, il suffit d'activer le menu Lancer > Restart (raccourci F4).

Figure 5.5 : *BiblioApp v0.01*

Note *> Si vous préférez travailler sans EDI, il suffit de lancer la commande* `python start_app.py` *sur votre console.*

4. Gestion des événements

Voyons maintenant la dernière étape que nous avions jusqu'ici laissée de côté : la gestion des événements. Dans notre application, imaginons – pour l'exemple – que nous voulons réagir à l'événement "clic sur bouton OK" en affichant le texte saisi dans une boîte de dialogue (voir Figure 5.6). Pour définir ce comportement dans notre application PyQt, il faut :

1. définir un *slot*, c'est-à-dire une méthode susceptible de réagir aux événements ;
2. faire une *connexion* du signal correspondant à l'événement vers ce slot.

PyQt propose plusieurs approches pour faire cela. Nous allons voir ici les deux plus simples et les plus pratiquées dans les applications de type Qt Widgets.

4.1. Approche 1 : connexion explicite

Commençons par l'approche la plus traditionnelle, qui consiste à définir explicitement les connexions signaux-slots. On ajoute la méthode `onPushButtonOKClicked`, qui est la méthode "slot", et on ajoute une instruction de connexion signal-slot dans `MainWindowBiblio.__init__`, via la méthode `connect`. En clair, ceci veut dire : "*chaque fois qu'on clique sur le bouton OK, je veux exécuter* `onPushButtonOKClicked`".

***Note** > Le code de cet exemple est disponible dans le projet* `connexion-explicite`  .

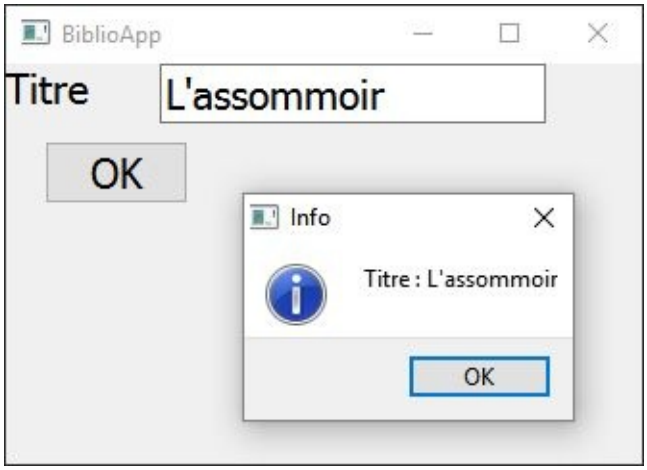
```
# main_window_biblio.py
# avec connexion signal-slot explicite

…
from PyQt5.QtWidgets import QMessageBox

class MainWindowBiblio(QMainWindow):

    def __init__(self):
        … # voir listing plus haut
        self.pushButtonOK.clicked.connect(self.onPushButtonOKClicked)

    def onPushButtonOKClicked(self):
```

```
        QMessageBox.information(self,"Info",
                                "Titre : "+self.lineEditTitre.text())
```

Que fait la méthode `onPushButtonOKClicked` ? Elle appelle la méthode statique `QMessageBox.information` de PyQt qui affiche une boîte de message simple avec un texte donné : ce texte, fourni en troisième argument, est la chaîne de caractères `"Titre: "` suivie de la chaîne saisie dans le QLineEdit, obtenue par l'expression `self.lineEditTitre.text()`.

On peut à présent tester l'application : une boîte de dialogue apparaît maintenant chaque fois qu'on clique sur OK. Cette boîte est dite *modale* : elle bloque toute autre activité sur la fenêtre principale tant qu'elle n'a pas été quittée.

Figure 5.6 : *BiblioApp v0.02*



4.2. Approche 2 : connexion par décorateur

Il existe une approche alternative qui semble, a priori, un peu plus lourde. Dans cette approche, la connexion ne se fait plus explicitement par appel à la méthode `connect` mais en décorant la méthode slot ciblée par `@pyqtSlot()` puis en renommant cette méthode selon une convention bien définie : elle aura un nom de la forme `on_`*`nomObjet_nomSignal`* ; dans le cas présent, ce sera donc `on_pushButtonOK_clicked`.

Pour des raisons techniques, ceci nécessite aussi quelques adaptations dans le constructeur de la fenêtre pour aider PyQt à établir les connexions : 1° en nommant les objets émetteurs du signaux via la méthode `setObjectName` et 2° en appelant la

méthode `QMetaObject.connectSlotsByName`.

Le listing suivant vous en donne un exemple.

***Note** > Le code est disponible dans le projet `connexion-par-decorateur` .*

```
# main_window_biblio.py
# avec connexion signal-slot par décorateur


…
from PyQt5.QtWidgets import QMessageBox
from PyQt5.QtCore import QMetaObject, pyqtSlot

class MainWindowBiblio(QMainWindow):

    def __init__(self):
        … # voir listing plus haut
        self.pushButtonOK.setObjectName("pushButtonOK")
        QMetaObject.connectSlotsByName(self)

    @pyqtSlot()
    def on_pushButtonOK_clicked(self):
        QMessageBox.information(self,"Info",
                                "Titre : "+self.lineEditTitre.text())
```

On peut vérifier à l'exécution qu'on obtient le même comportement que précédemment.

Quelques détails sur *pyqtSlot*

Le décorateur `pyqtSlot` est une fonction assez élaborée qui comporte plusieurs arguments optionnels. Ces arguments, omis dans l'exemple ci-dessus, permettent de contrôler finement les connexions signaux-slots. L'utilisation la plus importante de ces arguments consiste à spécifier les types des arguments du signal à connecter. Ceci est nécessaire notamment pour lever l'ambiguïté quand le même nom de signal a plusieurs signatures possibles, ce qui arrive parfois dans la bibliothèque C++ Qt. Par exemple, la classe QSpinBox propose deux signaux homonymes :

```
// Extrait de la classe QSpinBox de Qt ...
// Ceci est du C++ !
void valueChanged(int i);
void valueChanged(const QString &text);
```

Ceci signifie que *deux* signaux sont émis lorsque la valeur de la QSpinBox change :

l'un avec un nombre, l'autre avec une chaîne de caractères. Techniquement, on appelle cela la *surcharge des fonctions* ; elle est permise en C++ mais n'est pas possible en Python. Dès lors, pour connecter l'un des deux signaux de façon univoque, on indique la signature voulue comme argument de `pyqtSlot`, en utilisant les types Python. Par exemple, pour une QSpinBox nommée `spinBoxChoix`, on aura :

```
@pyqtSlot(int)
def on_spinBoxChoix_valueChanged(self,i):
    print ("nombre entier : %d" % i)
```

Dans cet exemple, `i` sera la valeur entière transportée par le premier signal. Si l'on veut *aussi* traiter le second signal, on ne peut plus utiliser la nomenclature habituelle, car Python écraserait silencieusement la première méthode par la seconde du même nom. Pour pallier ce problème, `pyqtSlot` propose l'argument `name` pour associer explicitement les noms de méthodes.

```
@pyqtSlot(int,name='on_spinBoxChoix_valueChanged')
def on_spinBoxChoix_valueChanged_int(self,i):
    print ("nombre entier : %d" % i)

@pyqtSlot(str,name='on_spinboxChoix_valueChanged')
def on_spinBoxChoix_valueChanged_str(self,msg):
    print ("chaîne de caractères : '%s'" % msg)
```

***Note** > Dans l'exemple qui précède, on a ajouté un suffixe aux noms de méthodes Python pour bien les distinguer. C'est juste un choix : rien n'impose de suivre une quelconque convention, car l'association est explicite par l'argument `name`.*

Pour plus de détails sur la fonction `pyqtSlot`, vous pouvez consulter la page de référence PyQt5.

4.3. Comparaison des deux approches de connexion

Quelle est l'utilité de cette seconde approche (décorateur `pyqtSlot`) par rapport à la première, qui paraît plus simple ? En fait, nous verrons bientôt qu'en utilisant l'outil Qt Designer, on pourra générer automatiquement les instructions que nous avons écrites dans `MainWindowBiblio.__init__`, y compris les formalités administratives comme les appels à `setObjectName`. Dans ce cas, le programmeur n'aura donc plus qu'à

décorer les méthodes slots et suivre une convention de nom pour établir les connexions. C'est facile, clair et de bon ton : le nom d'une méthode indique ce qui la déclenche. C'est très utile pour s'y retrouver quand on a plusieurs dizaines de méthodes slots.

Ceci dit, indépendamment de l'utilisation ou non de l'outil Qt Designer, la première approche (appel à `connect`) garde cependant son utilité dans certains cas :

- si on veut connecter plusieurs signaux au même slot ;
- si on veut établir ou rompre les connexions dynamiquement, selon certaines conditions.

La bonne nouvelle, c'est qu'on peut écrire une application PyQt qui utilise simultanément les deux approches, en tirant parti au mieux des avantages de chacune !

***Note** > Comme indiqué dans l'introduction de la présente section, les deux approches présentées ne sont pas les seules possibles ; elles sont simplement les plus courantes dans les applications Qt Widgets.*

Signalons par ailleurs, que le décorateur `pyqtSlot` a pour vocation première d'enregistrer explicitement comme slot la méthode Python décorée. Outre la possibilité de faire des connexions implicites (comme expliqué plus haut), ceci est très utile dans deux cas :

- *pour développer des extensions (ou plugins) à intégrer dans Qt Designer ;*
- *pour rendre une méthode Python accessible en Qt Quick (voir chapitre Communiquer avec Python).*

5. Mise en page intelligente avec QLayout

Notre première application a un gros défaut : nous avons positionné ses éléments en leur attribuant des coordonnées à la main via des instructions `move(x,y)`. C'est déconseillé pour plusieurs raisons :

- cela oblige à faire des calculs de coordonnées assez laborieux ; les modifications (ajout/suppression/déplacement de widgets) sont malaisées car nécessitent chaque fois de nouveaux calculs ;

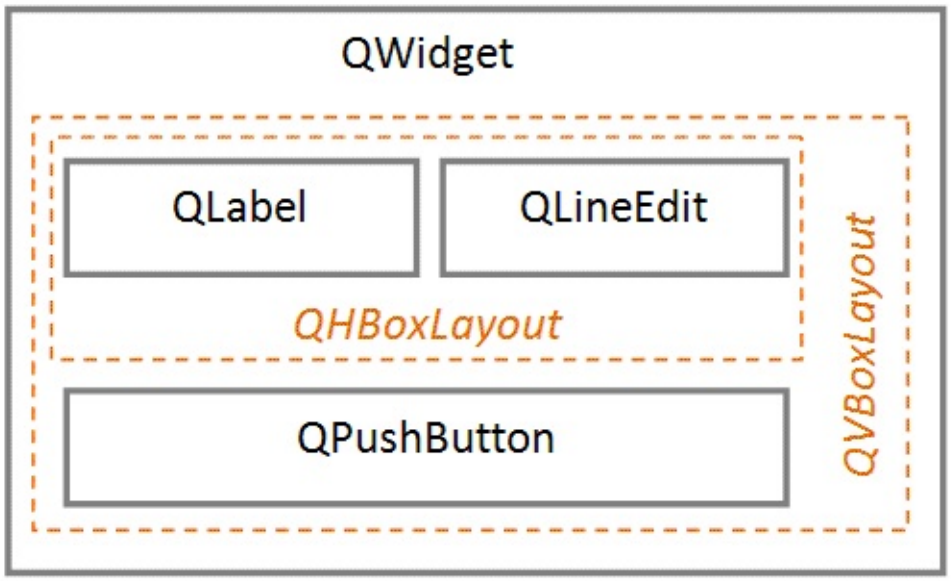
- l'espace disponible défini par la taille de la fenêtre principale n'est pas pleinement exploité ; c'est en général une donnée variable qui dépend des actions de l'utilisateur ; par exemple, il peut redimensionner la fenêtre.

PyQt propose un mécanisme puissant pour répondre aux besoins de mise en page. Ce mécanisme appelé *gestionnaire de géométrie* (*geometry manager* en anglais) a pour pilier principal la classe QLayout et ses classes filles. Le principe général consiste à regrouper les composants affichés dans des *positionneurs*, instances des classes filles de QLayout définissant chacune un certain type d'alignement (vertical, horizontal, grille, etc). Ces positionneurs, nous le verrons, peuvent être combinés hiérarchiquement pour faire des mises en page plus complexes. À l'exécution, ils sont utilisés par PyQt pour placer chaque composant selon la taille de leur fenêtre parente ; ces calculs sont faits lors de la création de la fenêtre, mais aussi chaque fois que celle-ci change de taille, typiquement suite à une action utilisateur. Grâce à QLayout, la mise en page de chaque fenêtre peut se faire dans un style déclaratif de haut niveau, sans devoir jamais se soucier de coordonnées ou des redimensionnements possibles.

Pour illustrer ces principes, nous allons modifier notre application en remplaçant les coordonnées en dur par une mise en page basée sur les positionneurs. Plus précisément, nous allons créer des alignements horizontaux et verticaux, respectivement via les classes QHBoxLayout et QVBoxLayout. Ces classes héritent de QBoxLayout, qui factorise les points communs aux deux types d'alignement et qui hérite elle-même de QLayout. La première étape consiste à déterminer la structure de mise en page souhaitée : nous décidons par exemple ici que le libellé (QLabel) doit être aligné horizontalement avec la zone texte (QLineEdit) et que le bouton (QPushButton) doit être aligné verticalement par rapport à ces deux éléments. Pour réaliser cela, on va placer le libellé et la zone texte dans un QHBoxLayout (alignement horizontal) qu'on place ensuite avec le bouton dans un QVBoxLayout (alignement vertical). Ce

QVBoxLayout est enfin associé au widget central.

Figure 5.7 : *Mise en page 1 avec QLayout*



Voici comment cela se traduit en PyQt :

```
# main_window_biblio.py
# avec mise en page par layout

…
from PyQt5.QtWidgets import QHBoxLayout, QVBoxLayout

class MainWindowBiblio(QMainWindow):

    def __init__(self):
        super(MainWindowBiblio,self).__init__()
        self.resize(300,150)
        self.setWindowTitle("BiblioApp")
        self.centralWidget = QWidget(self)
        self.setCentralWidget(self.centralWidget)
        self.label = QLabel("Titre",self.centralWidget)
        self.lineEditTitre = QLineEdit(self.centralWidget)
        self.pushButtonOK = QPushButton("OK",self.centralWidget)
        self.hBoxLayout = QHBoxLayout()
        self.hBoxLayout.addWidget(self.label)
        self.hBoxLayout.addWidget(self.lineEditTitre)
        self.vBoxLayout = QVBoxLayout(self.centralWidget)
        self.vBoxLayout.addLayout(self.hBoxLayout)
        self.vBoxLayout.addWidget(self.pushButtonOK)
        self.pushButtonOK.clicked.connect(self.onPushButtonOKClicked)
    …
```

Il y a plusieurs points à noter dans ce listing :

- visuellement, les éléments s'ajoutent de gauche à droite pour les QHBoxLayout et de bas en haut pour les QVBoxLayout ;
- il y a deux méthodes différentes selon le type d'objet qu'on veut ajouter (widget ou positionneur) : respectivement `addWidget` et `addLayout` ;
- le positionneur de plus haut niveau est créé en spécifiant en argument le widget qu'il gouverne : `QVBoxLayout(self.centralWidget)` dans l'exemple.

À l'exécution, on peut voir que les choses se présentent comme spécifiées. De plus, et c'est là un gain substantiel, quand on change la taille de la fenêtre principale, par exemple en faisant glisser ses coins, les éléments s'adaptent automatiquement.

Figure 5.8 : *BiblioApp v0.03*

Attention > *La hiérarchie de QLayout montrée ici ne doit pas être confondue avec la hiérarchie de QWidget, expliquée à la Section 2, Fenêtre principale : la classe QMainWindow. C'est une erreur fréquente chez les débutants en PyQt ! Soulignons, par exemple, que* `self.label` *(qu'on a placé dans un QHBoxLayout) n'a qu'un parent, qui est et reste* `self.centralWidget`*. Les deux hiérarchies coexistent sans interférer. Les QLayout ne sont pas des widgets, ils ne sont d'ailleurs pas visibles à l'exécution ; ils sont juste des directives invisibles pour placer les widgets enfants sur leur widget parent. C'est un peu déroutant au début mais, en définitive, c'est cohérent et on s'y fait vite !*

On voit l'intérêt de cette technique pour l'utilisateur : le champ textuel s'étend à mesure que la fenêtre est étirée horizontalement, ce qui est un gain en terme d'ergonomie.

Toutefois, les redimensionnements de la fenêtre donnent lieu aussi à quelques bizarreries : lors d'un étirement horizontal, le bouton OK s'élargit pour prendre toute la place disponible ; lors d'un étirement vertical, les éléments se placent au 1/3 et au 2/3 de la hauteur de la fenêtre. Toutes ces caractéristiques, qu'elles soient agréables ou non, ont en fait la même cause : sans autres indications, les éléments placés dans les positionneurs sont appelés à occuper au maximum l'espace disponible. Chaque QLayout définit les tailles et positions selon des règles définies, mais aussi selon des attributs que vous pouvez modifier. Tout ceci est puissant mais requiert une certaine expertise.

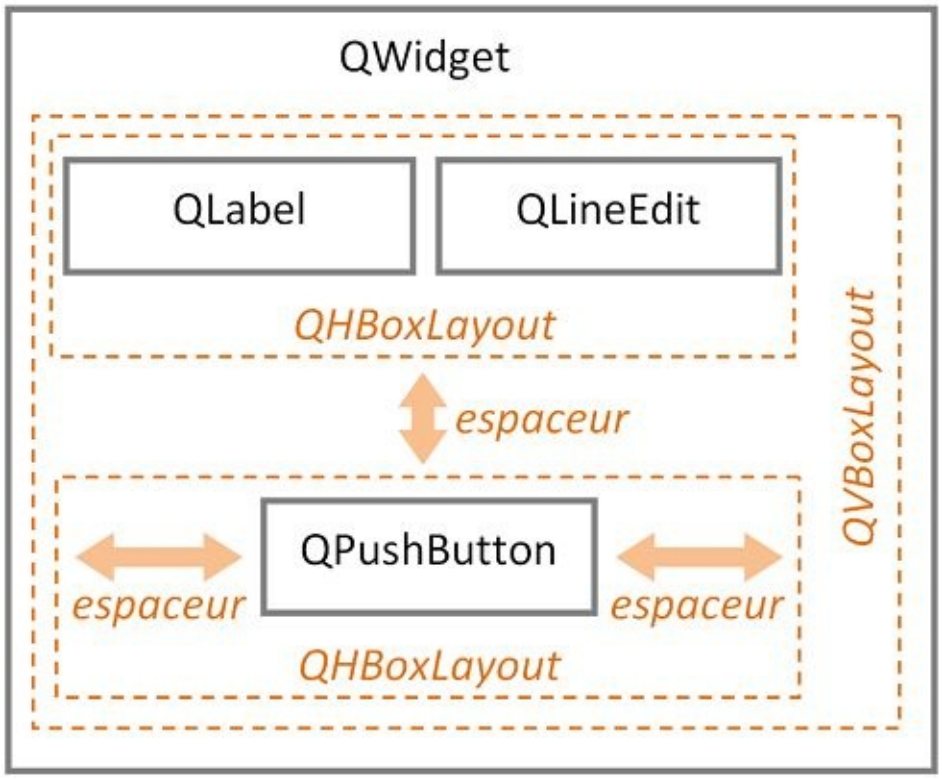
PyQt offre heureusement une technique beaucoup simple pour corriger ces petits défauts : les *espaceurs*. Un espaceur est une instance de la classe QSpacerItem ; c'est un objet rectangulaire qu'on peut placer dans un QLayout et qui a la caractéristique amusante… d'être invisible ! En fait, son seul but est de grossir, c'est-à-dire d'occuper l'espace qui pourrait être pris par les composants voisins. On peut aussi le voir comme un ressort qui va pousser les composants visuels pour qu'ils présentent au mieux, quelle que soit la dimension de la fenêtre.

Concrétisons tout cela sur notre exemple. On va :

- ajouter un espaceur au milieu du QVBoxLayout : ceci aura pour effet de pousser le bouton OK en bas de la fenêtre ;

- ajouter deux espaceurs de part et d'autre du bouton OK dans un nouveau QHBoxLayout : ceci aura pour effet de centrer le bouton et de l'empêcher de s'élargir.

Figure 5.9 : *Mise en page 2 avec QLayout et espaceurs*



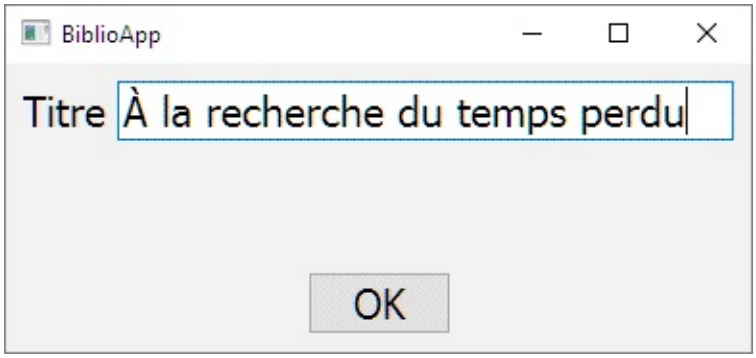
Les QSpacerItem peuvent être instanciés et ajoutés dans les QLayout par le programmeur ; toutefois, si on n'a pas besoin d'un contrôle fin, le QBoxLayout offre une méthode `addStretch()`, qui fait ce petit travail en une fois et est, donc, plus simple. Tout ceci est illustré dans le listing qui suit.

```
# main_window_biblio.py
# avec mise en page par layout

…

class MainWindowBiblio(QMainWindow):

    def __init__(self):
        super(MainWindowBiblio,self).__init__()
        self.setWindowTitle("BiblioApp")
        self.centralWidget = QWidget(self)
        self.setCentralWidget(self.centralWidget)
        self.label = QLabel("Titre",self.centralWidget)
        self.lineEditTitre = QLineEdit(self.centralWidget)
        self.pushButtonOK = QPushButton("OK",self.centralWidget)
        self.hBoxLayout = QHBoxLayout()
        self.hBoxLayout.addWidget(self.label)
        self.hBoxLayout.addWidget(self.lineEditTitre)
        self.vBoxLayout = QVBoxLayout(self.centralWidget)
        self.vBoxLayout.addLayout(self.hBoxLayout)
        self.vBoxLayout.addStretch()
```

```
        self.hBoxLayout2 = QHBoxLayout()
        self.hBoxLayout2.addStretch()
        self.hBoxLayout2.addWidget(self.pushButtonOK)
        self.hBoxLayout2.addStretch()
        self.vBoxLayout.addLayout(self.hBoxLayout2)
        self.pushButtonOK.clicked.connect(self.onPushButtonOKClicked)
    …
```

À l'exécution, on peut à présent apprécier le nouveau comportement quand on redimensionne la fenêtre.

Figure 5.10 : *BiblioApp v0.04*



Les positionneurs (classes QLayout) et les espaceurs (classe QSpacerItem) que nous venons de présenter permettent de faire à peu de frais des mises en page à la fois complexes et ergonomiques : on définit la disposition des widgets dans les grandes lignes et on laisse PyQt faire les calculs laborieux. Par ailleurs, cette technique de positionnement permet aussi d'adapter automatiquement les fenêtres selon la longueur des libellés, qui varie en fonction de la langue (voir fin du chapitre Internationaliser son application).

Nous allons voir dans la suite que tous ces concepts se retrouvent dans Qt Designer, avec la possibilité intéressante de tester directement le résultat.

Développer avec Qt Designer

Niveau : débutant

Objectifs : concevoir des interfaces avec Qt Designer et les convertir en code Python

Prérequis : Anatomie d'une GUI

Dans le chapitre précédent, nous avons vu comment créer une fenêtre principale, ajouter des widgets, gérer la mise en page et répondre aux événements. Avec de la patience et en lisant attentivement la documentation de Qt (excellente, par ailleurs !), on peut programmer à peu près tout ce qu'on veut. Cependant, pour juste concevoir la partie *écran* de l'application, on n'échappe pas à des cycles de programmation/exécution qui peuvent être laborieux.

Pour le développement d'interfaces utilisateur, il est souvent bénéfique d'impliquer au plus tôt les utilisateurs potentiels en leur soumettant une *maquette* de la future application : typiquement, il s'agira des fenêtres avec tous leurs éléments, mais sans aucune réaction aux actions de la part des utilisateurs. Cette maquette peut alors évoluer, de façon itérative, selon les remarques et suggestions obtenues. Quand la maquette est stabilisée et correspond aux attentes de chacun, le développement proprement dit peut alors commencer. Si on en a le temps et les moyens, on peut même envisager de confier l'élaboration de la maquette à un graphiste ou un concepteur de GUI qui peut réaliser ce travail sans avoir de compétences en programmation.

Pour répondre à ces besoins et aider la vie du programmeur, Qt propose un outil puissant et bien pensé : *Qt Designer*.

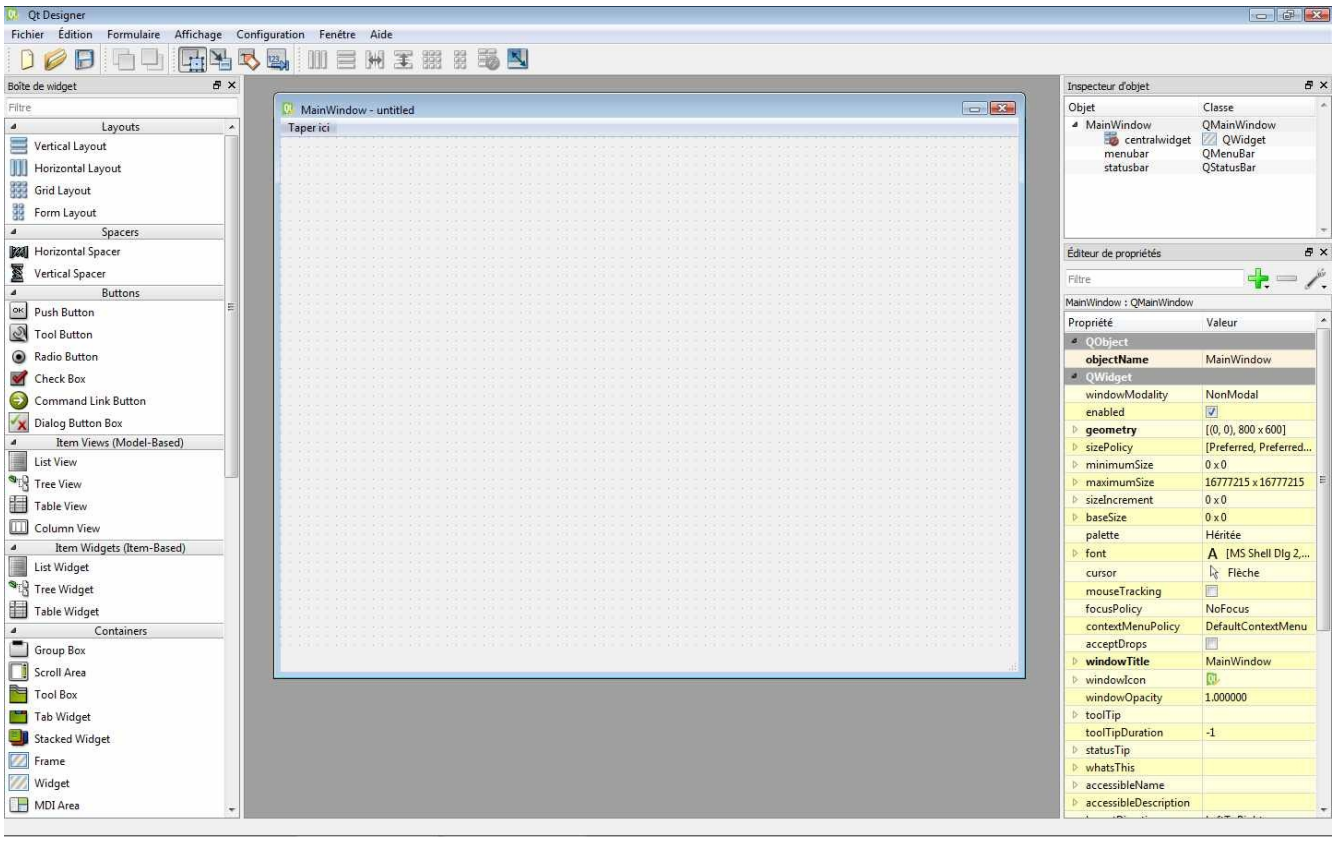
Qt Designer permet au concepteur de GUI de dessiner chaque fenêtre sans avoir à écrire une seul ligne de code : on choisit les widgets adéquats dans une grande boîte à outils et on les fait glisser sur la fenêtre à la position souhaitée. De même, on peut définir les stratégies de dispositions intelligentes et tester directement le résultat dans l'outil. Qt Designer est intuitif et facile d'utilisation ; pour la création de maquettes, il peut être laissé sans soucis à un non-programmeur. Une fois que la maquette est stable, le programmeur peut (re-)prendre la main et définir, toujours via Qt Designer, les noms qui serviront à identifier les composants, à générer les slots et leur connecter les signaux.

1. Présentation de Qt Designer

Il y a plusieurs manières de lancer Qt Designer :

- en activant l'icône Designer sur le bureau ou la barre de démarrage (dépend de l'OS) ;
- en double-cliquant sur un fichier ayant l'extension `.ui` ;
- en lançant simplement la commande `designer` sur une console ;
- en utilisant eric6 (voir plus bas).

Figure 6.1 : *Qt Designer*



Astuce *> Pour avoir la version française si elle n'a pas été sélectionnée à l'installation, il faut définir la variable d'environnement* LANG *avant de lancer l'application :*

`LANG=fr_FR`

La manière de définir cette variable d'environnement dépend de votre OS (renseignez-vous, au besoin !).

L'interface de Qt Designer est complètement configurable : les boîtes d'outils affichées peuvent être ajoutées ou enlevées via le menu Affichage ; les éléments latéraux (voir ci-après) peuvent aussi être déplacés d'un bord à l'autre en les faisant glisser.

Au démarrage, Qt Designer affiche par défaut les éléments les plus importants :

- la Boîte de widgets, qui est la grande boîte à outils de widgets fournis par Qt, qu'on va pouvoir placer sur notre fenêtre ;
- l'Inspecteur d'objet, qui répertorie tous les objets (widgets et layouts) placés sur la fenêtre en cours de conception, en détaillant leur structure arborescente. Guère utile au début, il peut rapidement devenir pratique pour sélectionner les objets difficilement atteignables sur la vue graphique ;
- l'Éditeur de propriétés, qui montre toutes les propriétés du widget sélectionné et permet de les éditer. La liste précise dépendra du widget sélectionné. Les propriétés sont rangées par classe héritée, en commençant par la classe racine QWidget.

***Note** > Par expérience, nous jugeons que la configuration d'affichage de Qt Designer par défaut est bien pensée pour démarrer. Il n'y a pas lieu dans un premier temps de la modifier.*

Que fait exactement Qt Designer ? La chose à comprendre avant tout est que Qt Designer est un éditeur de fenêtres Qt, *ni plus ni moins* ! On crée une fenêtre ou on ouvre une fenêtre existante, on y place/déplace des widgets, on définit leurs propriétés, etc. ; ensuite, la fenêtre peut être sauvegardée dans un fichier d'extension `.ui`, sous un nom choisi par l'utilisateur[3].

***Note** > Le terme fenêtre que nous utilisons ici doit être compris dans un sens très large : il peut s'agir d'une fenêtre principale, d'une boîte de dialogue, voire (plus rarement) d'un widget conteneur destiné à être placé à une autre fenêtre. Pour s'en convaincre, il suffit de regarder la boîte de dialogue Nouveau formulaire qui s'affiche au démarrage ou à l'activation du bouton Nouveau.*

Tous les éléments créés, leurs positions, leurs propriétés (si valeur différente de la valeur par défaut) sont sauvegardés dans ce fichier. Notons accessoirement que les

fichiers `.ui` suivent le format XML : on peut donc afficher leur contenu dans un éditeur texte ou XML, voire les modifier à la main (ce n'est toutefois pas la manière normale de procéder).

Un fichier `.ui` écrit par Qt Designer est donc juste une description statique et déclarative de tous les détails d'une fenêtre. L'isolation par rapport au langage de programmation cible est complète : on peut noter d'ailleurs que le même fichier `.ui` peut être utilisé aussi bien pour un développement PyQt que pour un développement natif Qt en C++.

Mais comment ces fichiers `.ui` vont-ils en définitive s'intégrer dans nos programmes Python ? Il existe deux techniques principales : la génération de code Python et le chargement dynamique de l'`.ui`. Ces techniques, qui comportent chacune des variantes, seront détaillées un peu plus loin, à la Section 3, Génération du code.

Signalons, enfin, que Qt Designer est ouvert aux *extensions* (ou *plugins*) : ce sont des widgets spécialisés qui peuvent être ajoutés à la Boîte de widgets de Qt Designer. Ils permettent notamment de factoriser dans des *super-widgets* des éléments d'interface récurrents, composés de plusieurs widgets en interaction[4]. Ces extensions peuvent être développées par tout un chacun… à la condition d'avoir une bonne expérience de Qt et du développement de composants. Ils sont classiquement programmés en C++, mais PyQt fournit tout ce qu'il faut pour le faire en Python. Nous n'aborderons toutefois pas ici ce sujet destiné à un usage avancé.

Tout ce qui précède est certes un peu abstrait mais, rassurez-vous, ces points vont être détaillés concrètement sur l'exemple de notre application BiblioApp. Plus précisément, dans les sections qui suivent, nous allons décrire les grandes étapes d'un développement en PyQt basé sur Qt Designer :

1. construire la fenêtre principale dans Qt Designer ;

2. générer le code Python correspondant ;

3. programmer l'application, en introduisant au passage les notions de *modèle* et *vue*.

***Note** > Dans ce qui suit, nous repartirons du projet eric6 BiblioApp créé au chapitre précédent ; le fichier `start_app.py` (programme principal) restera inchangé ; quant au fichier `main_window_biblio.py`, il va être complètement réécrit pour implémenter l'application conçue avec Qt Designer. Au besoin, vous pouvez bien entendu créer un nouveau projet eric6 et copier les quelques lignes de `start_app.py`.*

[3] Le choix de l'extension `.ui` par Qt vient bien sûr de l'expression *User Interface* !

[4] Expérience personnelle : j'ai ainsi développé, dans le cadre d'une application de gestion de satellites, un composant de saisie de temps ayant la particularité d'afficher à la fois le format habituel date/temps et le format "temps orbital" (numéro d'orbite + delta en secondes) ; ce composant groupe des widgets qui se synchronisent entre eux pour montrer toujours le même temps, quel que soit le mode de saisie. Ce composant a pu ainsi être intégré dans la boîte à outils Qt Designer et, dès lors, être simplement *déposé* sur les boîtes de dialogue qui le nécessitaient.

2. Construction de la fenêtre BiblioApp

Nous allons utiliser Qt Designer pour construire pas à pas la fenêtre principale de notre application BiblioApp. Nous sauvegarderons notre travail sous le nom `main_window_biblio.ui`.

Note *> Ce fichier et le code source présenté dans ce chapitre sont disponibles dans le dossier* `qt-designer`  *.*

2.1. Création du fichier .ui

Il y a différentes manières de procéder. Pour notre part, nous allons repartir du micro-projet BiblioApp créé au chapitre précédent. C'est juste une question de facilité, rien n'oblige à utiliser eric6 : un autre EDI peut aussi convenir, voire pas d'EDI du tout. L'avantage d'eric6 réside surtout dans la partie *génération de code*, qui est très bien intégrée.

La première chose à savoir, c'est qu'eric6 appelle *feuilles* les fichiers `.ui`. Dès lors, pour créer notre fichier une fois le projet BiblioApp ouvert dans eric6, on va logiquement sur le panneau Feuilles du projet, dans le Gestionnaire de projet. Ce panneau, initialement vide, affiche les fichiers `.ui` associés au projet (il pourra y en avoir plusieurs si l'application comporte plusieurs fenêtres, typiquement des fenêtres de dialogue). Sur ce panneau, activons Nouvelle feuille… dans le menu contextuel : une boîte de dialogue apparaît demandant de choisir le type de feuille ; nous choisissons Fenêtre principale. Enfin, nous saisissons `main_window_biblio.ui` comme nom de fichier. Après confirmation, le fichier est créé, il apparaît comme une nouvelle feuille du projet. En faisant un double-clic sur cet élément, Qt Designer se lance et ouvre le fichier `main_window_biblio.ui` : une fenêtre principale vide apparaît, prête à être décorée comme un sapin de Noël !

Note *> Si vous voulez travailler sans eric6, il suffit de lancer Qt Designer comme expliqué plus haut. La démarche est fort semblable : choisissez le type de fenêtre à créer : Main Window (non traduit en français !) et sauvegardez le fichier sous le nom* `main_window_biblio.ui`*. Au besoin, vous pourrez intégrer ce fichier plus tard à un projet eric6 via l'action Ajouter des feuilles… du menu contextuel.*

2.2. Placement des éléments principaux

Nous sommes prêts à placer les widgets de l'application. Avant de foncer tête baissée, il est utile de réfléchir à la structure générale de la fenêtre. Outre la barre de menu, il y a trois zones principales :

- une zone *Liste*, affichant les livres enregistrés ;
- une zone *Détails*, affichant tous les détails d'un livre, qu'il soit en cours de saisie ou d'édition ;
- une zone *Action*, avec les différents boutons permettant de créer, sauvegarder ou effacer un livre.

Sachant que la zone Détails contient des widgets logiquement liés, il est intéressant de les regrouper dans un widget conteneur. Ainsi, si nous voulons réorganiser plus tard l'interface utilisateur, nous pourrons facilement déplacer tous ces éléments en une fois en modifiant simplement le conteneur.

Pour débuter, nous allons suivre notre canevas composé de trois zones et placer les widgets principaux sur la fenêtre principale, en les glissant-déposant depuis la Boîte de widgets (voir la Figure 6.2) :

- un Tree View pour la liste de livres ;
- une Group Box pour les détails du livre ; après double-clic, on la renomme `Détails` ;
- trois Push Button ; après double-clic, on les renomme respectivement `Nouveau`, `Sauvegarder` et `Supprimer`.

Astuce *> Il n'est pas nécessaire de passer du temps à positionner ou aligner précisément les widgets. Nous allons très bientôt utiliser les techniques de disposition intelligente introduites au chapitre précédent ; un positionnement très sommaire suffit amplement à ce stade.*

Pour terminer cette première étape, nous allons créer un menu principal minimaliste avec un élément `Fichier`, lui-même contenant un sous-menu avec les éléments `Ouvrir…`, `Enregistrer` et `Quitter`.

Note *> Dans ce qui suit, les esperluettes ont pour effet, lors de l'exécution, de*

souligner les lettres qui les suivent et de créer les raccourcis correspondants sans devoir programmer. Dans l'exemple, on pourra invoquer la fonction `Ouvrir` en faisant Alt+F suivi de O. Nous verrons au point 5 comment ajouter, en sus, des raccourcis rapides (genre Ctrl+O) via la propriété shortcut.

1. Cliquez sur Taper ici dans le menu principal et saisissez `&Fichier`.

2. En cliquant sur le menu `Fichier` créé, le sous-menu correspondant s'ouvre : cliquez sur Taper ici et saisissez `&Ouvrir…` et `&Enregistrer`.

3. Cliquez sur Ajouter séparateur pour placer une ligne horizontale dans le menu (dans les grands menus, cela aide l'utilisateur à distinguer les groupes de fonctions).

4. Cliquez sur Taper ici et saisissez `&Quitter`.

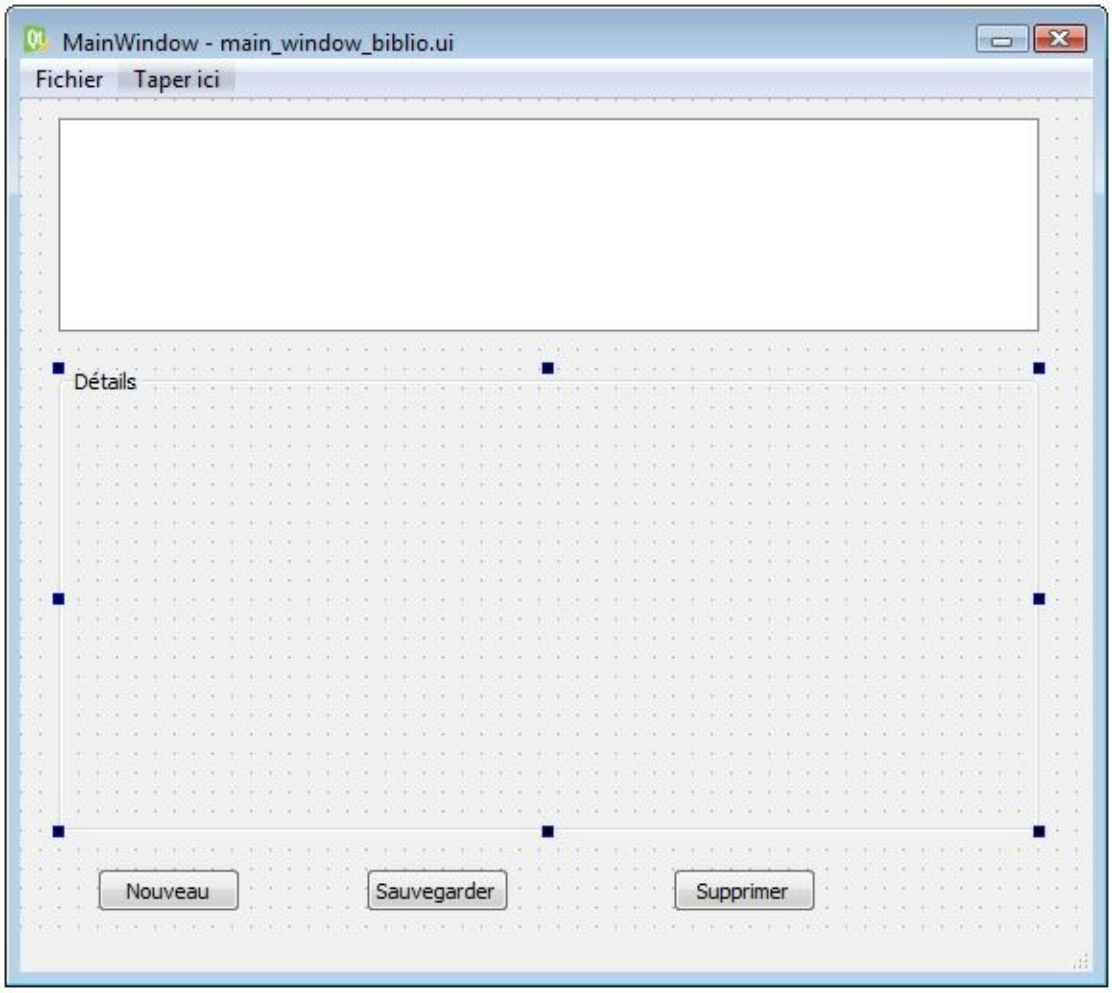
5. Pour avoir des raccourcis clavier rapides, sélectionnez l'élément `Ouvrir` créé, allez dans l'Éditeur de propriétés, sélectionnez la case à droite de shortcut et tapez Ctrl+O au clavier ; faites la même chose pour l'élément `Enregistrer` créé, avec le raccourci Ctrl+S.

***Astuce** > Pour changer l'ordre des éléments ou séparateurs d'un menu, utilisez le glisser-déposer ou Ctrl+flèche haut/bas. Pour supprimer un élément, pressez Suppr au clavier ou sélectionnez Supprimer l'action "…" via le menu contextuel.*

N'oubliez pas de sauvegarder votre travail : Fichier > Enregistrer ou Ctrl+S !

Après ces différentes actions, la fenêtre en construction ressemble à la Figure 6.2.

***Figure 6.2 :** Qt Designer — BiblioApp (étape 1)*



2.3. Placement des champs de saisie

À présent, nous allons placer les éléments spécifiques d'un livre sur la Group Box `Détails` :

- sept Label : libellés fixes ; après double-clic, on saisit respectivement `Titre`, `Auteur(s)`, `Genre`, `Éditeur`, `Année de parution`, `Résumé` et `Prix` ;
- trois Line Edit : champs de texte pour la saisie du titre, des auteurs et de l'éditeur ;
- une Combo Box : liste déroulante pour la saisie du genre ;
- un Date Edit : bouton fléché pour la saisie de l'année de parution ;
- une Double Spin Box : bouton fléché pour la saisie du prix.

On placera chacun de ces éléments à droite des libellés correspondants.

***Astuce** > Qt Designer est une application qui suit les standards et raccourcis habituels : glisser/déposer, copier/couper/coller, défaire/refaire, multisélection, etc. On peut ainsi, de façon assez simple et intuitive, dessiner les fenêtres en procédant à des ajustements successifs. Par ailleurs, Qt Designer intègre la notion de parenté entre widgets ; ainsi, en déplaçant la boîte `Détails`, on peut en un coup déplacer tous ses widgets enfants ; au besoin, on peut même faire un couper/coller de cette boîte.*

***Astuce** > Dans la zone Inspecteur d'objet, un arbre affiche la hiérarchie des widgets de la fenêtre en construction ; cela peut s'avérer très utile pour sélectionner un widget qu'on a perdu visuellement, par exemple s'il est occulté par un autre ou s'il se retrouve en dehors du cadre de la fenêtre.*

Pour la liste déroulante (combo box), nous supposerons que les genres possibles sont connus et fixes. On peut donner les valeurs possibles par double-clic sur l'élément : une boîte de dialogue s'ouvre alors permettant de saisir les choix possibles (bouton +) et de les réordonner (flèches haut/bas).

***Note** > Nous définissons en première position un choix vide (symbolisé ici par des tirets), qui sera la valeur affichée par défaut.*

***Figure 6.3 :** Qt Designer — BiblioApp*

Pour avoir une fenêtre claire et agréable, il y a lieu de faire quelques petites adaptations. Ceci va nous permettre d'introduire l'Éditeur de propriétés.

- Pour changer le nom affiché sur la barre de titre ("MainWindow" par défaut), on

sélectionne la fenêtre principale, en s'aidant éventuellement de l'Inspecteur d'objet ; ensuite on va dans l'Éditeur de propriétés et on modifie la propriété :

  - windowTitle : `BiblioApp`

- On constate que le bouton fléché destiné à l'année de parution (date edit) affiche une date complète. Le format d'affichage de ce widget peut être adapté très facilement via sa propriété displayFormat. On va utiliser cela pour limiter l'affichage à l'année (le jour exact de la parution du livre est souvent inconnu et peu important !). Pour cela, on va sélectionner ce bouton fléché (date edit) et modifier la propriété displayFormat, en bas de l'Éditeur de propriétés :

  - displayFormat : `yyyy` — pour avoir juste l'année en quatre chiffres.

- Le bouton fléché utilisé pour le prix (double spin box) peut être amélioré en éditant quelques-unes de ces propriétés :

  - suffix : `€` — préfixé d'un caractère d'espacement pour bien détacher le symbole monétaire ;

  - maximum : `9999,99` — ceci redimensionne automatiquement le widget pour assurer que tous les chiffres soient affichés ;

  - Alignment - Horizontal : AlignementDroite — c'est mieux pour l'affichage d'un montant ;

  - optionnellement (non fait ici) : buttonSymbols : NoButtons — si on ne juge pas utile d'avoir des boutons d'incrément/décrément pour la saisie du prix.

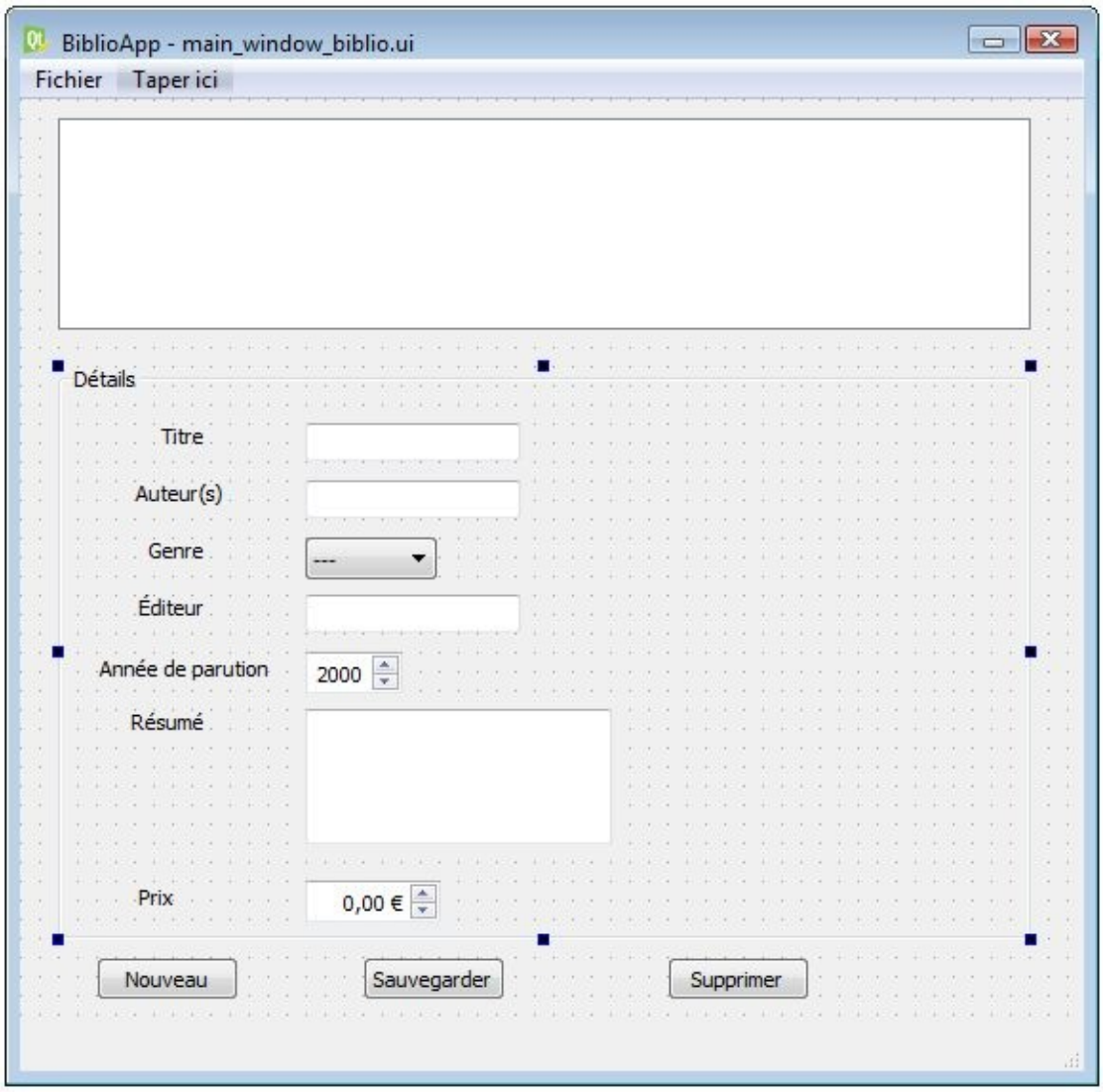
- Tous les libellés peuvent être alignés sur la droite ; lors de la mise en page (voir un peu plus bas), on aura ainsi les libellés serrés au plus près de leur champ de saisie. Pour éviter des manipulations fastidieuses, on peut ici sélectionner les sept widgets QLabel (par exemple, Ctrl+clic sous Windows) et changer l'alignement en un coup :

  - Alignment - Horizontal : AlignementDroite

Mis à part ces petits ajustements cosmétiques et comme déjà signalé, il est inutile de positionner les choses au pixel près : ce sera fait à la prochaine section via des positionneurs (QLayout).

Après toutes ces étapes, la fenêtre principale ressemble à la Figure 6.4.

Figure 6.4 : *Qt Designer — BiblioApp (étape 2)*



2.4. Mise en page

À présent que tous les éléments requis sont là, nous allons terminer le travail en définissant la mise en page.

Attention *> Ce qui suit n'est pas tout à fait trivial quand on débute dans Qt Designer ! Relisez au besoin la Section 5, Mise en page intelligente avec QLayout pour bien comprendre les positionneurs et les espaceurs. Signalons par ailleurs qu'il existe plusieurs techniques possibles pour aboutir au même résultat. Nous donnons ici celle qui nous paraît la plus simple. N'hésitez pas à expérimenter d'autres stratégies et à vous forger votre propre expérience.*

Avant de nous lancer dans la création des QLayout qui prennent en charge le positionnement et le dimensionnement des widgets, nous devons nous prémunir de quelques effets indésirables (ou pour le moins déroutants pour l'utilisateur) : à l'inverse des champs de saisie textuels, il serait préférable que les boutons, les champs `Genre`, `Année` et `Prix` gardent une taille fixe, quelle que soit la taille de la fenêtre principale. Pour faire cela, une technique simple consiste à placer des espaceurs horizontaux : comme déjà expliqué, ceux-ci vont absorber l'espace à proximité des éléments qu'on voudrait garder fixes.

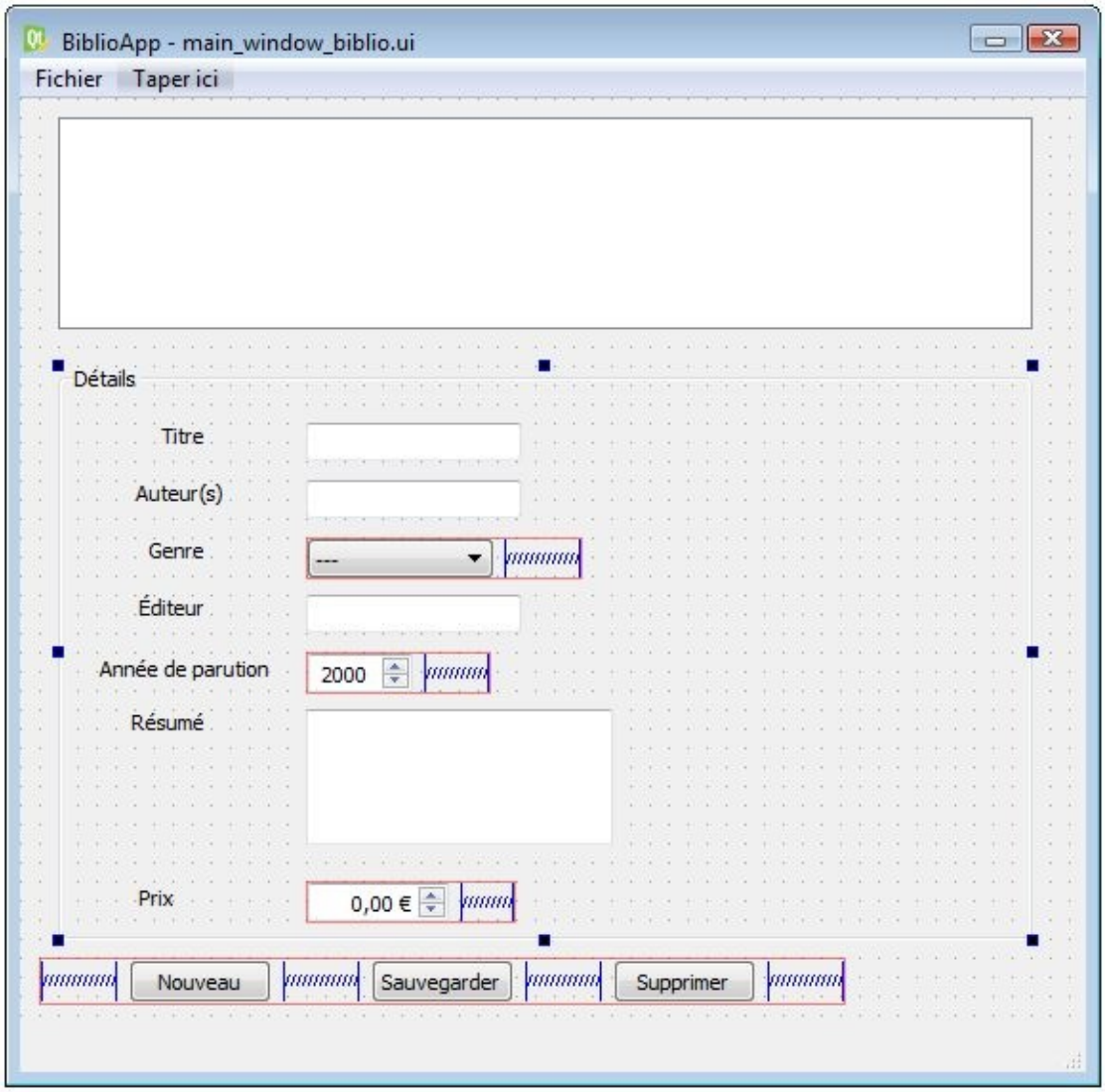
1. Plaçons des Horizontal Spacer comme sur la Figure 6.5 (il y en a sept).

2. Faisons une multisélection du bouton fléché `Genre` avec l'espaceur à sa droite ; ensuite, on active le bouton Mettre en page horizontalement sur la barre d'outils de Qt Designer (également accessible par menu contextuel ou le raccourci Ctrl+1) ; ceci a pour effet de créer un QHBoxLayout autour des widgets sélectionnés, ce qui est représenté par un rectangle rouge.

3. Répétons la même opération Mettre en page horizontalement pour assembler les espaceurs dans les trois autres groupes, `Année de parution`, `Prix` et boutons.

Après ces opérations, notre fenêtre ressemble à celle de la Figure 6.5.

Figure 6.5 : *Qt Designer — BiblioApp (étape 3)*



À ce stade, on ne voit pas encore bien l'intérêt des positionneurs et des espaceurs : les positions et les dimensions des widgets ne s'adaptent pas encore harmonieusement aux dimensions de la fenêtre. C'est normal : on n'a pas encore défini ni la mise en page de la zone `Détails` ni celle de la fenêtre principale. Heureusement, après les étapes un peu laborieuses qui précèdent, ce qui reste à faire est très simple !

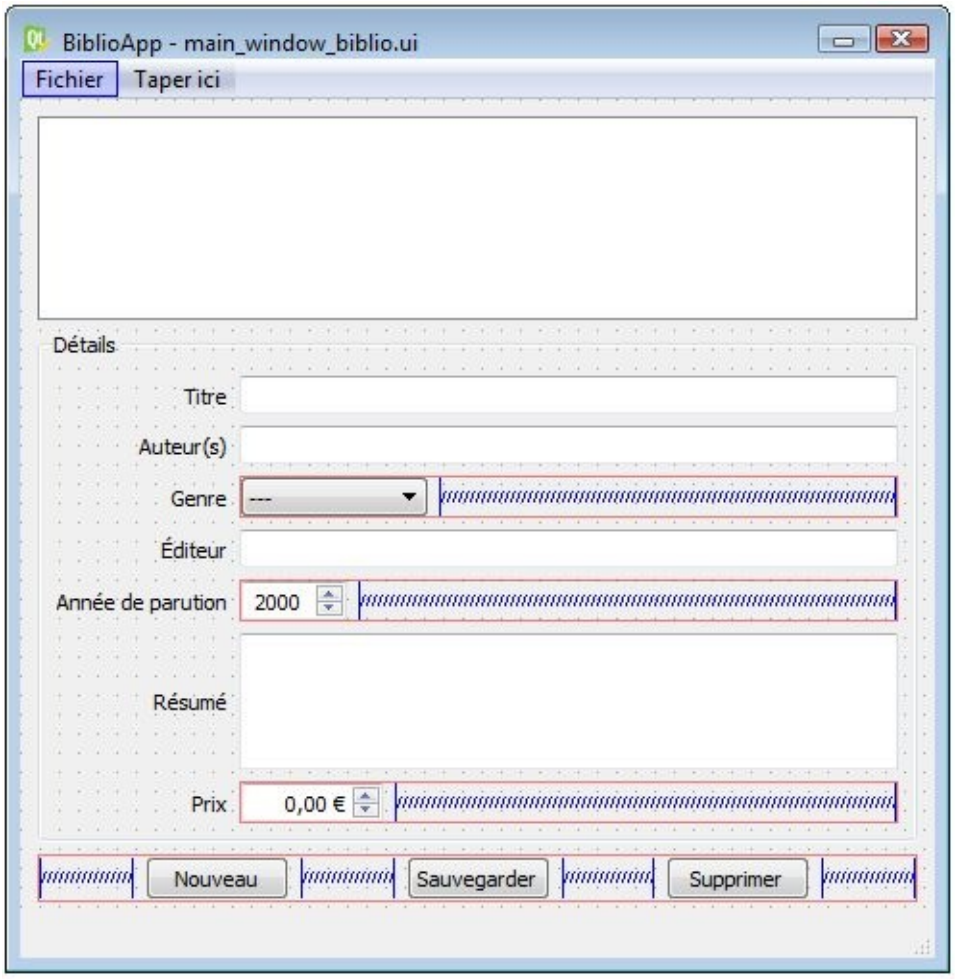
1. On sélectionne la boîte `Détails` et on active le bouton Mettre en page dans une grille sur la barre d'outils (également accessible par menu contextuel ou le raccourci Ctrl+5) ; vu la disposition initiale des widgets enfants de la boîte `Détails`, ceci a pour effet de créer une mise en page à deux colonnes : libellés et champs.

2. On sélectionne la fenêtre principale, en s'aidant au besoin de l'Inspecteur d'objet, et on active le bouton Mettre en page verticalement sur la barre d'outils

(également accessible par menu contextuel ou le raccourci Ctrl+2).

La Figure 6.6 montre le résultat final.

Figure 6.6 : *Qt Designer — BiblioApp (étape 4)*



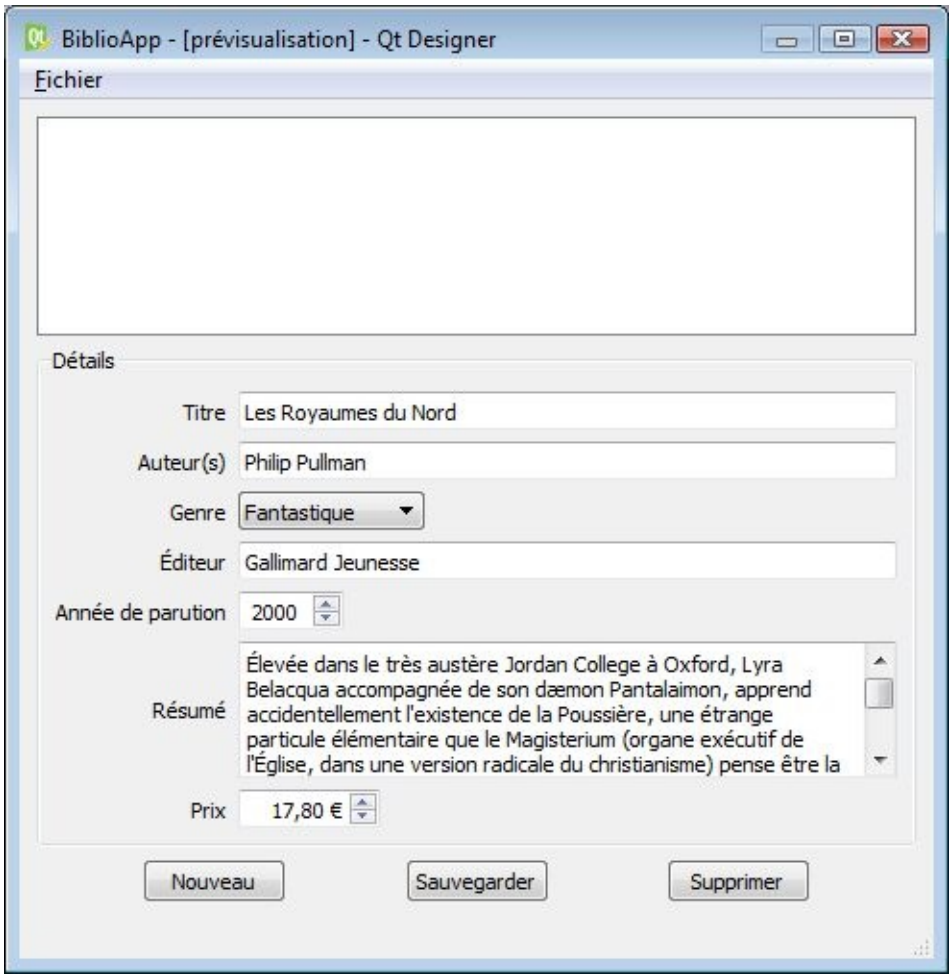
Après quelques efforts, nous avons enfin une fenêtre principale qui présente bien et ceci quelles que soient les dimensions qu'on lui donne. Notez que nous n'avons pas eu à programmer quoi que ce soit !

2.5. Prévisualisation

Pour se faire une meilleure idée de ce qu'on a construit (ou montrer quelque chose de parlant à un futur utilisateur), il existe une fonction incontournable de Qt Designer : la prévisualisation. L'idée est de montrer la fenêtre *telle qu'elle apparaîtra à l'exécution* : on peut dès lors interagir avec tous les widgets définis et se faire une

première idée de l'application. Pour faire cela, activez Formulaire > Prévisualisation… dans le menu principal ou Ctrl+R — un raccourci dont vous ne pourrez bientôt plus vous passer !

Figure 6.7 : *Qt Designer — BiblioApp (prévisualisation)*



Grâce à la prévisualisation, on peut facilement voir les problèmes d'ergonomie les plus flagrants et les corriger.

2.6. Attribution des noms

Une fois que nous sommes satisfaits du résultat visuel, nous allons progressivement quitter Qt Designer et nous diriger vers la programmation de l'application. Pour pouvoir coder, on devra évidemment être capable de nommer dans le programme les widgets sur lesquels on voudra agir. On va dès lors donner un nom identifiant — et de

préférence informatif ! — à chaque widget digne d'intérêt pour l'application BiblioApp (c'est-à-dire, dans notre cas, *tous sauf* les QLabel et le QGroupBox). Pour tout widget sélectionné, vous pouvez voir ce nom en consultant sa propriété objectName. Vous remarquerez que Qt Designer a déjà attribué un nom unique à chaque widget, composé du nom de la classe sans le "Q" suivi éventuellement d'un numéro d'ordre pour éviter les doublons. Pour facilement reconnaître l'objet auquel on a affaire, il est recommandé de garder le préfixe classe et de remplacer le numéro en suffixe par un identifiant fonctionnel. Pour réaliser cela, sélectionnez les widgets un par un et mettez à jour la propriété objectName (à noter : on peut faire ceci aussi via l'Inspecteur d'objet). Nous définirons ainsi :

- les noms des widgets de type *saisie* : `treeViewLivres`, `lineEditTitre`, `lineEditAuteur`, `comboBoxGenre`, `lineEditEditeur`, `dateEditParution`, `plainTextEditResume`, `doubleSpinBoxPrix` ;
- les noms des boutons : `pushBoutonNouveau`, `pushBoutonSauvegarder`, `pushBoutonSupprimer` ;
- la fenêtre principale : `mainWindowBiblio`.

***Note** > Comme nous l'avons vu à la Section 2, Fenêtre principale : la classe QMainWindow, on peut désigner chaque widget de notre classe Python MainWindowBiblio en écrivant simplement* `self.treeViewLivres`, `self.lineEditTitre`, *etc.*

Concluons cette section d'introduction à Qt Designer… Certes, la partie mise en page telle qu'on l'a montrée nécessite un certain savoir-faire et la connaissance de certaines recettes. Cependant, l'acquisition de ce savoir-faire est incontournable quelle que soit la méthode utilisée. À titre de comparaison, imaginons qu'on ait tenté de construire la fenêtre principale via une programmation classique, à l'instar de ce qui a été fait au chapitre précédent : la mise au point de tous les petits détails de présentation aurait probablement nécessité de nombreux cycles de programmation/exécution. Qt Designer offre en définitive une alternative très efficace pour concevoir des GUI sophistiquées avec des mises en pages intelligentes.

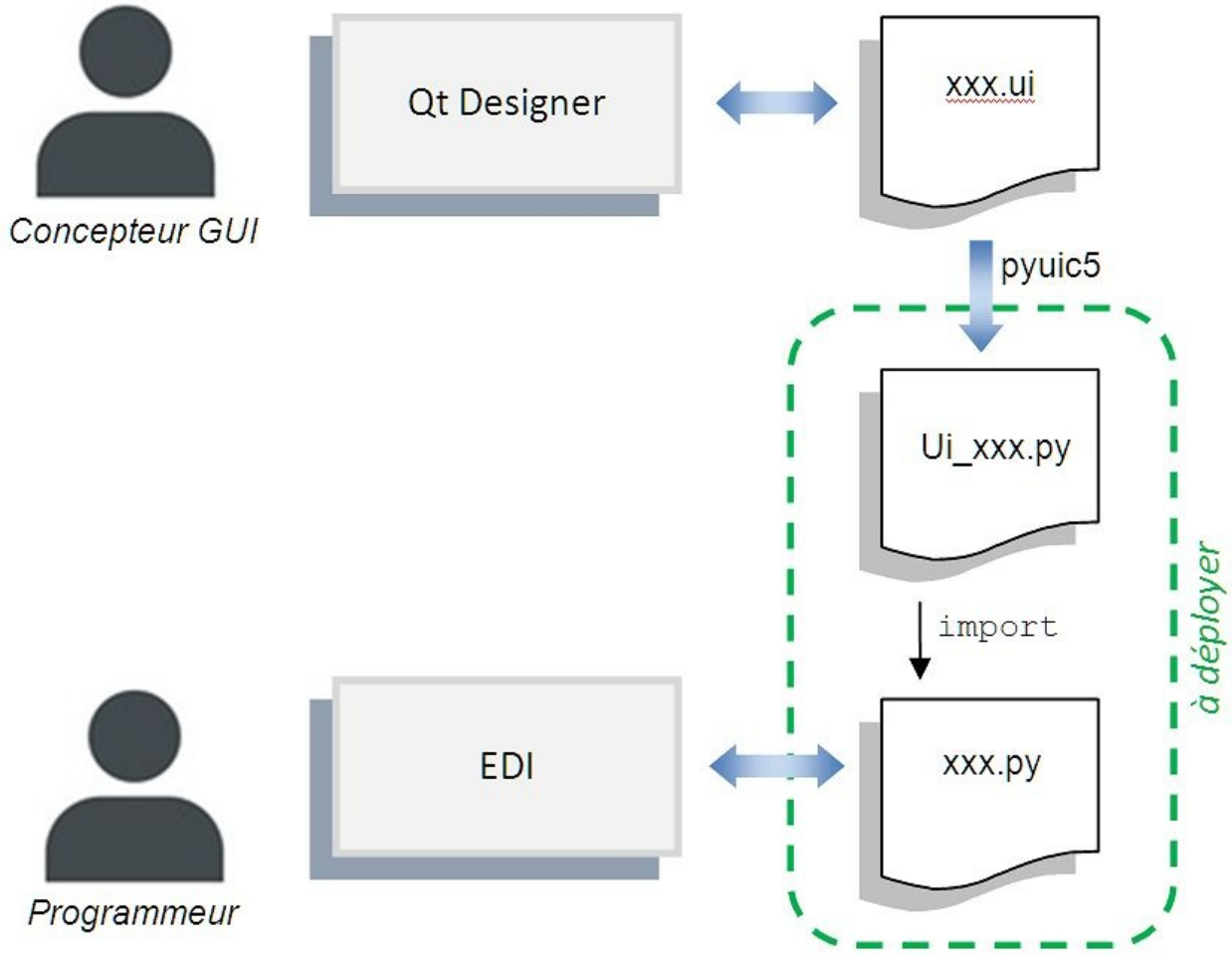
3. Génération du code

Maintenant que nous avons notre `main_window_biblio.ui`, la question se pose : qu'en fait-on ? Le but est de pouvoir injecter toute l'information de ce fichier dans la méthode `MainWindowBiblio.__init__` et se passer ainsi d'avoir à écrire les laborieuses instructions de création/configuration de widgets. Il existe principalement deux techniques pour faire cela :

La génération de code

PyQt fournit un compilateur[5] de fichier `.ui` appelé `pyuic5` ; il se charge de générer un fichier Python en traduisant en instructions les éléments dessinés dans Qt Designer. Le fichier généré n'est, bien sûr, pas destiné à être édité par le programmeur, sous peine de perdre tout ce qui a été modifié à la génération suivante. Le code généré sera intégré dans notre programme via héritage ou composition.

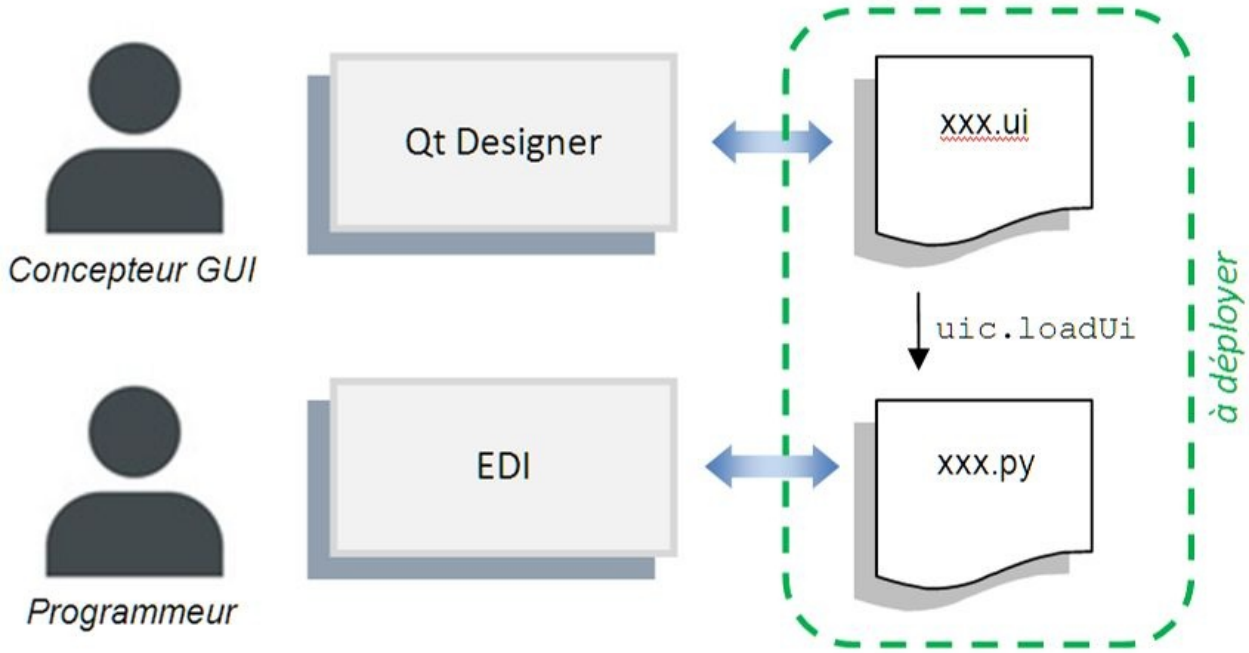
Figure 6.8 : *Technique 1 : génération de code à partir du fichier `.ui`*



Le chargement dynamique

À l'inverse de la technique précédente qui demande une étape avant l'exécution, on peut aussi choisir de charger directement le fichier `.ui` lors de l'exécution de l'application, via les méthodes Python idoines fournies par le module `uic` de PyQt (la méthode `uic.loadUi`, par exemple).

Figure 6.9 : *Technique 2 : chargement dynamique du fichier `.ui`*



Une différence importante à noter entre les deux techniques est que la première ne nécessite pas de livrer de fichiers `.ui` pour les déploiements, alors que la seconde les requiert.

La deuxième technique (chargement dynamique) peut être intéressante pour offrir la possibilité de modifier l'interface utilisateur sans avoir besoin de l'environnement de développement, si ce n'est Qt Designer (ou, pour des changements minimes, un simple éditeur XML). Bien sûr, le chargement dynamique présente des désavantages : il ralentit un peu le démarrage de l'application (lecture et compilation des fichiers `.ui` à chaque fois) ; de plus, toute erreur faite dans le fichier `.ui` n'est détectée qu'à l'exécution de l'application (exception levée par la méthode `uic.loadUi`).

Dans ce qui suit, nous appliquerons la première technique (génération de code), qui est sans doute la plus orthodoxe.

Nous allons générer le code Python à partir de `main_window_biblio.ui`. Dans eric6, sélectionnez ce fichier dans le panneau Feuilles du projet ; ensuite activez Compiler la feuille dans le menu contextuel. C'est tout ! Si vous retournez dans le panneau Sources, vous verrez apparaître un nouveau fichier : `Ui_main_window_biblio.py`. Par curiosité, vous pouvez éditer ce fichier et voir le code Python généré — plus d'une centaine de lignes — auquel nous avons heureusement échappé.

```
# -*- coding: utf-8 -*-
```

```
# Form implementation generated from reading ui file
'C:\Users\Public\Espace_Eric6\biblioapp2\main_window_biblio.ui'
#
# Created by: PyQt5 UI code generator 5.5.1
#
# WARNING! All changes made in this file will be lost!

from PyQt5 import QtCore, QtGui, QtWidgets

class Ui_MainWindowBiblio(object):
    def setupUi(self, MainWindowBiblio):
        MainWindowBiblio.setObjectName("MainWindowBiblio")
        MainWindowBiblio.resize(1008, 1115)
        self.centralwidget = QtWidgets.QWidget(MainWindowBiblio)
        self.centralwidget.setObjectName("centralwidget")
        self.verticalLayout =
QtWidgets.QVBoxLayout(self.centralwidget)
        self.verticalLayout.setObjectName("verticalLayout")
        self.treeViewLivres = QtWidgets.QTreeView(self.centralwidget)
        self.treeViewLivres.setObjectName("treeViewLivres")
        self.verticalLayout.addWidget(self.treeViewLivres)
        self.groupBox = QtWidgets.QGroupBox(self.centralwidget)
        …
…
```

Cette opération de compilation sera à faire chaque fois que vous modifierez le fichier feuille dans Qt Designer. Comme le rappelle le cartouche d'en-tête, il est *fortement déconseillé* de modifier à la main le fichier Python généré, sachant qu'il sera écrasé après toute nouvelle génération de code[6].

Pour compiler les feuilles sans eric6

Vous pouvez compiler un fichier `.ui` en invoquant le programme `pyuic5` fourni par PyQt. Vous spécifiez le nom du fichier source à écrire via l'option `-o` :

```
pyuic5 main_window_biblio.ui -o Ui_main_window_biblio.py
```

Ceci génère un fichier `Ui_main_window_biblio.py` comme vu précédemment. Pour information, pyuic5 propose d'autres options qui sont décrites en faisant :

```
pyuic5 -h
```

Comme le suggère le diagramme de la Figure 6.8, il reste maintenant à importer le fichier généré `Ui_main_window_biblio.py` dans un fichier "à nous", à savoir

`main_window_biblio.py`[7] dans lequel on va écrire la logique de l'application. Voici ce que doit contenir ce fichier, au minimum.

Note *> Pour rappel, ce fichier est disponible dans le projet* `qt-designer` .

```
# -*- coding: utf-8 -*-

# main_window_biblio.py

from PyQt5.QtWidgets import QMainWindow
from Ui_main_window_biblio import Ui_MainWindowBiblio
```

❶ class MainWindowBiblio(QMainWindow,Ui_MainWindowBiblio): ❷ def __init__(self,parent=None): super(MainWindowBiblio,self).__init__(parent) self.setupUi(self) ❸

Voici quelques explications sur ces lignes, qui sont à prendre essentiellement comme une recette :

❶ Nous importons la classe Ui_MainWindowBiblio (générée à partir de la feuille créée dans Qt Designer).

❷ La classe MainWindowBiblio hérite à la fois de la classe PyQt QMainWindow et de Ui_MainWindowBiblio (héritage multiple).

❸ Nous appelons la méthode `setupUi` de Ui_MainWindowBiblio : ceci crée tous les widgets/layouts comme attributs de `self` et donc facilement accessibles (on retrouve les noms qu'on a donnés dans Qt Designer : `self.lineEditTitre`, etc.).

Dans la suite du livre, nous ajouterons toute la logique de l'application en enrichissant progressivement notre classe MainWindowBiblio. Pour donner un aperçu rapide de la démarche, nous allons ici écrire quelques lignes pour réagir aux actions menus `Ouvrir` et `Quitter`.

Lorsque les actions définies sont activées, que ce soit par clic ou raccourci-clavier, elles émettent un signal `triggered`. Nous allons donc définir deux méthodes slots :

- `on_actionOuvrir_triggered` : l'implémentation devra afficher une boîte de dialogue standard permettant la sélection d'un fichier ; à cette fin, on utilisera la

méthode statique `QFileDialog.getOpenFileName` ;

- `on_actionQuitter_triggered` : l'implémentation devra fermer la fenêtre (méthode `close()`) après confirmation par l'utilisateur.

Voici l'implémentation de `on_actionOuvrir_triggered` :

```
# -*- coding: utf-8 -*-

# main_window_biblio.py

…
from PyQt5.QtWidgets import QFileDialog, QMessageBox
from PyQt5.QtCore import pyqtSlot

class MainWindowBiblio(QMainWindow,Ui_MainWindowBiblio):

    …
    @pyqtSlot()
    def on_actionOuvrir_triggered(self):
        (nomFichierBiblio,filtre) = QFileDialog.getOpenFileName(
```

❶ self,"Ouvrir fichier bibliothèque", filter="Bibliothèque (*.bib);; Tout (*.*)") if nomFichierBiblio: ❷ # TODO: trace temporaire à remplacer par la lecture du fichier QMessageBox.information(self,"TRACE", "Fichier à ouvrir:\n\n%s"%nomFichierBiblio)

La méthode statique `QFileDialog.getOpenFileName` permet de spécifier différents filtres sur les extensions. Dans l'exemple, nous supposons que la future extension sera `.bib` mais nous proposons aussi, en spécifiant un second filtre, de voir au besoin tous les fichiers. La méthode renvoie un tuple (nom du fichier choisi,
❶ filtre choisi), dont le second élément a peu d'intérêt. Si l'utilisateur a cliqué sur le bouton OK, on récupère dans le premier élément le nom complet du fichier sélectionné, prêt à être lu par notre programme; sinon, si c'est le bouton Cancel, une chaîne vide est renvoyée et aucun traitement n'est à faire. Le programmeur doit donc toujours tester la chaîne renvoyée avant de faire le traitement.

Ce test assure que l'utilisateur a bien cliqué sur OK ; on affiche une boîte de message indiquant le nom complet du fichier sélectionné. Cette implémentation est bien sûr temporaire puisque, à terme, on veut lire le fichier donné et charger les données des livres.

❷ ***Astuce*** *> Conventionnellement, les programmeurs marquent les parties de code*

restant à modifier avec le message TODO ; eric6 et d'autres EDI proposent une fonction pour retrouver aisément ces commentaires et sauter directement dans la partie de code marquée. Dans eric6, ceci est appelé le Visualisateur de tâches, invoqué avec Alt+Maj+T.

Passons à présent à la seconde méthode slot évoquée plus haut, qui concerne l'action menu `Quitter`. Pour éviter une sortie par erreur, il est classique de demander une confirmation à l'utilisateur. Nous pourrions facilement ajouter cela dans la méthode `on_actionQuitter_triggered` , c'est-à-dire lorsque l'utilisateur ferme l'application par le bouton Quitter du menu. Il est cependant plus intelligent de procéder autrement car ce garde-fou serait bienvenu dans *tous* les cas, qu'on ferme la fenêtre via le menu de l'application, via le menu système ou via la petite croix. Pour faire cela, nous surchargeons la méthode `closeEvent` appelée automatiquement dès lors que la sortie de l'application est demandée, peu importe le mécanisme qui déclenche cette fermeture ; cette méthode passe un événement en argument que la méthode devra choisir soit d'accepter (on quitte effectivement), soit d'ignorer (la sortie est abandonnée) en invoquant respectivement les méthodes `accept()` ou `ignore()`.

Ces quelques explications devraient vous aider à comprendre les méthodes qui suivent, à ajouter à notre classe MainWindowBiblio :

```
# -*- coding: utf-8 -*-

# main_window_biblio.py

…

class MainWindowBiblio(QMainWindow,Ui_MainWindowBiblio):

    …
    @pyqtSlot()
    def on_actionQuitter_triggered(self):
        self.close()

    def closeEvent(self,event):
        messageConfirmation = "Êtes-vous sûr de vouloir quitter
BibliApp ?"
        reponse = QMessageBox.question(self,"Confirmation",

messageConfirmation,QMessageBox.Yes,QMessageBox.No)
        if reponse == QMessageBox.Yes:
            event.accept()
        else:
            event.ignore()
```

Il est temps de tester tout ceci. Pour exécuter l'application via eric6, activez le menu Lancer > Lancer projet… (Maj+F2), ensuite OK. Pour rappel, les fois suivantes, il suffira de presser F4 au clavier pour lancer l'application. Sans eric6, lancez la commande

```
python start_app.py
```

Vous pouvez vérifier le bon comportement de l'application en activant les fonctions Ouvrir... et Quitter de différentes manières.

Le générateur de code d'eric6

eric6 propose une fonction Générateur de code… dans le menu contextuel d'une feuille. Cette fonction permet de générer un premier `main_window_biblio.py` contenant le squelette du code de l'application, c'est-à-dire, en gros, la classe MainWindowBiblio avec son constructeur, les signatures de méthodes slots sans leur implémentation (voir listings plus hauts). Nous n'avons pas utilisé cette fonction car elle comporte, dans eric6, quelques imperfections gênantes, notamment concernant les imports en Python 3. Cette fonction est tout à fait contournable — et contournée dans le présent chapitre ! Voici néanmoins quelques explications sommaires sur cette fonction qui pourrait s'améliorer dans une prochaine version d'eric.

Attention *> Cette fonction Générateur de code… est une fonction d'eric6, pas de PyQt ; elle ne doit pas être confondue avec la fonction Compiler la feuille que nous avons vue et qui génère le fichier `Ui_main_window_biblio.py` en appelant l'outil `pyuic5` de PyQt.*

Après activation de Générateur de code…, un dialogue s'ouvre : il permet de définir le nom de la classe à générer (MainWindowBiblio), le nom du fichier Python à écrire (`main_window_biblio.py`) et, par widget, les signaux qui nous intéressent (en cochant des cases). Après confirmation, le fichier spécifié est créé, avec la méthode `__init__`, les méthodes slots et les connexions aux signaux choisis. Le programmeur peut alors éditer ce fichier et remplacer les corps de méthodes vides (`raise NotImplementedError`) par ce qu'il faut.

---

[5] Pour les connaisseurs, on pourrait utiliser ici le terme *transpileur* (parfois appelé

aussi *transcompilateur*) : PyQt génère du code Python, qui est un langage qui a (au moins) le même niveau d'abstraction que le fichier source .ui. Ceci dit, le terme "compilateur" utilisé dans la documentation PyQt et eric6 n'est pas faux : il est simplement plus général.

[6] Le lecteur attentif aura remarqué à la fin de Ui_main_window_biblio.py un bloc d'instructions suggérant un programme principal. Oui … le fichier généré peut être exécuté comme un script individuel : la fenêtre apparaît alors exactement comme en mode prévisualisation dans Qt Designer. La différence est qu'ici le code exécuté pour créer la fenêtre est du Python. Comme nous allons le voir, cette facilité n'est pas vraiment utile en fait ; cela sert juste à s'assurer à peu de frais que la compilation de la feuille a bien marché.

[7] Si ce fichier est récupéré du chapitre précédent, on remplace son contenu ; sinon, on le crée.

Programmer par modèle-vue

Niveau : intermédiaire

Objectifs : programmer une application PyQt/Qt Widgets en utilisant le patron de conception MVD

Prérequis : Créer une première application, Développer avec Qt Designer

Il y a plusieurs manières de programmer une application de type interface utilisateur. On considère comme une bonne pratique de bien séparer les données de la manière de les représenter. Ce concept très important est incarné dans plusieurs patrons de conception : observateur, modèle-vue-contrôleur (MVC), etc. Qt et PyQt proposent le patron de conception modèle-vue-délégué.

L'approche modèle-vue-délégué

Selon cette approche, le *modèle* définit les données, la *vue* les affiche à l'utilisateur et le *délégué* se charge plus particulièrement de l'affichage d'un élément donné, mais aussi de son édition. Ainsi, grâce à cette conception, pour proposer plusieurs manières de visualiser les mêmes données, il suffit de changer la vue : un tableau des livres, une grille des couvertures, etc.

***Note** > Pour plus d'informations sur le patron MVD, voir aussi le chapitre Utiliser la méthodologie modèle - vue (Développement d'une application avec Qt Quick).*

Dans ce qui suit, nous allons voir comment ce patron de conception MVD se concrétise en PyQt à travers le développement de l'application BiblioApp. Dans un premier temps, nous mettrons en place les fondations de l'application en présentant successivement : les données, le modèle, la vue et le délégué. Puis nous ajouterons les fonctionnalités nécessaires en enrichissant progressivement les objets précités. Au final, nous aurons une application opérationnelle, munie des fonctions classiques d'une interface utilisateur.

***Note** > Le code source final de ce chapitre est disponible dans le projet `modele-vue`*

 .

1. Données

Pour définir les données de notre bibliothèque, il faut commencer par définir ce qu'est un livre. On peut bien entendu écrire une petite classe Python `Livre` avec un constructeur `__init__` qui va stocker les différentes informations d'un livre donné. Ceci dit, comme on a juste besoin d'un simple conteneur de données, il est plus pratique d'utiliser la fonction Python `namedtuple` qui fait cela en une ligne :

```
from collections import namedtuple
Livre = namedtuple('Livre', ('titre', 'auteur', 'editeur', 'genre',
                             'annee', 'resume', 'prix'))
```

Cette construction permet aussi d'accéder aux attributs d'un livre par nom ou par indice. Notons que nous avons mis en premier les trois attributs à placer dans la liste (titre, auteur et éditeur) : ce n'est pas indispensable, mais nous verrons que ça simplifie les choses.

Sur la base de cette définition, notre structure de données sera une liste de livres qu'on pourra, par exemple, créer de la façon suivante :

```
livres = [ Livre("Une étude en rouge", "Conan Doyle", "Hachette",
"Policier",
                 1888, "...", 13.),
           Livre("Le Horla", "Guy de Maupassant", "Gallimard",
"Fantastique",
                 1887, "...", 11.),
           Livre("Napoléon", "André Castelot", "Perrin",
"Biographie",
                 2008, "...", 24.) ]
```

Nous verrons plus loin comment sauvegarder et récupérer ces données sur un fichier (ceci n'est pas du tout important à ce stade et nous pourrons le réaliser très simplement en Python).

2. Modèle

Pour rentrer dans le canevas vue-liste de PyQt, il faut à présent définir notre liste de livres selon un modèle tabulaire, adressable cellule par cellule : ceci veut dire qu'on doit être capable de répondre aux requêtes de la vue demandant, pour une ligne `x` donnée et une colonne `y` donnée, l'attribut d'indice `y` du livre d'indice `x`. Pour cela, on va définir une classe ModeleTableBiblio qui hérite de QAbstractTableModel ; cette classe maintient un attribut `livres` avec la liste décrite plus haut.

Le contrat à respecter quand on hérite de QAbstractTableModel consiste à implémenter les méthodes suivantes :

- `headerData` : titres des colonnes ;
- `columnCount` : nombre de colonnes ;
- `rowCount` : nombre de lignes ;
- `data` : l'attribut d'indice `y` du livre d'indice `x`.

Avec ces éléments en main, nous sommes prêts à écrire la classe ModeleTableBiblio, à placer dans un nouveau fichier `modele_biblio.py` :

```
# -*- coding: utf-8 -*-
# modele_biblio.py
# version de base: liste fixe de livres

from PyQt5.QtCore import Qt, QAbstractTableModel, QModelIndex,
QVariant

from collections import namedtuple

Livre = namedtuple('Livre', ('titre', 'auteur', 'editeur', 'genre',
                             'annee', 'resume', 'prix') )

class ModeleTableBiblio(QAbstractTableModel):

    def __init__(self,livres):
        super(ModeleTableBiblio,self).__init__()
        self.titresColonnes = ("Titre", "Auteur", "Éditeur")
```

❶ self.livres = livres def headerData(self,section,orientation,role): if role == Qt.DisplayRole and orientation == Qt.Horizontal: return self.titresColonnes[section]

return QVariant() def columnCount(self,parent): return len(self.titresColonnes) def rowCount(self,parent): return len(self.livres) def data(self,index,role): ❷ if role == Qt.DisplayRole and index.isValid(): return (self.livres[index.row()][index.column()]) return QVariant()

❶ Pour faciliter les évolutions futures et avoir un style plus déclaratif, nous avons mis les titres de colonnes dans l'attribut `titresColonnes` : ainsi, les méthodes `headerData` et `columnCount` ne doivent plus être modifiées.

La méthode `data` est assez simple, sur base de notre structure de données : on reçoit un objet `index` (instance de QModelIndex) qui contient :

- `index.row()` : un numéro de ligne qui est l'indice d'un livre dans la liste ;
- `index.column()` : un numéro de colonne qui est l'indice d'un attribut d'un livre : selon la définition du tuple nommé `Livre` vue plus haut, on a 0 pour le titre, 1 pour l'auteur et 2 pour l'éditeur).

❷

Comme nos livres sont stockés dans une liste de tuples nommés, l'expression `self.livres[index.row()][index.column()]` renvoie en un coup l'élément demandé. Par exemple, quand la vue demandera *Que dois-je afficher dans la cellule (ligne 1, colonne 2) ?*, le modèle répondra `self.livres[1][2]`, c'est-à-dire *Gallimard*.

3. Vue et délégué

À présent que nous avons notre modèle de bibliothèque, nous sommes prêts à enrichir la classe MainWindowBiblio pour représenter visuellement ce modèle. Pour débuter en douceur, on va se fixer deux objectifs, qui concernent la vue et le délégué :

1. La liste doit afficher les livres du modèle (objectif de la vue).
2. Les champs doivent afficher les détails du livre sélectionné (objectif du délégué).

Concrètement, pour faire cela, on va :

- créer une instance de ModeleTableBiblio avec (pour l'exemple) la liste de trois livres donnés plus haut ;
- déclarer à la vue `treeViewLivres` qu'elle représente ce modèle en invoquant la méthode `setModel` ;
- réagir au signal `selectionChanged` pour afficher les détails du livre sélectionné.

Voici le code qui réalise ces points (à compléter dans `main_window_biblio.py`).

```
# -*- coding: utf-8 -*-
# main_windows_biblio.py

from PyQt5.QtCore import pyqtSlot, QDate
from PyQt5.QtWidgets import QMainWindow, QMessageBox, QFileDialog

from Ui_main_window_biblio import Ui_MainWindowBiblio

from modele_biblio import Livre, ModeleTableBiblio

class MainWindowBiblio(QMainWindow, Ui_MainWindowBiblio):

    def __init__(self, parent=None):
        super(MainWindowBiblio,self).__init__(parent)
        self.setupUi(self)
        # code de test à supprimer …
```

❶ livresTest = [Livre("Une étude en rouge","Conan Doyle", …), …]
self.modeleTableBiblio = ModeleTableBiblio(livresTest) ❷
self.treeViewLivres.setModel(self.modeleTableBiblio) ❸
self.treeViewLivres.selectionModel().selectionChanged.connect(

self.on_treeViewLivres_selectionChanged) ❹ def on_treeViewLivres_selectionChanged(self,selected,deselected): ❺ indexesSelection = selected.indexes() if len(indexesSelection) > 0: self.indexSelection = indexesSelection[0] self.indiceLivreSelectionne = self.indexSelection.row() self.afficheLivre(self.modeleTablesBiblio.livres[self.indiceLivreSelectionne]) def afficheLivre(self,livre): ❻ self.lineEditTitre.setText(livre.titre) self.lineEditAuteur.setText(livre.auteur) self.comboBoxGenre.setCurrentText(livre.genre) self.lineEditEditeur.setText(livre.editeur) self.dateEditAnnee.setDate(QDate(livre.annee,1,1)) self.plainTextEditResume.setPlainText(livre.resume) self.doubleSpinBoxPrix.setValue(livre.prix)

❶ Comme on n'a pas encore implémenté de fonction pour créer des livres à partir de données saisies, on écrit en dur une petite liste de trois livres. Ce code de test temporaire sera supprimé très bientôt : on optera pour une lecture de fichier ou d'une base de données.

❷ On crée notre modèle `ModeleTableBiblio` sur la base de la liste de livres construite…

❸ … et on déclare ce modèle à la vue `self.treeViewLivres` en invoquant `setModel` : sachant que notre classe ModeleTableBiblio a été programmée en respectant le protocole demandé, ceci suffit à afficher les livres dans la liste.

❹ Ce qui est un peu plus compliqué, c'est la manière de retrouver le livre sélectionné pour le passer à `afficheLivre`. Il faut d'abord mettre la main sur le signal émis lorsque l'utilisateur sélectionne un livre dans la liste. On l'obtient à travers une chaîne d'objets : on part du widget affichant la liste des livres `self.treeViewLivres` ; ensuite, on récupère l'objet qui gère la sélection des éléments de cette liste par `selectionModel()` ; l'objet renvoyé est une instance de QItemSelectionModel, duquel on obtient enfin le signal `selectionChanged`. On a donc le signal qui est émis lorsque la sélection change (par exemple lorsque l'utilisateur clique sur un livre). Pour réagir à ce signal, on le connecte à la méthode slot `on_treeViewLivres_selectionChanged`.

La méthode `on_treeViewLivres_selectionChanged` reçoit deux arguments

fournis par le signal précité : `selected` et `deselected` ; le premier indique les éléments sélectionnés et le second les éléments qui viennent d'être désélectionnés.

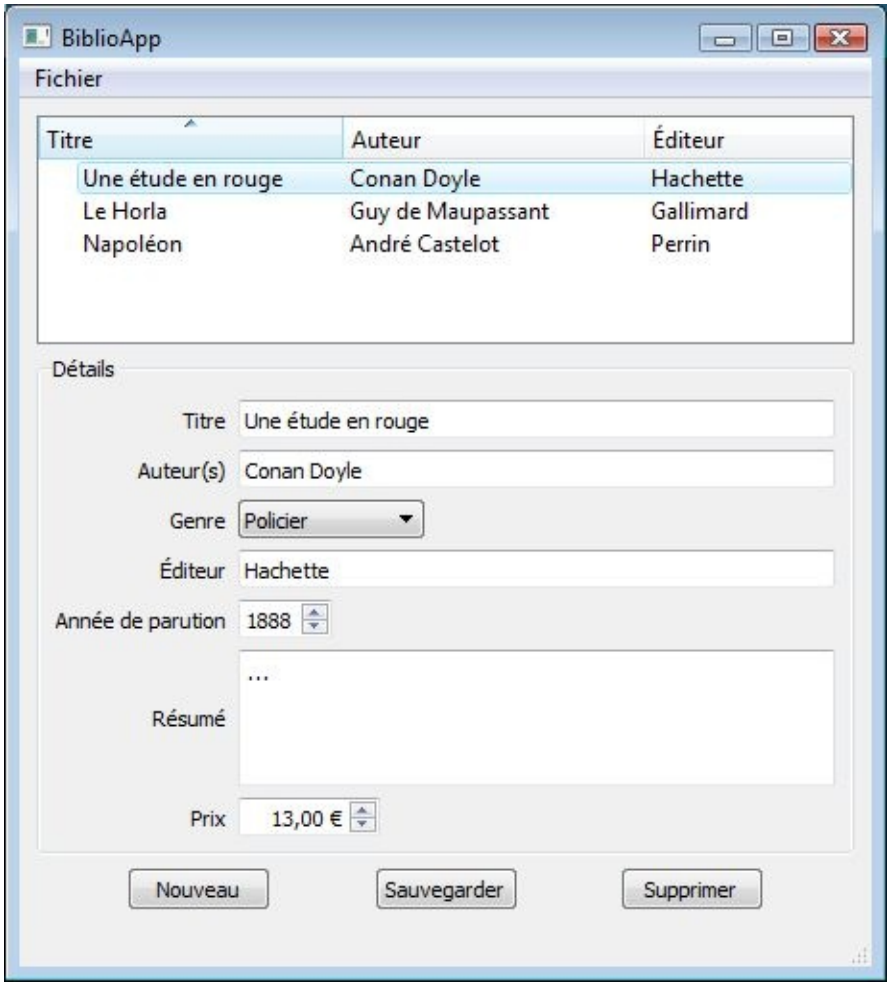
❺ À ce stade, seul le premier argument est intéressant. À partir de celui-ci, on peut retrouver l'indice de la ligne sélectionnée en faisant en cascade les opérations indiquées : extraction des indices, vérification qu'il y a bien une sélection, extraction de l'index unique duquel on peut finalement obtenir l'indice de colonne.

❻ La méthode `afficheLivre` est assez simple à comprendre : on passe un livre (tuple nommé tel que défini dans le modèle) et on affiche chacun de ses attributs dans le widget correspondant (voir la documentation des classes widgets QLineEdit, QComboBox, QDateEdit, QPlainTextEdit et QDoubleSpinBox).

Évidemment, tout cela est un peu laborieux de prime abord ; néanmoins cette sophistication n'est pas gratuite de la part des concepteurs de Qt : dans des applications plus élaborées, ces méthodes permettent de gérer les sélections multiples, par cellules, groupes de cellules et bien d'autres choses encore (pour en savoir plus, voir la documentation de la classe QItemSelectionModel).

Il est temps de tester l'application. On peut voir que les trois livres *codés en dur* s'affichent bien dans la liste ; on peut consulter leurs détails en sélectionnant chaque livre à la souris ou via les touches flèches du clavier.

Figure 7.1 : *BiblioApp v0.10*


BiblioApp
Fichier
Titre
Auteur
Éditeur
Une étude en rouge
Conan Doyle
Hachette
Le Horla
Guy de Maupassant
Gallimard
Napoléon
André Castelot
Perrin
Détails
Titre
Une étude en rouge
Auteur(s)
Conan Doyle
Genre
Policier
Éditeur
Hachette
Année de parution
1888
Résumé
...
Prix
13,00 €
Nouveau
Sauvegarder
Supprimer

4. Gestion des événements

Il reste à présent à compléter l'application pour gérer effectivement notre bibliothèque, en réagissant à l'activation des boutons (ajout, modification, suppression) et des actions du menu (ouvrir/enregistrer un fichier).

On va procéder en deux temps, en suivant le canevas MVD :

1. ajouter des méthodes sur notre modèle (classe ModeleTableBiblio), pour ajouter/supprimer/modifier un livre et sauvegarder/lire nos données sur fichier ;

2. ajouter des méthodes dans la fenêtre principale (classe MainWindowBiblio), pour réagir aux événements en invoquant les méthodes précitées du modèle.

Commençons par enrichir notre classe ModeleTableBiblio.

Choix du format de fichier

Pour enregistrer notre bibliothèque sur fichier, nous avons choisi le format JSON. Ceci est juste un choix pragmatique, sachant que ce format est portable, qu'il permet de sérialiser facilement des structures de données simples et qu'il est supporté via le module standard Python `json`. JSON sera présenté plus en détail au chapitre Présentation de JavaScript, Section 5, Structures de données. Alternativement, on aurait pu choisir par exemple :

- le format XML, supporté par le paquet `xml` de Python — standard, ouvert mais verbeux et plus long à coder ;

- le format CSV (*comma separated values*), supporté par le module `csv` de Python — simple mais… attention à bien choisir le séparateur pour qu'il soit absent des données (la virgule n'est certainement pas le bon choix !) ;

- le format *pickle* défini par le module Python `pickle` — capable de représenter des structures de données plus complexes, mais qui est non standard en dehors de Python.

***Note** > Signalons que, au chapitre Accès à une base de données, nous verrons par ailleurs comment utiliser une base de données au lieu d'un fichier.*

```
# -*- coding: utf-8 -*-
# modele_biblio.py
# version finale

…

import json

class ModeleTableBiblio(QAbstractTableModel):

    …

    def enregistreDansFichier(self,nomFichierBiblio):
```

❶ with open(nomFichierBiblio,'w') as f: json.dump(self.livres,f) @staticmethod def creeDepuisFichier(nomFichierBiblio): ❷ with open(nomFichierBiblio,'r') as f: attributsLivres = json.load(f) livres = [Livre(*attrLivre) for attrLivre in attributsLivres] return ModeleTableBiblio(livres) def ajouteLivre(self,livre): ❸ indiceLivre = len(self.livres) self.beginInsertRows(QModelIndex(),indiceLivre,indiceLivre) self.livres.append(livre) self.endInsertRows() def supprimeLivre(self,indiceLivre): ❹ self.beginRemoveRows(QModelIndex(),indiceLivre,indiceLivre) del self.livres[indiceLivre] self.endRemoveRows() def remplaceLivre(self,indiceLivre,livre): ❺ self.livres[indiceLivre] = livre self.dataChanged.emit(self.createIndex(indiceLivre,0), self.createIndex(indiceLivre,2))

Détaillons chaque méthode une par une.

❶ `enregistreDansFichier` : la méthode `json.dump` transforme notre structure de données en une longue chaîne de caractères au format JSON et l'écrit sur le fichier donné ;

❷ `creeDepuisFichier` : c'est le symétrique de la méthode précédente ; on lit le contenu JSON du fichier donné, on obtient une liste de listes (`attributsLivres`) via la méthode `json.load` ; on définit une liste en compréhension pour transformer cela en liste de tuples nommés (`livres`) ; on passe cette liste au constructeur de `ModeleTableBiblio` ; la méthode est déclarée statique car elle crée une nouvelle instance, sans avoir besoin de `self`.

`ajouteLivre` : on ajoute le nouveau livre en fin de liste via `append` ; pour

❸ respecter le protocole de QAbstractTableModel, on doit invoquer la méthode `beginInsertRows` avec les indices de début et de fin des lignes insérées; ici, il n'y a qu'une nouvelle ligne : elle a pour indice le nombre de livres avant ajout (comme en Python, les indices commencent à zéro) ; on termine avec `endInsertRows`, qui signale la mise à jour aux vues concernées ;

❹ `supprimeLivre` : on supprime de la liste le livre de l'indice donné ; à nouveau, suivant un protocole similaire, on invoque `beginRemoveRows` et `endRemoveRows` ;

❺ `remplaceLivre` : on remplace le livre de l'indice donné par un nouveau livre ; pour signaler cela aux vues concernées, on émet un signal `dataChanged` ; il faut spécifier la plage d'indices des éléments susceptibles d'avoir changé : depuis (indiceLivre,0) jusque (indiceLivre,2), comme on ne touche qu'une seule ligne et qu'il y a trois colonnes.

À présent que le modèle a été muni des fonctions de mise à jour, on est prêt à enrichir la classe MainWindowBiblio pour appeler ces fonctions en réponse aux événements clics sur boutons ou activations de menus.

Attention *> Pour qu'il n'y ait pas d'ambiguïté dans ce qui suit, vous devez bien comprendre la distinction faite — arbitrairement, il est vrai — entre les termes "sauvegarder" et "enregistrer" : le bouton Sauvegarder sert à placer un nouveau livre dans la liste à partir des champs de saisie ou à mettre à jour un livre existant à partir de ces mêmes champs, le menu Enregistrer, quant à lui, sert à sauvegarder la liste complète sur fichier.*

Voici l'implémentation revue et complétée de la classe MainWindowBiblio.

```
# -*- coding: utf-8 -*-
# main_windows_biblio.py

from PyQt5.QtCore import pyqtSlot, QDate
from PyQt5.QtWidgets import QMainWindow, QMessageBox, QFileDialog

from Ui_main_window_biblio import Ui_MainWindowBiblio

from modele_biblio import Livre, ModeleTableBiblio

class MainWindowBiblio(QMainWindow, Ui_MainWindowBiblio):
```

```
    def __init__(self, parent=None):
        super(MainWindowBiblio,self).__init__(parent)
        self.setupUi(self)
        self.nomFichierBiblio = None
```

❶ self.modeleTableBiblio = ModeleTableBiblio([]) ❷
self.treeViewLivres.setModel(self.modeleTableBiblio)
self.treeViewLivres.selectionModel().selectionChanged.connect(
self.on_treeViewLivres_selectionChanged) self.modificationAEnregistrer(False) ❸
self.dateEditAnnee.setMinimumDate(QDate(101,1,1)) ❹
self.dateEditAnnee.setSpecialValueText(" ")
self.doubleSpinBoxPrix.setMinimum(-0.01)
self.doubleSpinBoxPrix.setSpecialValueText(" ") self.effaceLivre()
self.pushButtonSupprimer.setEnabled(False) def
modificationAEnregistrer(self,fichierNonEnregistre): ❺ self.fichierNonEnregistre =
fichierNonEnregistre titre = "BiblioApp" if self.nomFichierBiblio is not None: titre +=
" - " + self.nomFichierBiblio if self.fichierNonEnregistre: titre += " *"
self.setWindowTitle(titre) self.actionEnregistrer.setEnabled(fichierNonEnregistre) def
on_treeViewLivres_selectionChanged(self,selected,deselected): indexesSelection =
selected.indexes() if len(indexesSelection) == 0: ❻ self.effaceLivre()
self.pushButtonSupprimer.setEnabled(False) else: indexSelection =
indexesSelection[0] indiceLivreSelectionne = indexSelection.row()
self.afficheLivre(self.modeleTableBiblio.livres[indiceLivreSelectionne])
self.pushButtonSupprimer.setEnabled(True) def effaceLivre(self): for lineEdit in
(self.lineEditTitre, self.lineEditAuteur, self.lineEditEditeur): lineEdit.setText("")
self.comboBoxGenre.setCurrentIndex(0) self.plainTextEditResume.setPlainText("")
self.dateEditAnnee.setDate(self.dateEditAnnee.minimumDate()) ❼
self.doubleSpinBoxPrix.setValue(self.doubleSpinBoxPrix.minimum()) def
afficheLivre(self,livre): self.lineEditTitre.setText(livre.titre)
self.lineEditAuteur.setText(livre.auteur)
self.comboBoxGenre.setCurrentText(livre.genre)
self.lineEditEditeur.setText(livre.editeur)
self.dateEditAnnee.setDate(QDate(livre.annee,1,1))
self.plainTextEditResume.setPlainText(livre.resume)
self.doubleSpinBoxPrix.setValue(livre.prix) def closeEvent(self,event): if not
self.fichierNonEnregistre: ❽ event.accept() else: messageConfirmation = "Êtes-vous
sûr de vouloir quitter " \ "BibliApp sans sauvegarder le fichier ?" reponse =
QMessageBox.question(self,"Confirmation",
messageConfirmation,QMessageBox.Yes,QMessageBox.No) if reponse ==
QMessageBox.Yes: event.accept() else: event.ignore() @pyqtSlot() def
on_actionQuitter_triggered(self): self.close() @pyqtSlot() def
on_actionOuvrir_triggered(self): if self.fichierNonEnregistre: ❾ messageConfirmation

= "Modifications en cours.\n\n" \ "Êtes-vous sûr de vouloir continuer" \ " sans enregistrer le fichier ?" reponse = QMessageBox.question(self,"Confirmation", messageConfirmation,QMessageBox.Yes,QMessageBox.No) if reponse == QMessageBox.No: return (nomFichierBiblio,filtre) = QFileDialog.getOpenFileName(self,"Ouvrir fichier bibliothèque", filter="Bibliothèque (*.bib);; Tout (*.*)") if nomFichierBiblio: self.modeleTableBiblio = ModeleTableBiblio.creeDepuisFichier(❿ nomFichierBiblio) self.treeViewLivres.setModel(self.modeleTableBiblio) self.treeViewLivres.selectionModel().selectionChanged.connect(self.on_treeViewLivres_selectionChanged) self.nomFichierBiblio = nomFichierBiblio self.modificationAEnregistrer(False) @pyqtSlot() def on_actionEnregistrer_triggered(self): if self.nomFichierBiblio is None: ⓫ (nomFichierBiblio,filtre) = QFileDialog.getSaveFileName(self,"Enregistrer fichier", filter="Bibliothèque (*.bib);; Tout (*.*)") if nomFichierBiblio: self.nomFichierBiblio = nomFichierBiblio if self.nomFichierBiblio is not None: ⓬ self.modeleTableBiblio.enregistreDansFichier(self.nomFichierBiblio) self.modificationAEnregistrer(False) @pyqtSlot() def on_pushButtonNouveau_clicked(self): ⓭ self.treeViewLivres.selectionModel().clearSelection() self.effaceLivre() @pyqtSlot() def on_pushButtonSauvegarder_clicked(self): ⓮ livre = Livre(titre = self.lineEditTitre.text(), auteur = self.lineEditAuteur.text(), editeur = self.lineEditEditeur.text(), genre = self.comboBoxGenre.currentText(), annee = self.dateEditAnnee.date().year(), resume = self.plainTextEditResume.toPlainText(), prix = self.doubleSpinBoxPrix.value()) selectionModel = self.treeViewLivres.selectionModel() indexesSelectionnes = selectionModel.selectedRows() if len(indexesSelectionnes) == 0: ⓯ self.modeleTableBiblio.ajouteLivre(livre) self.on_pushButtonNouveau_clicked() else: ⓰ indiceLivreSelectionne = indexesSelectionnes[0].row() self.modeleTableBiblio.remplaceLivre(indiceLivreSelectionne, livre) self.modificationAEnregistrer(True) @pyqtSlot() def on_pushButtonSupprimer_clicked(self): ⓱ selectionModel = self.treeViewLivres.selectionModel() indexesSelectionnes = selectionModel.selectedRows() if len(indexesSelectionnes) > 0: indiceLivreSelectionne = indexesSelectionnes[0].row() self.modeleTableBiblio.supprimeLivre(indiceLivreSelectionne) self.modificationAEnregistrer(True)

Voici des explications sur les points nouveaux importants dans le listing qui précède.

L'attribut `nomFichierBiblio` est le nom complet du fichier bibliothèque ouvert ; il est mis à `None` au début pour signifier qu'aucun fichier n'est encore ouvert. Ce nom

❶ sera défini par l'utilisateur lors de la toute première sauvegarde (`on_actionEnregistrer_triggered`) ou lors de l'ouverture d'un fichier (`on_actionOuvrir_triggered`) ; il est utilisé pour les sauvegardes et est affiché dans la barre de titre (`modificationAEnregistrer`).

❷ Au démarrage, on crée un modèle ModeleTableBiblio avec une liste vide : on a une nouvelle bibliothèque vide en mémoire, prête à être remplie. Dans une version plus élaborée de l'application, on ferait typiquement cette opération quand on active le menu Fichier > Nouveau (cette fonction a été omise pour ne pas alourdir l'exemple).

❸ L'appel `self.modificationAEnregistrer(False)` signale qu'il n'y a encore rien à sauvegarder sur fichier, sachant que le modèle vient d'être créé. La méthode `modificationAEnregistrer` configure visuellement la fenêtre pour bien indiquer cet état (voir ❺).

❹ Les boutons fléchés qui héritent de QAbstractSpinBox tels que QDateEdit et QDoubleSpinBox affichent leurs données selon un format bien défini ; ils ne peuvent pas a priori afficher un texte particulier ou être vide. La méthode `setSpecialValueText` permet de déclarer une chaîne de caractères spéciale, qui sera affichée à la place de la valeur minimale admise sur le widget. Ainsi, le champ `"Prix"` affichera `" "` (un espace blanc) quand la valeur interne sera la valeur minimale admise (—0.01). Ces valeurs minimales sont donc des valeurs *sentinelles* qui ont une signification particulière à gérer par l'application ; dans notre cas, ces valeurs indiquent que les champs n'ont pas encore été saisis (voir méthode `effaceLivre`).

***Note** > Notons que ces instructions pourraient être omises en utilisant Qt Designer pour assigner directement les propriétés ; il ne faut pas oublier dans ce cas d'invoquer la fonction Compiler la feuille d'eric6 (ou exécuter directement `pyuic5`).*

❺ La méthode `modificationAEnregistrer` gère l'attribut `fichierNonEnregistre`, qui est un booléen indiquant si on a déjà enregistré les dernières modifications dans un fichier ; on le met à vrai à chaque modification et à faux après chaque sauvegarde. De manière assez standard, la méthode en profite pour indiquer le nom du fichier ouvert (si existant), suivi d'un astérisque si les dernières modifications ne sont pas encore enregistrées. L'attribut `fichierNonEnregistre` sera utilisé par

ailleurs pour déterminer s'il y a lieu de mettre un message de confirmation avant la sortie de l'application (`closeEvent`) ou avant l'ouverture d'un fichier (`on_actionOuvrir_triggered`).

❻ On a ajouté ce test pour gérer l'absence de sélection dans la liste, ce qui arrive lorsque la liste est vide, lors de l'ouverture d'un fichier ou si l'utilisateur fait Ctrl+clic sur la ligne sélectionnée ; dans ce cas, on efface les champs de détail et on rend le bouton Supprimer inactivable.

❼ Comme expliqué plus haut, le fait de mettre la valeur minimum admise dans les widgets QAbstractSpinBox a pour effet d'afficher la valeur spéciale assignée (voir constructeur `__init__`) et ainsi d'effacer la valeur.

❽ Lors de la tentative de fermeture, si l'attribut `fichierNonEnregistre` est Faux, cela signifie que le modèle est déjà enregistré sur fichier : on accepte l'événement et l'application se termine sans autre forme de procès ; dans le cas contraire, on demande confirmation à l'utilisateur.

❾ Lors de la demande d'ouverture d'un fichier (menu Fichier > Ouvrir…), si l'attribut `fichierNonEnregistre` est Vrai, on demande confirmation à l'utilisateur car cette action va faire perdre les données non enregistrées. Si l'utilisateur se ravise, on sort de la méthode par `return` ; sinon, on continue.

❿ Une fois le fichier choisi par l'utilisateur, on lit ce fichier et on crée un nouveau modèle via notre méthode statique `ModeleTableBiblio.creeDepuisFichier`. Comme il s'agit d'un nouveau modèle, il faut le déclarer à la vue et refaire la connexion du signal `selectionChanged`. On mémorise le nom du fichier dans `nomFichierBiblio` et on appelle `modificationAEnregistrer(False)` pour afficher ce nom dans la barre de titre.

***Note** > Comme on est en Python, on n'a pas à se soucier du modèle existant avant l'ouverture du fichier : la zone mémoire utilisée sera récupérée automatiquement.*

⓫ Lors de la demande d'enregistrement (menu Fichier > Enregistrer), si on n'a pas encore de nom de fichier, on demande à l'utilisateur d'en fournir un en utilisant la

boîte de dialogue ad-hoc (méthode statique `QFileDialog.getSaveFileName`).

⓬ Le présent test sert à être sûr qu'on a bien un nom de fichier défini ; il faut en effet se prémunir du cas où l'utilisateur a fermé la boîte de dialogue sans choisir de fichier (voir point ⓫).

⓭ À l'activation du bouton Nouveau, on commence par déselectionner la ligne éventuellement sélectionnée dans la liste. Si une ligne était effectivement sélectionnée, le signal `selectionChanged` est émis et la méthode `on_treeViewLivres_selectionChanged` est appelée : dès lors, les champs de saisie sont effacés et on n'a rien d'autre à faire. Oui, seulement… comme rien ne garantit qu'une ligne était sélectionnée à l'activation du bouton, il est nécessaire d'appeler `effaceLivre` explicitement.

⓮ À l'activation du bouton Sauvegarder, on commence par créer une nouvelle instance de livre, en récupérant les données saisies dans les différents champs…

⓯ … ensuite, on teste si aucune ligne n'est sélectionnée dans la liste. Si oui, on sait qu'on a un nouveau livre à ajouter ; on invoque la méthode `ajouteLivre` sur le modèle en lui passant le nouveau livre ; ceci aura pour effet de notifier la vue et de faire apparaître le livre dans la liste (mais il ne faut plus se soucier de cela !). Pour aider l'utilisateur, on appelle `on_pushButtonNouveau_clicked()`, pour permettre des saisies à la chaîne (c'est juste un choix de conception d'interface).

⓰ Au contraire, si une ligne est sélectionnée dans la liste, on sait que l'utilisateur est en train de mettre à jour les données d'un livre existant ; on récupère l'indice du livre sélectionné et on invoque `remplaceLivre` sur le modèle, avec cet indice et le nouveau livre. Ceci aura pour effet de notifier la vue et rafraîchir le livre sélectionné dans la liste (si, bien sûr, l'utilisateur a modifié le titre, l'auteur ou l'éditeur de ce livre).

⓱ À l'activation du bouton Supprimer, on vérifie qu'on a bien un livre sélectionné ; normalement, c'est le cas, car nous avons pris soin de désactiver le bouton si aucune ligne n'est sélectionnée (voir `on_treeViewLivres_selectionChanged`) — mais

on n'est jamais trop prudent ! On récupère l'indice du livre sélectionné et on invoque `supprimeLivre` sur le modèle avec cet indice ; ceci aura pour effet de notifier la vue et supprimer le livre sélectionné de la liste.

À l'exécution, on peut voir enfin les différentes fonctions à l'œuvre : ajout, modification, suppression de livres, enregistrement et ouverture de fichier et les différents messages de confirmation.

5. Touches finales

On a à présent une application qui permet de faire toutes les opérations attendues, y compris la persistance sur fichier. Il y a cependant encore des points d'ergonomie à améliorer. Le point le plus critique est sans doute qu'il n'y a pas de contrôle sur la sauvegarde des données saisies[8] : l'utilisateur peut facilement oublier de cliquer sur le bouton Sauvegarder et perdre les dernières saisies. De plus, ce bouton est toujours activable, même si rien n'a été modifié dans les champs de saisie. Enfin, rien n'interdit de créer des livres sans titre. Les problèmes d'ergonomie de ce type ne doivent pas être sous-estimés ; ils peuvent faire la différence entre le prototype sympa et l'application *de production* réellement utilisable. Nous allons terminer ce chapitre en expliquant comment régler les quelques points signalés (il restera sans aucun doute encore bien des choses à améliorer). Au passage, nous pourrons voir la puissance des connexions signaux-slots à l'œuvre.

Dans ce qui suit, plutôt que d'introduire un nouvel attribut booléen indiquant si une saisie est en cours au niveau des champs de détail, on va utiliser un petit artifice : le bouton Sauvegarder va se charger lui-même de maintenir cette information. En testant s'il est activable (via la méthode `isEnabled`), on saura si une saisie est en cours. Ce dont il faut juste s'assurer, c'est de bien gérer l'état de ce bouton suivant les différentes actions de l'utilisateur.

***Note** > Le fait de pouvoir interroger l'état des widgets est une faculté souvent sous-utilisée par les programmeurs ; dans le cas présent, on l'exploite pour éviter d'avoir à créer un attribut redondant qui risque, si on n'est pas rigoureux à 100 %, d'être désynchronisé par rapport à l'état du bouton. Notons que cette technique pourrait être utilisée par ailleurs pour supprimer l'attribut booléen `fichierNonEnregistre`, sachant qu'il est toujours corrélé à l'état de `actionEnregistrer`.*

Sur la base de cette idée, voyons les choses à modifier par rapport à l'implémentation précédente (les points de suspension indiquent que les lignes qui suivent sont à *ajouter* par rapport au listing donné précédent).

```
 -*- coding: utf-8 -*-

# main_windows_biblio.py

from PyQt5.QtCore import pyqtSlot, QDate, QItemSelectionModel
…
```

```
class MainWindowBiblio(QMainWindow, Ui_MainWindowBiblio):

    def __init__(self, parent=None):
        …
        self.pushButtonSauvegarder.setEnabled(False)
```

❶ for lineEdit in (self.lineEditTitre, ❷ self.lineEditAuteur, self.lineEditEditeur):
lineEdit.textEdited.connect(self.declareSaisieEnCours)
self.comboBoxGenre.currentIndexChanged.connect(self.declareSaisieEnCours)
self.dateEditAnnee.dateChanged.connect(self.declareSaisieEnCours)
self.plainTextEditResume.textChanged.connect(self.declareSaisieEnCours)
self.doubleSpinBoxPrix.valueChanged.connect(self.declareSaisieEnCours) def
declareSaisieEnCours(self): ❸ self.pushButtonNouveau.setEnabled(False)
saisieValide = len(self.lineEditTitre.text().strip()) > 0
self.pushButtonSauvegarder.setEnabled(saisieValide) def
modificationAEnregistrer(self,fichierNonEnregistre): …
self.pushButtonNouveau.setEnabled(True) ❹
self.pushButtonSauvegarder.setEnabled(False) def
on_treeViewLivres_selectionChanged(self,selected,deselected): indexesSelection =
selected.indexes() if self.pushButtonSauvegarder.isEnabled(): ❺ reponse =
QMessageBox.question(self,'Confirmation', 'Abandonner la saisie en cours ?',
QMessageBox.Yes,QMessageBox.No) if reponse == QMessageBox.No:
selectionModel = self.treeViewLivres.selectionModel()
selectionModel.selectionChanged.disconnect(❻
self.on_treeViewLivres_selectionChanged)
selectionModel.select(selected,QItemSelectionModel.Deselect)
selectionModel.select(deselected,QItemSelectionModel.Select)
selectionModel.selectionChanged.connect(self.on_treeViewLivres_selectionChanged)
return if len(indexesSelection) == 0: ❼ self.effaceLivre()
self.pushButtonSupprimer.setEnabled(False) else: indexSelection =
indexesSelection[0] indiceLivreSelectionne = indexSelection.row()
self.afficheLivre(self.modeleTableBiblio.livres[indiceLivreSelectionne])
self.pushButtonSupprimer.setEnabled(True) self.pushButtonNouveau.setEnabled(True)
self.pushButtonSauvegarder.setEnabled(False) @pyqtSlot() def
on_pushButtonNouveau_clicked(self): …
self.pushButtonSauvegarder.setEnabled(False) ❽

❶ À l'initialisation, on commence par rendre le bouton Sauvegarder inactivable (comme il n'y a pas encore eu de saisie).

Comme on veut détecter tout changement sur n'importe lequel des champs de saisie,

❷ on va connecter tous les signaux de changement (propres à chacun des widgets) à la *même* méthode slot `declareSaisieEnCours`. On peut ainsi facilement intercepter n'importe quelle action atomique de saisie — frappe d'une touche clavier, sélection dans la liste déroulante, etc. — et réagir de la même manière (voir point ❸).

❸ La méthode `declareSaisieEnCours` désactive le bouton Nouveau (pour ne pas risquer de perdre le contenu des champs saisis) ; ensuite on teste si la saisie est valide en regardant simplement si le champ *"Titre"* est vide (par précaution, on appelle la méthode Python `strip()`, qui élimine les espaces éventuels et ainsi détecte les titres *blancs*) ; enfin, on rend le bouton Sauvegarder activable selon que cette la saisie est valide ou non.

❹ La méthode `modificationAEnregistrer` est appelée chaque fois que les champs saisis sont rapatriés dans le modèle ; c'est donc le bon moment pour rendre le bouton Nouveau activable et le bouton Sauvegarder non-activable.

❺ Lorsque la sélection change, on teste si une saisie est en cours en regardant l'état du bouton Sauvegarder ; le cas échéant, on le signale à l'utilisateur et on demande sa confirmation.

❻ Si l'utilisateur se ravise, il faut faire en sorte de revenir à la ligne précédemment sélectionnée (ou pas de sélection, dans le cas d'un nouveau livre). À cette fin, on utilise les arguments `selected` et `deselected` pour défaire l'action de l'utilisateur : on désélectionne la ligne sélectionnée et on resélectionne la ligne sélectionnée, si elle existe. Très bien… mais une petite sophistication supplémentaire est encore nécessaire : si on se contente simplement de changer la sélection comme expliqué, le signal `selectionChanged` va être émis et la méthode `on_treeViewLivres_selectionChanged` va être à nouveau invoquée ! Pour éviter cet appel récursif non désiré, on déconnecte temporairement le signal du slot juste avant l'opération et on le reconnecte juste après. C'est un peu comme un chirurgien qui endort son patient avant l'intervention : il évite ainsi des cris qui pourraient compromettre la réussite de l'opération !

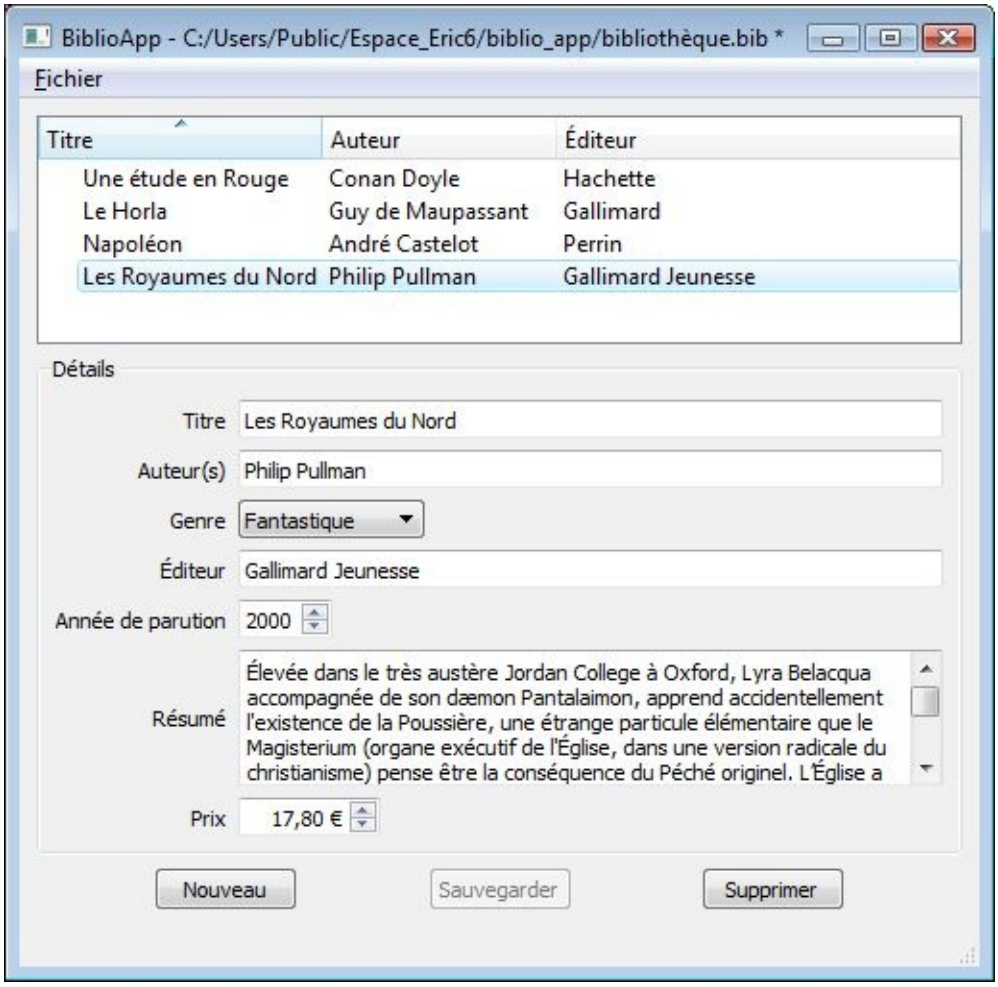
❼ À ce niveau, la nouvelle sélection est acceptée (soit qu'il n'y ait pas de saisie en cours, soit que l'utilisateur ait accepté de la perdre). On fait les mêmes traitements

qu'avant et on termine en désactivant le bouton Sauvegarder pour rendre compte de cet état.

❽ À l'activation du bouton Nouveau, on rend le bouton Sauvegarder inactivable, en attendant la première saisie.

Après ces dernières modifications, nous avons une application fonctionnelle et tolérante aux erreurs de manipulation les plus communes.

Figure 7.2 : *BiblioApp v1.00*



Nous avons couvert dans ces premiers chapitres, les principes de programmation MVD en PyQt, l'utilisation des widgets et les connexions signaux-slots. Bien sûr, la bibliothèque Qt Widgets est très vaste et nous n'avons abordé qu'une infime partie des possibilités offertes. Néanmoins, les concepts principaux de PyQt ont été présentés.

Vous devriez à ce stade avoir les connaissances nécessaires pour aller plus loin, en vous aidant notamment de la documentation excellente fournie par Qt/PyQt et des applications de démonstration. Vous êtes invité à expérimenter par vous-même, à modifier l'implémentation qui a été présentée ici. Python, à l'inverse de C++ et Java, est un langage interprété, concis, simple mais puissant, qui se prête particulièrement bien à l'apprentissage par expérimentation.

---

[8] Rappel : à ne pas confondre avec l'*enregistrement* des données sur fichier !

Internationaliser son application

Niveau : débutant à intermédiaire

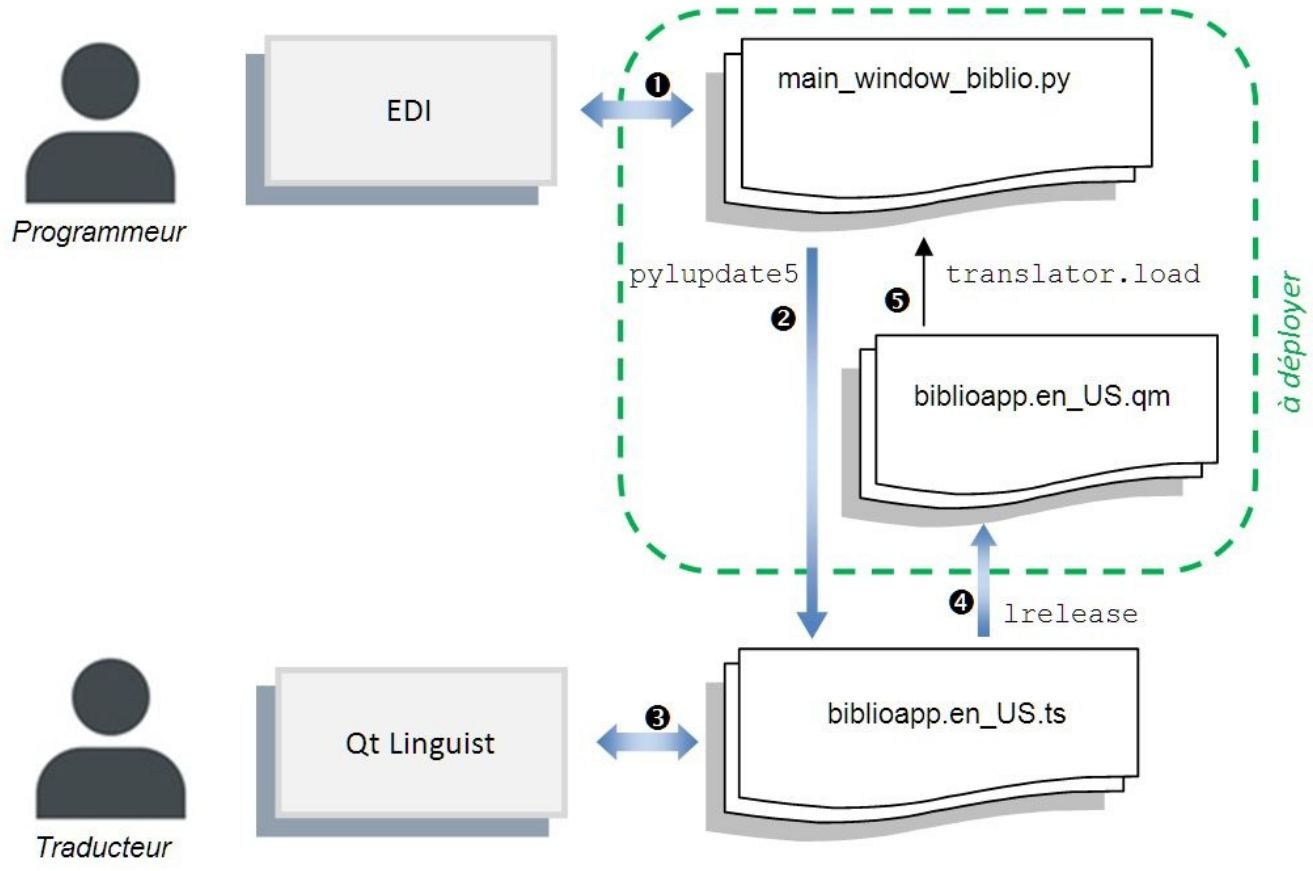
Objectifs : utiliser les outils PyQt

Prérequis : Créer une première application

L'internationalisation d'une application est une activité qui, idéalement, doit être découplée du développement. Bien qu'elle soit plutôt réalisée à la fin de celui-ci, le cycle de développement nécessitera forcément plusieurs itérations où des libellés, titres, messages vont apparaître, être modifiés ou même disparaître. Il est important que le travail de traduction puisse être bien isolé, indépendant des programmes, pour pouvoir éventuellement être effectué par un traducteur ou une équipe de traducteurs qui ne connaissent a priori rien de PyQt ou de la programmation.

Qt et PyQt proposent un système d'internationalisation bien pensé, qui répond à tous ces besoins. Ce système repose sur des formats de fichiers (`.ts` et `.qm`), des commandes (`pylupdate5`, `lrelease`) et une application pour aider les traducteurs (Qt Linguist). La Figure 8.1 en présente le principe. Nous y avons distingué les deux rôles, programmeur et traducteur.

Figure 8.1 : *Chaîne de compilation PyQt pour l'internationalisation*



Voici les grandes étapes illustrées sur cette figure.

❶ Le programmeur édite ses programmes sources Python, en appelant la méthode `tr` ou `translate` pour chaque chaîne de caractères à traduire. Ceci est fait automatiquement pour le code généré à partir des fichiers `.ui` (voir chapitre Développer avec Qt Designer).

❷ L'utilitaire `pylupdate5` analyse le code source Python pour retrouver les chaînes de caractères à traduire. Ces chaînes sont placées dans des fichiers ayant l'extension `.ts` (au format XML). Il y a un fichier distinct pour chaque langue à traduire.

❸ L'application Qt Linguist permet au traducteur d'ouvrir un fichier `.ts`, de voir les chaînes de caractères restant à traduire, de saisir les traductions et de sauvegarder son travail dans le même fichier `.ts`.

Pour tester ou déployer l'application, on invoque l'utilitaire `lrelease` qui compile le fichier `.ts` en un fichier `.qm` qui est directement exploitable par l'application.

❹ ***Note*** *> Vous l'aurez peut-être remarqué : les noms des deux derniers outils cités, Qt Linguist et lrelease, ne contiennent pas de Py. Ce n'est pas un oubli : ils font tous deux partie de l'environnement Qt natif, indépendant donc de PyQt. Vous pouvez donc, pour ces deux outils, utiliser toute référence documentaire de Qt, voire votre expertise acquise sur un projet Qt/C++.*

❺ À l'exécution, l'application lit le fichier `.qm` (le plus souvent au démarrage), selon la langue configurée. Pour ce faire, le programmeur instancie un objet QTranslator et invoque la méthode `load` en passant le nom du fichier à charger, par exemple `biblioapp_en.qm` pour avoir une version anglaise.

Nous avons parlé d'un découplage fort entre les tâches de programmation et celles de traduction. Concrètement, cela veut dire que la chaîne de traitement présentée peut être itérée :

- la traduction du fichier `.ts` n'a pas besoin d'être terminée à 100 % pour pouvoir exécuter l'application ; les chaînes de caractères non traduites seront simplement affichées dans leur langue d'origine. Cette caractéristique est d'ailleurs utile pour les libellés qu'on veut garder inchangés dans toutes les langues (par exemple, le nom "BiblioApp" placé dans la barre titre) ;

- l'invocation de l'utilitaire `pylupdate5` ne crée les fichiers `.ts` que la toute première fois ; les fois suivantes, les fichiers sont mis à jour de manière intelligente, en n'ajoutant que les nouveaux éléments à traduire sans toucher à ceux déjà traduits.

Nous allons concrétiser tout cela sur l'exemple de l'application BiblioApp pour la rendre accessible à des utilisateurs francophones et anglophones. Pour cela, nous allons décliner la chaîne présentée plus haut en six étapes :

1. modification du programme principal ;

2. marquage des éléments textuels à traduire dans les sources ;

3. création d'un fichier `.pro` ;

4. génération des fichiers `.ts` avec `pylupdate5` ;

5. traduction avec Qt Linguist ;

6. génération des fichiers `.qm` avec `lrelease`.

Note *> Le code source final de ce chapitre et les fichiers d'éléments textuels associés sont disponibles dans le projet* `internationalisation`  *.*

1. Modification du programme principal

Pour que l'application soit internationalisable, le programme principal requiert quelques adaptations. Il faut, dans l'ordre, créer un objet traducteur (instance de QTranslator), y charger un fichier d'éléments textuels `.qm` pour la langue cible et installer ce traducteur dans l'objet application. Nous donnons ci-après une méthode pour faire cela, mais il est possible de le faire de nombreuses autres façons.

Le programme proposé ici aura deux modes de fonctionnement, selon qu'on le lance avec un argument ou non :

- sans argument : la langue à utiliser est déterminée selon l'environnement configuré au niveau de l'OS ;
- avec argument : la langue à utiliser est déterminée par l'argument (`fr_FR` ou `en_US` dans notre cas).

L'option *sans argument* est la plus importante ; elle est utilisée pour l'application déployée en production. L'option *avec argument* est juste une facilité qu'on se donne pour tester les différentes traductions. Ainsi, à la fin du présent chapitre, on pourra lancer l'application selon l'une des commandes suivantes :

```
python start_app.py
python start_app.py fr_FR
python start_app.py en_US
```

Voici le listing du programme principal de `start_app.py` adapté pour réaliser cela.

```
# start_app.py
# version internationale

import sys
from PyQt5.QtWidgets import QApplication
from PyQt5.QtCore import QTranslator, QLocale
from main_window_biblio import MainWindowBiblio

app = QApplication(sys.argv)

translator = QTranslator()
```

❶ if len(sys.argv) == 1: ❷ locale = QLocale() ❸ translator.load(locale,"biblioapp",".") ❹ else:

translator.load("biblioapp."+sys.argv[1]) ❺ app.installTranslator(translator) ❻ mainWindowBiblio = MainWindowBiblio() mainWindowBiblio.show() rc = app.exec_() sys.exit(rc)

Note *> Comme auparavant, ce programme principal PyQt peut être pris comme une recette de cuisine. Il n'est pas obligatoire de le comprendre en détail pour la suite ; le lecteur pressé peut résolument passer à la section suivante !*

Voici des explications sur les lignes qui ont été ajoutées.

❶ On crée un traducteur, instance de QTranslator. Simple !

❷ On teste le nombre d'arguments : s'il n'y aucun argument passé `sys.argv` ne contient que le nom du programme : `['start_app.py']`, une liste de longueur 1.

❸ Dans le mode *sans argument*, on crée une instance de QLocale sans argument : ceci récupère les données configurées au niveau de l'OS, notamment la langue et le pays.

❹ On charge dans le traducteur le fichier d'éléments textuels `.qm` suivant la langue et le pays récupéré dans `locale`. Par exemple, sur un ordinateur configuré en English/UnitedStates, le fichier chargé sera `biblioapp.en_US.qm`.

Note *> Si aucun fichier n'est trouvé pour la langue et le pays configuré, il n'y a pas d'erreur : l'application s'affichera avec les éléments textuels du code source non traduits (voir l'encadré Faut - il traduire toutes les chaînes ?).*

❺ Dans le mode *avec argument*, on appelle une autre mouture de la méthode `translator.load`[9] qui charge un fichier nommé explicitement. On obtient ce nom de fichier par concaténation avec l'argument passé (le suffixe `.qm` n'est pas nécessaire). Comme signalé au point ❹, si l'argument ne correspond à aucun des fichiers `.qm` connus, aucune erreur n'est reportée[10].

❻ On termine en "installant" le traducteur dans l'application. Ceci veut dire que son rôle est à présent officialisé : tout objet PyQt qui aura besoin de traductions, depuis les menus jusqu'aux boutons, s'adressera à cet objet traducteur, instance de

QTranslator.

Ce court programme fonctionne parfaitement pour tous les éléments textuels définis dans l'application. Il y a malheureusement une petite lacune : les boutons figurant sur les boîtes de dialogue standards ne sont pas traduits : on aura par exemple des boutons Cancel dans les boîtes de message (au lien de Annule, par exemple). Nous verrons à la fin du chapitre comment remédier à cela.

Pour aller plus loin…

Signalons que les méthodes `QTranslator.load`, dans leurs deux formes, ont de nombreux paramètres optionnels qui peuvent aider à spécifier finement les choses. On peut notamment indiquer un répertoire particulier pour les fichiers d'éléments textuels, si on ne veut pas les mélanger avec les fichiers sources Python.

Signalons enfin que, en alternative avec ce qu'on a montré ici, on pourrait créer des instances de QLocale avec les références explicites aux langues et pays. Voici comment faire :

```
locale = QLocale(QLocale.French,QLocale.France)
# ou
locale = QLocale(QLocale.English,QLocale.UnitedStates)
```

***Note** > Référez-vous à la documentation des classes QTranslator et QLocale pour plus de détails.*

---

[9] Pour être précis, la méthode QTranslator.load est *surchargée*, ce qui signifie qu'il existe plusieurs implémentations distinctes selon la signature, c'est-à-dire le nombre et les types d'arguments passés (voir la documentation de QTranslator). Ceci est à la base une possibilité offerte par C++ (le langage natif de Qt) qui fait une résolution statique : le compilateur est capable de reconnaître les différentes signatures et se brancher sur l'implémentation adéquate. En Python, la surcharge statique n'est pas possible, car les types des arguments ne sont connus qu'à l'exécution du programme. Les concepteurs de PyQt ont heureusement prévu un mécanisme qui, à l'exécution, va intercepter les appels, reconnaître les types des arguments passés et appeler la méthode associée. PyQt transforme donc la surcharge statique (C++) en surcharge dynamique (Python).

[10] Au besoin, on pourrait changer le programme pour signaler une erreur.

2. Marquage des éléments textuels à traduire dans les sources

Dans nos fichiers sources, certaines chaînes de caractères sont à traduire et d'autres pas : par exemple `"Titre"` est à traduire, tandis que `", "` (par exemple, un séparateur sur lequel on applique la méthode `join`) ne l'est pas. En définitive, seul le programmeur sait faire la différence.

Avant de lancer la chaîne de traduction présentée plus haut, le programmeur doit appliquer deux règles dans ses fichiers sources pour que les différents outils puissent repérer et traiter les chaînes de caractères à traduire. Il doit ainsi :

- placer toutes les chaînes à traduire dans un appel à la méthode `self.tr(…)` ;
- s'assurer que toutes les chaînes ne contiennent que des caractères ASCII (notamment, pas de caractères accentués).

Par exemple, voici ce que cela donne sur la méthode `MainBiblioApp.closeEvent` :

```
    def closeEvent(self,event):
        if not self.fichierNonEnregistre:
            event.accept()
        else:
            messageConfirmation = self.tr("Etes-vous sur de vouloir
quitter" \
                                  " BiblioApp sans enregistrer le
fichier ?")
            reponse =
QMessageBox.question(self,self.tr("Confirmation"),
                                             messageConfirmation,

QMessageBox.Yes,QMessageBox.No)
            if reponse == QMessageBox.Yes:
                event.accept()
            else:
                event.ignore()
```

L'appel à `self.tr(…)` a deux buts : d'une part, il sert à marquer les chaînes à traduire pour que `pylupdate5` puisse les extraire ; d'autre part, il permet, à l'exécution, de substituer la chaîne par sa traduction dans la langue choisie. Ces points vont être explicités dans la suite (voir respectivement Section 4, Génération des fichiers .ts :

pylupdate5 et Section 7, Exécution de l'application).

Quant à l'utilisation de caractères ASCII, ce n'est pas une obligation stricte. Cependant, quand on travaille en PyQt, il est prudent d'édicter en règle cette recommandation faite dans la documentation de Qt. En effet, les chaînes de caractères destinées à l'affichage qui se trouvent dans les fichiers sources deviennent par la traduction des clés vers leur équivalent dans les langues cibles. Or, la gestion des caractères Unicode entre les mondes C++, PyQt et Python est compliquée à maîtriser — sans même parler de la portabilité entre les systèmes ! Il est donc vivement recommandé d'éviter l'emploi de ces caractères dans le code source[11]. Ceci n'implique évidemment pas qu'on doive sacrifier les accents à l'affichage : il suffit de mettre le français avec accent en langue cible pour garantir un affichage sans défaut. Certes, cette pratique peu paraître ennuyeuse mais, comme indiqué dans l'encadré Faut - il traduire *toutes* les chaînes ? , l'activité de traduction franco-française consistera simplement à copier et ajouter les accents aux éléments textuels qui le nécessitent ; ceux qui s'écrivent sans accent ne requièrent aucune action : ils seront repris tels quels à l'écran. Par ailleurs, il peut être bénéfique d'avoir un contrôle centralisé et uniforme de TOUS les éléments textuels en dehors du code (dans les fichiers `.ts`, en l'occurrence), quelle que soit leur langue.

Pour appliquer ces règles au fichier `main_window_biblio.ui`, il faut changer à la main les quelques libellés accentués : "Détails" devient "Details", "Résumé" devient "Resume", "€" devient "euro", etc. Ensuite, on recompile la feuille. C'est tout ! Le générateur de code prend soin de faire le reste du travail : ainsi, si l'on ouvre le fichier `Ui_main_window_biblio.py` généré, on y verra une méthode `retranslateUi` qui assigne tous les libellés en appelant la fonction `_translate` (qui est un alias pour `QtCore.QCoreApplication.translate`). En gros, cette fonction va jouer le même rôle que `self.tr()` sans qu'on doive coder ces appels nous-même.

Traitement des éléments textuels avec arguments

Qu'en est-il des éléments textuels qui incluent des informations dynamiques, telle que *Il y a 1000 livres dans le fichier 'alexandrie.bib'.*, où 1000 et 'alexandrie.bib' sont récupérés à partir de variables du programme ? La technique naïve de concaténer les chaînes de caractères :

```
self.tr("Il y a ") + str(nbLivres) + self.tr(" livres dans le
fichier '") \
    + nomFichier + self.tr("'.")
```

fonctionne mais est lourde et très mal adaptée. Il en résulte trois éléments textuels

séparés à traduire — ce qui peut poser des problèmes grammaticaux insolubles aux traducteurs. La bonne approche consiste à utiliser les techniques natives de Python pour l'interpolation des chaînes de caractères (en anglais, *string formating*). Il faut juste veiller à ne donner en argument de `self.tr` que la chaîne de caractères statique, donc avant substitution des arguments. Voici quelques exemples possibles, qui donnent un résultat équivalent :

```
self.tr("Il y a %d livres dans le fichier '%s'.") %
(nbLivres,nomFichier)
# ou
self.tr("Il y a {:d} livres dans le fichier
'{}'.").format(nbLivres,nomFichier)
# ou
self.tr("Il y a {0} livres dans le fichier
'{1}'.").format(nbLivres,nomFichier)
# ou
…
```

Beaucoup d'autres variantes sont permises par Python (voir la documentation).

L'élément textuel source à traduire sera celui passé en argument de `self.tr`, sans aucune altération. Le traducteur humain devra simplement veiller à bien respecter les arguments formels de l'élément textuel. Par exemple, avec la troisième mouture vue plus haut, l'élément textuel source sera `"Il y a {0} livres dans le fichier '{1}'."` qu'on traduira par exemple en `"There are {0} books in the file '{1}'."`.

Attention *> La documentation Qt parle d'une méthode `arg(…)`, applicable sur des chaînes contenant des arguments `%1`, `%2`, etc. Ceci n'est PAS applicable en PyQt 5. En effet, cette méthode suppose l'utilisation de la classe C++ QString, qui n'est pas portée en PyQt 5 ,car elle fait largement double emploi avec la classe `str` native de Python. Ceci n'est vraiment pas un problème sachant que les options de formatage sont bien plus puissantes en Python (les exemples précédents ne forment qu'un minuscule échantillon).*

---

[11] Si l'on tente de laisser les chaînes unicode inchangées, on a notamment des ennuis avec l'utilitaire pylupdate5 : le caractère "euro" (€) provoque une erreur ; la première exécution génère un fichier .ts correct (avec par exemple <source>Résumé</source>) mais, curieusement, les exécutions suivantes corrompent les chaînes de caractères avec accents (<source>RÃ©sumÃ©</source>), etc.

3. Création de biblioapp.pro

Pour configurer la chaîne de traduction, on a besoin de spécifier les noms de fichiers sources Python qui contiennent du texte à traduire et les langues cibles. Ces informations sont placées dans un petit fichier ayant l'extension `.pro`.

Concrètement, ce fichier contient au moins deux paramètres :

- `SOURCES` : le nom des fichiers sources Python ;
- `TRANSLATIONS` : le nom des fichiers `.ts` ; il y en a un pour chaque langue cible.

Pour notre application, nous allons créer un fichier `biblioapp.pro` , à mettre dans le répertoire où se trouvent les fichiers sources. Son contenu est le suivant :

```
# biblioapp.pro
SOURCES += modele_biblio.py \
           main_window_biblio.py \
           Ui_main_window_biblio.py
TRANSLATIONS += biblioapp.fr_FR.ts \
                biblioapp.en_US.ts
```

Dans `SOURCES`, nous avons mis uniquement les fichiers Python concernés par la traduction, ceux donc qui contiennent des appels à une des méthodes `tr` ou `translate` ; le programme principal `start_app.py` ne s'y trouve donc pas (mais on pourrait l'ajouter sans souci).

Dans `TRANSLATIONS`, on utilise la convention `langue_pays` : `fr_FR` signifie français de France, `en_US` signifie anglais américain. On peut ainsi tenir compte des particularismes spécifiques à chaque pays : pour couvrir l'anglais parlé au Royaume-Uni par exemple, on ajouterait `biblioapp.en_EN.ts` (un exemple classique est le mot "color" utilisé par les américains alors que les anglais utilisent "colour").

4. Génération des fichiers .ts : pylupdate5

La génération des fichiers de traduction est très simple. Il suffit d'exécuter la commande

```
pylupdate5 biblioapp.pro
```

Ceci a pour effet de créer les fichiers configurés dans `biblioapp.pro`, à savoir `biblioapp.fr_FR.ts` et `biblioapp.en_US.ts`.

Si l'on utilise eric6, pour simplifier les opérations suivantes de la chaîne de traduction, il est utile de déclarer les fichiers `.ts` générés. Pour cela, activez l'option Ajouter des fichiers de traduction… dans le menu contextuel de l'onglet Traductions du Gestionnaire de projet. On fait ainsi apparaître les deux fichiers `.ts` dans l'onglet susnommé. Cette opération est faite une fois pour toutes.

Par curiosité, vous pouvez éditer ces fichiers dans un éditeur texte ou XML (dans eric6 : menu contextuel Ouvrir dans l'éditeur). On retrouve, dans un habillage XML, toutes les chaînes de caractères qui ont été marquées dans nos fichiers sources Python. On constate que les deux fichiers (français et anglais) sont rigoureusement identiques à ce stade ; c'est normal : l'outil `pylupdate5` n'a pas de connaissances linguistiques pour différencier les deux langues, pas plus que les autres outils mis à disposition dans PyQt.

5. Traduction avec Qt Linguist

On arrive enfin à la phase de traduction proprement dite. L'opération consiste à éditer chaque fichier `.ts` pour ajouter les traductions de chaque élément textuel extrait du code source.

Comme les fichiers `.ts` sont des fichiers XML, il est tout à fait possible de le faire avec un éditeur de texte ou XML. C'est toutefois laborieux et peu aisé, surtout pour un traducteur qui n'est pas forcément habitué à une syntaxe comme celle de XML.

Fort heureusement, Qt propose une application puissante et conviviale pour éditer les fichiers de traduction : Qt Linguist.

Astuce *> Certains outils de traduction reconnaissent le format `.ts` de Qt : citons Wordfast, Swordfish et Virtaal. Les traducteurs professionnels habitués à ces outils peuvent donc se passer de Qt Linguist. Si vous désirez faire des recherches vous-même, l'acronyme CAT (Computer Aided Translation) peut être utile à connaître.*

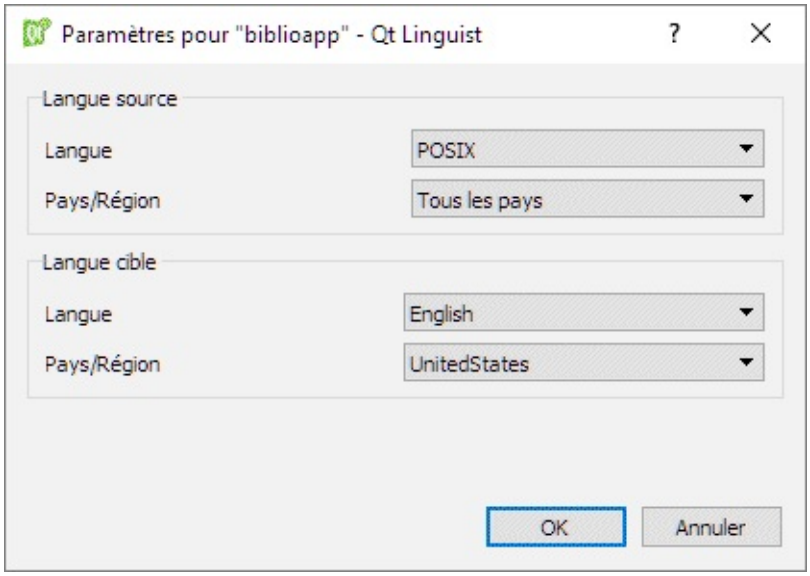
Pour lancer Qt Linguist depuis eric6, il suffit de double-cliquer sur un fichier `.ts` dans l'onglet Traductions du Gestionnaire de projet (voir plus haut). Sans eric6, on peut lancer cette application à partir du bureau, du menu Démarrer sous Windows ou en exécutant sur la console

```
linguist
```

suivie éventuellement du nom du fichier à ouvrir.

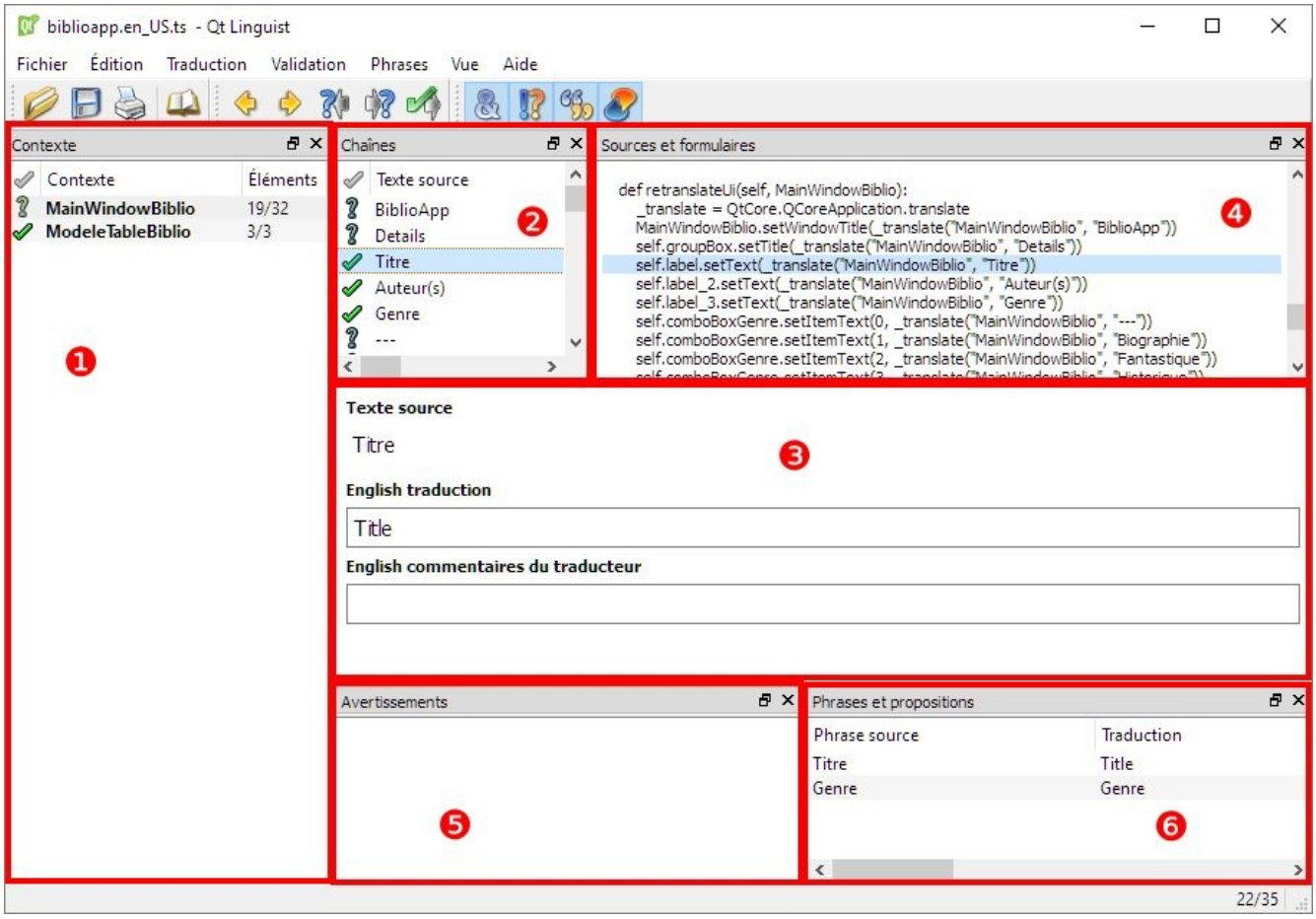
Lors de la première ouverture de `biblioapp.en_US.ts`, Qt Linguist présente une boîte de dialogue pour demander quelles sont les langues source et cible du fichier.

Figure 8.2 : *Qt Linguist — dialogue de paramètres*



Comme on a suivi la recommandation plus haut (éléments textuels en ASCII dans les sources), on peut laisser le cadre Langue source inchangé. Dans le cadre Langue cible, on va sélectionner la langue et le pays correspondant au suffixe de notre fichier de traduction : English/UnitedStates.

Figure 8.3 : *Qt Linguist*



L'interface de Qt Linguist est assez intuitive. On a différentes vues qu'on peut déplacer, faire apparaître ou disparaître (menu Vue > Vues). Voici un survol rapide de ces vues :

❶ Contexte : affiche les différentes classes Python d'où sont extraits les éléments textuels à traduire. Dans notre cas, on en a deux : MainWindowBiblio et ModeleTableBiblio. Ces classes servent à donner un contexte d'utilisation au traducteur : ainsi, un même mot pourrait être traduit de deux manières différentes selon le contexte.

***Note** > Chaque contexte indique un petit compteur de l'avancement de la traduction. 19/32 signifie qu'on a traduit 19 des 32 éléments à traduire dans le contexte.*

❷ Chaînes : liste dans la langue source tous les éléments textuels du contexte sélectionné. Une icône indique l'état de la traduction : un point d'interrogation signale que l'élément textuel n'est pas traduit, une coche verte indique qu'il est

traduit. Ces icônes de statut sont indicatives, elles peuvent être changées en cliquant dessus.

❸ Zone de saisie de la traduction de l'élément textuel sélectionné dans la vue Chaînes : c'est la zone centrale de l'application qu'on ne peut pas faire disparaître ; les vues viennent se placer autour.

❹ Sources et formulaires : affiche le code source Python en surlignant la ligne d'où l'élément textuel sélectionné est issu ; en cas de doute ou d'ambiguïté, ceci peut aider à comprendre ce qu'il signifie.

❺ Phrases et propositions : liste les traductions usuelles sauvegardées par l'utilisateur. Ces traductions sont sauvegardées dans un fichier "livre de phrases" avec l'extension `.qph`. Ces fichiers sont gérés au niveau de l'élément Phrases du menu principal. Il s'agit d'une fonction surtout utile pour les grosses applications. Nous ne la couvrirons pas dans cette introduction.

❻ Avertissements : liste les anomalies potentielles détectées dans les traductions ; il s'agit de différences par rapport au livre de phrases ou d'incohérences entre les éléments textuels source et cible : esperluette manquante (accélérateur), ponctuation différente en fin d'élément ou différence dans les arguments (`%1`, `%2`, etc.). Ces validations peuvent être désactivées ou réactivées à tout moment via le menu Validation ou l'un des boutons Basculer la vérification… sur la barre d'outils.

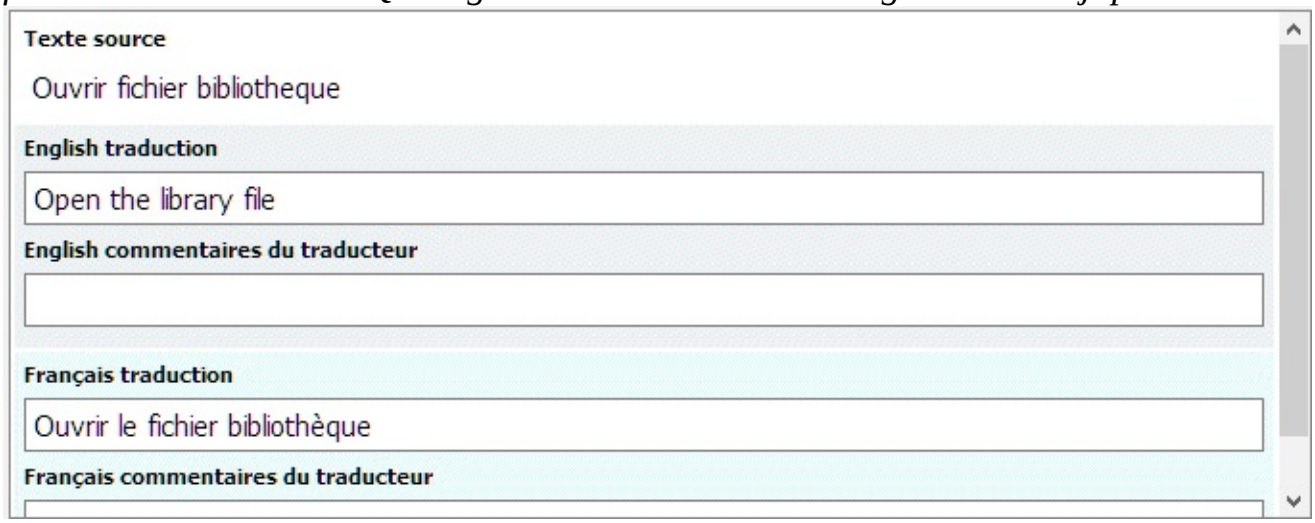
Vous pouvez procéder à la saisie des traductions élément par élément, dans le champ English traduction. Pour qu'un élément textuel traduit soit effectivement pris en compte dans la suite des traitements, il doit être placé dans l'état Validé. À cette fin, le plus simple est de faire Ctrl+Retour (équivalent du menu Traduction > Valider et suivant ou du bouton avec infobulle Marque l'élément comme fini et aller à l'élément suivant non terminé) ; l'icône d'état de l'élément textuel courant (un point d'interrogation) doit alors se changer en coche verte.

***Note** > Cette obligation d'opérer une validation est propice à une phase de relecture des éléments traduits, par exemple par un traducteur tiers ; on peut aussi s'en servir pour laisser non validées les traductions pour lesquelles on a un doute et qui requièrent une vérification. Notez par ailleurs que le menu Traduction propose*

d'autres actions et raccourcis utiles.

On peut à tout moment sauvegarder le travail fait dans le fichier `.ts` suivant les modalités habituelles (par exemple Ctrl+S).

Astuce *> Il est possible d'éditer plusieurs fichiers `.ts` en même temps dans Qt Linguist, en les ouvrant en séquence. Dans ce cas, la zone centrale contiendra plusieurs champs de saisie, pour chacune des langues ciblées par les fichiers ouverts. On peut voir ainsi simultanément les différentes traductions d'un même élément et les saisir à la volée. Cette faculté, qui a l'air anodine, est extrêmement pratique pour les traductions de langues proches (par exemple, anglais britannique et américain) ; elle peut aussi aider un traducteur qui ne maîtrise pas bien la langue source utilisée. Prenons un exemple. Je viens de traduire complètement mon application du français vers l'anglais (`biblioapp.en_US.ts`). J'ai une amie japonaise qui est prête à m'aider à faire la traduction en japonais ; seulement, elle connaît l'anglais mais pas le français (la langue source de BiblioApp). Il suffit alors que je lui fournisse deux fichiers `biblioapp.en_US.ts` (rempli) et `biblioapp.ja_JP.ts` (vierge) pour qu'elle puisse travailler : elle ouvre les deux fichiers ensemble dans Qt Linguist et elle traduit de l'anglais vers le japonais.*



Faut - il traduire *toutes* les chaînes ?

Il arrive fréquemment de ne pas traduire tous les éléments, que ce soit pour une langue cible donnée ou pour toutes les langues cibles. Ce n'est pas un problème bloquant en soi : à l'exécution, les éléments textuels non traduits seront simplement affichés dans la langue source. On peut ainsi avoir des phases de test ou déployer des versions alpha/beta avant que tout ne soit traduit jusqu'au dernier carat. En outre, le traducteur peut s'économiser du travail inutile, notamment pour les noms

propres ("BiblioApp") et lorsque la langue source est proche de la langue cible. Ainsi, dans le cas particulier de BiblioApp, la traduction franco-française peut se limiter à traiter les éléments textuels qui n'ont pas d'accents : par exemple, "Details" est à traduire en "Détails" tandis que "Titre" ne nécessite aucun traitement.

6. Génération des fichiers .qm : lrelease

Une application Qt ne sait pas traiter les fichiers de traduction `.ts` édités par Qt Linguist. Pour pouvoir faire apparaître les traductions dans l'application, il y a encore une petite formalité à remplir : générer des fichiers d'éléments textuels `.qm` (un par fichier `.ts`). Pour rappel, seuls les fichiers `.qm` sont nécessaires pour exécuter l'application ; ces fichiers sont donc ceux qu'on devra déployer.

Pour compiler les fichiers `.ts` avec eric6, sélectionnez-les dans l'onglet Traductions, par exemple `biblioapp.fr_FR.ts`, et activez Compiler la traduction dans le menu contextuel. Ceci génère un nouveau fichier `biblioapp.fr_FR.qm`. On fera la même chose pour `biblioapp.en_US.ts`.

Cette opération peut aussi être effectuée en exécutant la commande :

```
lrelease biblioapp.pro
```

qui compile tous les fichiers `.ts` déclarés en fichiers `.qm`.

Cette opération est à faire évidemment chaque fois qu'on veut prendre en compte une nouvelle version d'un fichier `.ts`.

7. Exécution de l'application

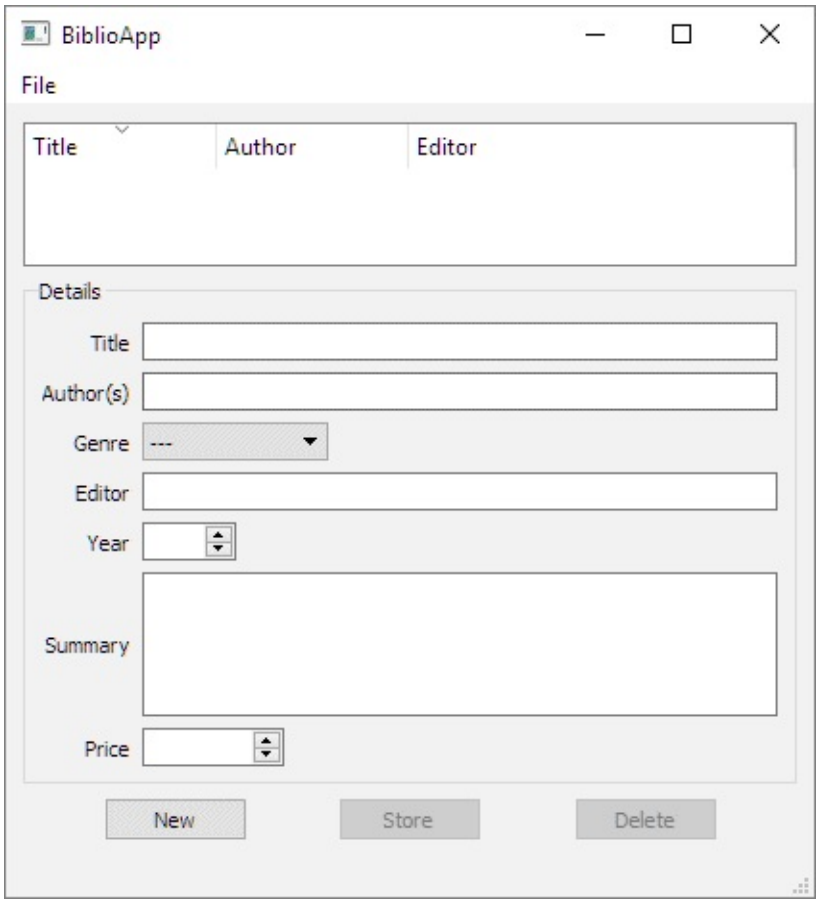
Une fois qu'on a fait toutes ces étapes, il est temps de voir le résultat.

Si on lance l'application de manière habituelle (sans argument), elle s'affichera dans la langue configurée sur le système pourvu qu'on ait choisi français/France ou anglais/États-Unis. Pour faire des tests, cela nécessite d'accéder à la configuration système et, parfois, de lancer une nouvelle session. Heureusement, pour faciliter les tests, nous avons prévu un mode spécial qui permet d'indiquer la langue en argument.

Pour voir l'application en anglais américain, on fait donc :

```
python start_app.py en_US
```

Figure 8.4 : *BiblioApp en anglais*



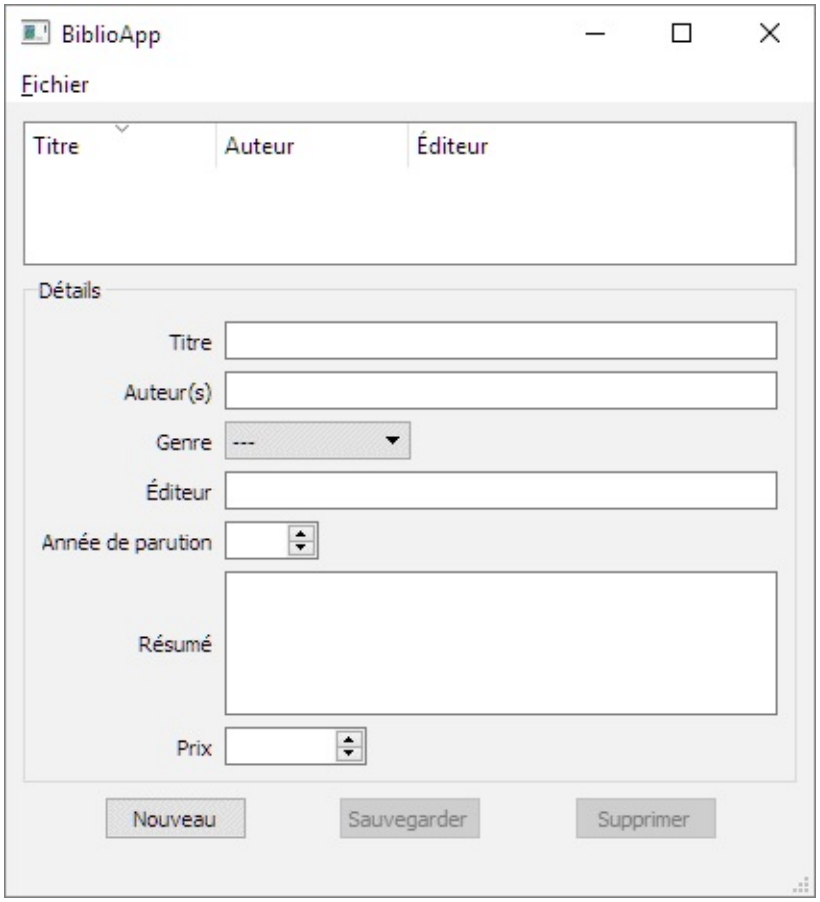
Si on a été suffisamment loin dans la traduction, on peut vérifier que les libellés,

menus, messages dans les boîtes de dialogue sont bien tous en anglais.

Et pour avoir l'application en français de France :

```
python start_app.py fr_FR
```

Figure 8.5 : *BiblioApp en français*



Si on a bien fait le travail de traduction franco-français (ASCII vers UTF-8), on peut vérifier qu'on a bien retrouvé tous nos accents.

On peut noter en comparant les deux affichages que, grâce au travail discret mais efficace des positionneurs, les tailles des libellés et champs se sont adaptés aux textes affichés : ainsi, la mise en page s'est automatiquement calée sur le libellé le plus long, à savoir "Author(s)" dans la version anglaise et "Année de parution" dans la version française..

Il reste malheureusement un petit souci que le testeur attentif aura remarqué : les boutons des boîtes de dialogue standards sont toujours en anglais, même si le français est choisi. Un remède à ceci est présenté dans Section 9, Traduction des boutons

standards.

8. Développement itératif

Les étapes qui viennent d'être expliquées sont, on s'en doute, rarement faites une seule fois : les développeurs ajoutent/modifient de nouvelles fonctions, les traducteurs complètent/affinent leurs traductions, etc. Si on a bien compris la chaîne de traduction PyQt, on sait quel outil doit être utilisé à quel moment. Voici quelques cas de figure.

- Lorsqu'une nouvelle fonction/fenêtre est développée, il faut invoquer `pylupdate5` pour ajouter des nouveaux éléments textuels aux fichiers `.ts` (pour rappel, cet outil conserve les traductions déjà faites). Si le nouveau développement est fait dans de nouveaux fichiers sources, il faut auparavant les ajouter au fichier projet `.pro`. Ensuite, on lance Qt Linguist pour traduire les nouveaux éléments textuels dans *toutes* les langues.

- Pour ajouter une nouvelle langue, on modifie le fichier projet `.pro` et on suit la chaîne complète, à partir de `pylupdate5`.

- Lorsqu'on veut déployer ou tester les derniers fichiers `.ts` mis à jour, on appelle `lrelease`.

Les modalités précises dépendront bien entendu du contexte de développement. Pour une grosse application impliquant une équipe de développeurs, une équipe de traducteurs et un responsable de distribution, il faudra sans doute mettre des procédures en place et s'aider d'un outil de gestion de configuration.

9. Traduction des boutons standards

Comme signalé, si on se limite à la procédure qui a été présentée, les boutons des boîtes de dialogue standards seront toujours en anglais quelle que soit la langue configurée : par exemple, la boîte de dialogue affichée à l'appel de `QMessageBox.question` affiche invariablement les boutons Yes et No. L'explication tient bien sûr dans le fait qu'on a ici des éléments textuels internes à la librairie PyQt et donc non traduits dans nos fichiers de traduction.

Pour pallier ce désagrément, il existe une solution : charger les fichiers de traduction propres à Qt, qui sont heureusement fournis dans l'installation. Nous allons présenter ici comment procéder.

La première chose à faire est de copier les fichiers d'éléments textuels standards Qt dans le même répertoire que nos `biblioapp.*.qm`. Ces fichiers se trouvent dans le répertoire d'installation de Qt, dossier `translations`. Dans ce répertoire, on trouve tous les fichiers de traduction, qu'ils soient destinés aux applications Qt écrites elles-mêmes en Qt (`assistant_*.qm`, `designer_*.qm`, `linguist_*.qm`, `qmlviewer_*.qm`) ou aux modules Qt utilisés dans nos programmes (`qt*_*.qm`). Pour l'application BiblioApp, on a juste besoin de prendre dans ce répertoire les fichiers `qt_*.qm` (qui sont en fait des catalogues incluant des références vers les autres) et les fichiers `qtbase_*.qm`, qui correspondent aux composants widgets de base.

Concrètement, pour avoir les versions françaises et anglaises, vous allez donc copier quatre fichiers, à savoir `qt_fr.qm`, `qt_en.qm`, `qtbase_fr.qm` et `qtbase_en.qm`, dans le répertoire de BiblioApp contenant nos fichiers `.qm`.

***Note** > Si votre application utilise des modules spécifiques de Qt, vous irez chercher les fichiers correspondants : `qtconnectivity_*.qm`, `qtmultimedia_*.qm`, etc.*

Ensuite, il faut adapter le programme principal afin de créer et installer les trois traducteurs, instances de QTranslator, pour les trois fichiers `.qm` correspondant à la langue sélectionnée : "biblioapp.", "qt_", "qtbase_". Le premier traducteur gère les éléments textuels propres à l'application BiblioApp, les deux autres gèrent les éléments textuels standards de Qt.

```
# start_app.py
# version internationale, incluant les boutons standards

import sys
```

```
from PyQt5.QtWidgets import QApplication
from PyQt5.QtCore import QTranslator, QLocale
from main_window_biblio import MainWindowBiblio

app = QApplication(sys.argv)

enLangueNative = len(sys.argv) == 1
if enLangueNative:
    locale = QLocale()
else:
    languePays = sys.argv[1]
translators = []
for prefixeQm in ("biblioapp.", "qt_", "qtbase_"):
    translator = QTranslator()
    translators.append(translator)
    if enLangueNative:
        translator.load(locale,prefixeQm)
    else:
        translator.load(prefixeQm+languePays)
    app.installTranslator(translator)

mainWindowBiblio = MainWindowBiblio()
mainWindowBiblio.show()
rc = app.exec_()
sys.exit(rc)
```

La technique utilisée est la même que celle présentée dans la première version. On a essentiellement factorisé les trois lectures dans une boucle.

Note *> Il peut sembler surprenant de stocker les instances de QTranslator dans une liste (`translators`) ... qui n'est pas utilisée ! Voici l'explication : ceci est une astuce pour contourner un problème subtil propre au portage de Qt en Python : à chaque itération, l'assignation à la variable `translator` libère en mémoire l'instance précédemment créée. Pour éviter cela, il est vital de garder dans le script Python au moins une référence vers chaque instance de QTranslator ; la sauvegarde dans une liste est un moyen simple de garantir cela.*

Après exécution, on peut voir que les boutons ont à présent les noms correspondant à la langue configurée.

Accès à une base de données

Niveau : intermédiaire

Objectifs : interfacer une application PyQt/Qt Widgets avec une base de données relationnelle

Prérequis : Programmer par modèle-vue

Jusqu'ici, la persistance des données de BiblioApp a été faite sur fichier en utilisant le format JSON. Si cette solution simple convient à certaines applications, elle comporte certaines limitations : le fichier doit être lu et écrit en entier, même si on a juste fait une minuscule modification comme corriger le nom d'un auteur ; on doit charger en mémoire vive toutes les données ; l'application est mono-utilisateur en ce qui concerne les mises à jour (même si le fichier est sur un disque partagé, le dernier utilisateur qui écrit le fichier écrase les modifications éventuelles faites par un autre utilisateur), etc. Par ailleurs, il y a une redondance des données : les informations "auteur", "éditeur" et "genre" sont recopiées dans chaque livre. En ayant une structure de données plus intelligente, on pourrait certainement factoriser ces informations et avoir juste des références au niveau de chaque livre ; ainsi, si le nom d'un éditeur doit être changé, il suffit de faire une correction localisée plutôt qu'avoir à éditer tous les livres de cet éditeur ; on pourrait par ailleurs avoir à ajouter des informations sur chaque éditeur (adresse, e-mail, etc.) qu'il serait absurde d'avoir à répliquer au niveau de chaque livre. Pour toutes ces raisons, il est clair que le mode de stockage sur fichier tel qu'exposé au chapitre Développer avec Qt Designer ne convient bien que pour la gestion d'une petite collection personnelle ; il est inadapté pour une bibliothèque municipale ou nationale.

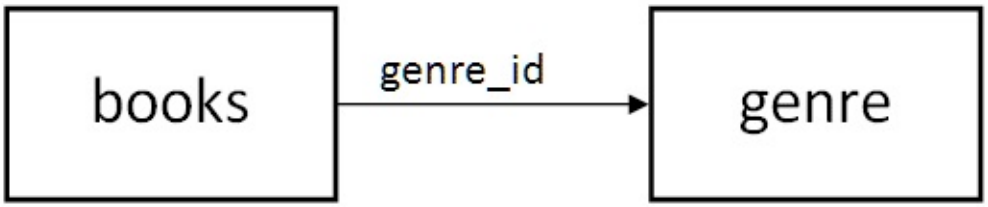
Pour pallier ces limitations, la solution par excellence consiste à utiliser une base de données. Il existe plusieurs types de bases de données ; nous nous pencherons dans ce chapitre sur le type sans doute le plus répandu, les bases de données relationnelles. Nous allons montrer comment l'application BiblioApp développée précédemment peut être adaptée pour stocker ses données dans une base de données relationnelle. Comme le sujet est très vaste, nous nous limiterons aux concepts de base et leur incarnation dans PyQt.

***Note** > Nous considérons dans ce qui suit que vous avez quelques notions du SQL. Si ce n'est pas le cas, reportez-vous à la Section 1, Deux mots sur SQL.*

1. Conception de la base de données

La première chose à faire quand on veut créer une base de données relationnelle est de définir les tables, leurs attributs et leurs relations. Pour les besoins de BiblioApp, nous pourrions nous contenter d'une seule table `Livres`, très similaire au fichier JSON utilisé jusqu'ici. Toutefois, comme évoqué dans l'introduction, un des intérêts des bases de données relationnelles est de pouvoir éviter les redondances : dans notre cas, il s'agit des attributs `auteur`, `éditeur` et `genre`. Pour la facilité, nous nous limiterons ici à la normalisation liée à l'attribut `genre`. Nous allons donc créer deux tables, `Livres` et `Genres` comme évoqué dans notre introduction à SQL, où chaque livre référence un genre selon un code `genre_id` (voir Figure 9.1).

Figure 9.1 : *Base de données Bibliothèque*



Dans les bases de données relationnelles plus élaborées, ce genre de diagramme, appelé schéma entités-relations, est très important à établir avant de se lancer dans l'implémentation. Il n'est pas rare d'avoir plusieurs dizaines de tables liées par un réseau de relations intriquées.

Une fois les schémas entités-relations établis, il y a des choix techniques à faire quant à l'implémentation :

- choix d'un système de gestion de bases de données (SGBD) : SQLite, MySQL, Oracle, PostgreSQL, etc. ;
- choix de la bibliothèques d'accès (API) : classes PyQt ou bibliothèques d'accès Python (API MySQLdb, etc.).

Dans la vie réelle, le choix de la SGBD — s'il n'est pas déjà déterminé — dépendra de nombreuses considérations : nombre d'utilisateurs, volume des transactions, robustesse, mode de licence, prix, pérennité, facilité de la maintenance, etc. Dans ce qui suit, nous faisons un choix pragmatique pour pouvoir démontrer les choses le plus aisément

possible : le SGBD sera SQLite auquel nous accéderons par les classes QSqlDatabase et QSqlQuery.

Par ailleurs, les modifications dans la table `genre` ne seront prises en compte qu'au redémarrage de l'application (encore une fois, pour simplifier). Typiquement, si l'on veut ajouter un nouveau genre littéraire, après un `INSERT` dans cette table, il faudra redémarrer l'application pour voir le nouveau genre apparaître dans la liste déroulante.

2. Évolutions dans BiblioApp

Nous allons à présent expliquer comment faire évoluer le code de l'application BiblioApp pour accéder à une base de données SQLite telle que présentée plus haut plutôt qu'à un fichier. Notons qu'il s'agit d'une évolution qui va au-delà d'un simple changement technique : elle implique le modèle de données et l'interface utilisateur. Nous visons une nouvelle application qui se connecte automatiquement à une base de données et qui gère automatiquement les modifications des enregistrements, sans avoir à demander à l'utilisateur de sauvegarder chaque fois toutes les données.

Note *> Le code source final de ce chapitre est disponible dans le projet* `acces_bd` .

2.1. Nettoyage préliminaire

Comme on part d'une application existante qui sauvegarde sur fichier, on va commencer par enlever tout ce dont nous n'avons plus besoin, c'est-à-dire tout ce qui concerne de près ou de loin la persistance sur fichier. Par ailleurs, on peut enlever les genres de livre définis "en dur" au niveau de la liste déroulante du fichier `.ui`.

Voici dès lors les quelques actions de nettoyage à effectuer (il est bien sûr conseillé de créer un nouveau projet en clonant le projet BiblioApp existant) :

1. dans `main_window_biblio.ui` : supprimer les sous-menus Ouvrir... et Enregistrer, effacer les choix "---", "Biographie"... de la liste déroulante `comboBoxGenre` (attention à bien recompiler la feuille `.ui`, après édition dans Qt Designer) ;

2. dans `modele_biblio.py` : supprimer les méthodes `creeDepuisFichier` et `enregistreDansFichier` ;

3. dans `main_window.py` : supprimer l'attribut `nomFichierBiblio` et les méthodes `closeEvent`, `on_actionOuvrir_triggered`, `on_actionEnregistrer_triggered` et `modificationAEnregistrer` et les appels à ces méthodes.

2.2. Modification du modèle

Comme on a suivi une conception MVD, le changement du mode de persistance des données concerne essentiellement le modèle. Dans la classe `ModeleTableBiblio`, on va changer le constructeur pour créer la base de données ou la lire et modifier les méthodes `ajouteLivre`, `remplaceLivre` et `supprimeLivre`. Commençons donc par éditer `modele_biblio.py`.

Comme SQLLite stocke sa base de données dans un fichier unique, on va définir un nom de fichier dans une variable :

```
NOM_FICHIER_DB = "bibliotheque.db"
```

***Note** > Dans d'autres SGBD, ce sera un peu plus compliqué : il faudra typiquement définir un nom de machine, un port, un nom de base de données, un nom d'utilisateur et un mot de passe.*

Ensuite, on ajoute dans le tuple nommé `Livre` un attribut `idLivre`, stockant l'identificateur du livre dans la base de données.

```
Livre = namedtuple('Livre', ('idLivre', 'titre', 'auteur', 'editeur',
                             'genre', 'annee', 'resume', 'prix'))
```

À l'inverse des autres attributs, `idLivre` est caché à l'utilisateur, il servira uniquement lors des mises à jour ou suppressions pour identifier l'enregistrement concerné. Attention : ceci nécessite une petite adaptation dans la méthode `data`: il faut incrémenter le numéro de colonne, sachant que le nouveau champ fait tout décaler d'une position à droite :

```
    def data(self,index,role):
        if role == Qt.DisplayRole and index.isValid():
            return self.livres[index.row()][index.column()+1]
        return QVariant()
```

Voici à présent le constructeur `ModeleTableBiblio.__init__` mis à jour, avec trois nouvelles méthodes `creeBD`, `litBD` et `genres`.

```
…
from PyQt5.QtSql import QSqlDatabase, QSqlQuery
from collections import namedtuple,  OrderedDict
import os
…

class ModeleTableBiblio(QAbstractTableModel):
```

```
    …
    def __init__(self):
```

❶ super(ModeleTableBiblio,self).__init__() bdExiste = os.path.exists(NOM_FICHIER_BD) ❷ bd = QSqlDatabase.addDatabase('QSQLITE') ❸ bd.setDatabaseName(NOM_FICHIER_BD) bd.open() if not bdExiste: ❹ self.creeBD() self.litBD() def creeBD(self): QSqlQuery(''' CREATE TABLE genres (❺ genre_id INTEGER PRIMARY KEY, genre TEXT) ''') QSqlQuery(''' CREATE TABLE books (book_id INTEGER PRIMARY KEY, title TEXT, author TEXT, publisher TEXT, genre_id INTEGER, year INTEGER, summary TEXT, price FLOAT) ''') QSqlQuery(''' INSERT INTO genres (genre) VALUES ("---"), ("Biographie"), ("Fantastique"), ("Historique"), ("Policier"), ("Science-Fiction") ''') def litBD(self): query = QSqlQuery(''' SELECT genre_id, genre ❻ FROM genres ORDER BY genre_id ''') self.idParGenre = OrderedDict() ❼ while query.next(): ❽ self.idParGenre[query.value(1)] = query.value(0) query = QSqlQuery(''' SELECT book_id, title, author, publisher, ❾ genre, year, summary, price FROM books JOIN genres ON genres.genre_id = books.genre_id ''') self.livres = [] while query.next(): livre = Livre(*(query.value(i) for i in range(8))) ❿ self.livres.append(livre) def genres(self): ⓫ return self.idParGenre.keys()

Voici quelques explications sur ces méthodes.

❶ Attention : on a enlevé l'argument `livres` ! La liste de livres sera dorénavant créée à l'intérieur du modèle, par lecture dans la base de données.

❷ On teste l'existence du fichier de base de données et on sauvegarde le résultat dans une variable car on a besoin de cette information plus loin (voir point ❹) ; si le fichier n'existe pas, il est créé dans les instructions qui suivent.

❸ Par ces trois instructions, on instancie QSqlDatabase , on crée le fichier de base de données SQLite s'il n'existe pas encore et on ouvre la connexion ; à partir de ce moment, on peut exécuter des requêtes SQL dans la base de données.

❹ Si c'est la première exécution, on appelle `creeBD` qui va créer les tables dans la base de données ; on appelle ensuite `litBD` pour lire les données de la base de données ; cette méthode est appelée même la première fois, juste après `creeBD` ;

c'est une manière de s'assurer que l'initialisation de `ModeleTableBiblio` fonctionne bien, même avec une base de données sans livres.

❺ La méthode `creeDB` exécute des requêtes SQL via la classe QSqlQuery pour créer les tables `genres`, `books` et insérer quelques genres littéraires dans la table correspondante. En fait, cette méthode pourrait être complètement séparée du programme principal et exécutée lors du déploiement. Les instructions `CREATE TABLE` sont appelées DDL (*Data Definition Language*) et, souvent, elles sont définies dans des fichiers dédiés et exécutés hors PyQt via une ligne de commande par un administrateur de base de données.

❻ La méthode `litBD` va lire les genres littéraires définis dans la table `genres`. À cette fin, on crée la requête SQL `SELECT` dans une instance de QSqlQuery ; on a ajouté une clause `ORDER BY` pour être sûr de retrouver les genres dans l'ordre de définition (ceci se verra sur la liste déroulante).

❼ On va stocker les genres et leurs identificateurs dans un dictionnaire afin de pouvoir facilement retrouver l'identificateur d'un genre donné. Comme on veut aussi pouvoir afficher les genres dans la liste déroulante en conservant leur ordre, on utilise un `OrderedDict`, importé du module `collections` (voir au début du listing). Il s'agit d'un dictionnaire qui conserve l'ordre d'insertion des éléments, à l'inverse du dictionnaire standard Python.

❽ On retrouve les enregistrements un par un dans une boucle qui avance en invoquant `next()` sur cette instance ; `next()` renvoie `true` ou `false` selon qu'il y a encore des enregistrements à lire ou non. On extrait un élément d'un enregistrement en appelant la méthode `value` avec l'indice de l'élément dans la requête, le premier étant 0. On insère les éléments retrouvés dans le dictionnaire, la clé étant le libellé du genre (indice 1) et la valeur étant son identificateur (indice 0).

❾ On fait une requête SQL `SELECT` pour retrouver les livres stockés dans la table `books`, la jointure avec la table `genre` permet d'obtenir directement le genre littéraire en toutes lettres plutôt que son code d'identification.

Comme on a pris soin d'écrire le `SELECT` en respectant l'ordre des huit attributs

⓾ définis dans le tuple nommé `Livre`, on peut se permettre d'instancier chaque tuple livre en faisant une *expression génératrice* équivalente à `Livre(query.value(0), query.value(1), … , query.value(7))` ; chaque livre est ensuite placé dans la liste `self.livres`, comme dans la première version de BiblioApp.

⓫ On termine par une petite méthode qui va renvoyer la liste des libellés des genres, qui sont les clés du dictionnaire qui a été construit dans `litBD`. Cette méthode est utile pour bien encapsuler la structure de données interne du modèle.

Bien évidemment, l'intérêt de la table `genre` est peu perceptible dans ce programme. Il y a bien sûr un gain concernant l'espace disque : le livre ne stocke qu'un code identifiant le genre au lieu du libellé complet. Toutefois, l'intérêt principal apparaîtra quand on voudra par exemple modifier le libellé d'un genre : il suffira d'un `UPDATE` sur un enregistrement de la table `genre`.

Nous allons à présent modifier les méthodes `ajouteLivre`, `remplaceLivre` et `supprimeLivre` pour faire une mise à jour directe dans la base de données (plus besoin de demander à l'utilisateur de faire des sauvegardes). La signature de ces méthodes reste inchangée et l'implémentation actuelle qui met à jour le modèle va être augmentée d'un prétraitement pour que cette mise à jour soit persistante.

Voici la méthode `ajouteLivre` mise à jour (les quatre dernières lignes sont inchangées) :

```
    def ajouteLivre(self,livre):
        query = QSqlQuery()
        query.prepare(''' INSERT INTO books
                                (book_id, title, author, publisher,
                                 genre_id, year, summary, price)
                          VALUES (NULL, :titre, :auteur, :editeur,
                                  :idGenre, :annee, :resume, :prix)
''')
        query.bindValue(":titre",livre.titre)
        query.bindValue(":auteur",livre.auteur)
        query.bindValue(":editeur",livre.editeur)
        query.bindValue(":idGenre",self.idParGenre[livre.genre])
        query.bindValue(":annee",livre.annee)
        query.bindValue(":resume",livre.resume)
        query.bindValue(":prix",livre.prix)
        query.exec_()

        idLivre = query.lastInsertId()
        livre = Livre(idLivre,*livre[1:])
```

```
        indiceLivre = len(self.livres)

        self.beginInsertRows(QModelIndex(),indiceLivre,indiceLivre)
        self.livres.append(livre)
        self.endInsertRows()
```

On crée ici une requête `INSERT` qui contient des paramètres formels (préfixés par deux points). Ces paramètres sont ensuite remplacés, via les appels à `bindValue`, par les valeurs correspondantes du livre à ajouter. Puis, l'exécution de la requête par `query.exec_()` ajoute effectivement le livre donné de la table `books` en base de données.

Expliquons quelques points concernant l'assignation des paramètres.

- Pour faire le lien avec les attributs codés en Python, on a utilisé des paramètres formels ayant une convention et une langue différente de celles suivies dans la base de données. Cela ne pose aucun problème technique, cela illustre juste un problème assez fréquent en informatique quand deux mondes de culture différente se rencontrent, en l'occurrence celui de la programmation en Python/PyQt et celui des bases de données. Idéalement, quand c'est possible, on veillera à uniformiser les conventions. Sinon, on veillera à ce que les interfaces vers la base de données soient bien localisées et encapsulées.

- Pour le paramètre formel `:idGenre`, on utilise le dictionnaire `self.idParGenre` lu à partir de la table `genre` de la base de données ; on retrouve ainsi facilement l'identificateur associé au libellé `livre.genre` (par exemple : Historique → 3).

- On a mis la valeur spéciale `NULL` pour le champ `bookId` ; c'est une manière de demander à SQLite de choisir lui-même un numéro unique pour le nouveau livre (il fait cela par simple incrémentation). C'est un des services qu'offrent les SGBD ; il peut être très utile pour garantir qu'il n'y ait pas de doublons dans les applications multiutilisateurs. Après insertion dans la base de données, il faut retrouver l'identificateur assigné par le SGBD via la méthode `lastInsertId()` et le stocker dans le livre en mémoire ; à cette fin, comme un tuple Python est non modifiable, on crée un nouveau tuple `Livre` avec l'identificateur en premier argument suivi des autres attributs inchangés à partir de l'indice 1 (l'expression Python utilisée ici est équivalente à `Livre(idLivre,livre[1],livre[2], …,livre[7])`).

Passons ensuite à la méthode `supprimeLivre` (les trois dernières lignes sont inchangées).

```
    def supprimeLivre(self,indiceLivre):
```

```
        query = QtSql.QSqlQuery()
        query.prepare(''' DELETE
                          FROM books
                          WHERE book_id = :idLivre ''')
        query.bindValue(":idLivre",self.livres[indiceLivre].idLivre)
        query.exec_()

        self.beginRemoveRows(QModelIndex(),indiceLivre,indiceLivre)
        del self.livres[indiceLivre]
        self.endRemoveRows()
```

On crée ici une requête `DELETE` en lui passant l'identificateur du livre à supprimer dans la condition `WHERE`. Attention : cet identificateur *n'est pas* `indiceLivre`, qui est l'indice du livre dans le tableau du modèle en mémoire. Pour retrouver l'identificateur du livre à supprimer, il faut d'abord retrouver l'objet livre dans le modèle et ensuite extraire l'attribut identificateur : c'est le sens de l'expression `self.livres[indiceLivre].idLivre`.

Pour terminer, voici la méthode `remplaceLivre` mise à jour (les deux dernières instructions sont inchangées) :

```
    def remplaceLivre(self,indiceLivre,livre):
        query = QSqlQuery()
        query.prepare(''' UPDATE books
                          SET title = :titre,
                              author = :auteur,
                              publisher = :editeur,
                              genre_id = :idGenre,
                              year = :annee,
                              summary = :resume,
                              price = :prix
                          WHERE book_id = :idLivre ''')
        query.bindValue(":idLivre", self.livres[indiceLivre].idLivre)
        query.bindValue(":titre", livre.titre)
        query.bindValue(":auteur", livre.auteur)
        query.bindValue(":editeur", livre.editeur)
        query.bindValue(":idGenre",self.idGenres[livre.genre])
        query.bindValue(":annee", livre.annee)
        query.bindValue(":resume", livre.resume)
        query.bindValue(":prix", livre.prix)
        query.exec_()

        self.livres[indiceLivre] = livre
        self.dataChanged.emit(self.createIndex(indiceLivre,0),
                              self.createIndex(indiceLivre,2))
```

Comme on ne sait pas a priori quels champs ont changé, on fait un `UPDATE` sur tous les champs, sauf (bien sûr) l'identificateur du livre qui est indiqué dans la condition `WHERE`.

Pour une explication sur les associations `:idGenre` et `:idLivre`, référez-vous respectivement aux méthodes `ajouteLivre` et `supprimeLivre` vues plus haut.

2.3. Modifications dans la fenêtre principale

Comme nous avons gardé quasiment la même interface pour le modèle, il y a peu de choses à changer dans la classe `MainWindowBiblio` (`main_window_biblio.py`).

Le constructeur

Le premier point à modifier concerne le constructeur. Il y a une ligne à modifier et une ligne à ajouter :

```
class MainWindowBiblio(QMainWindow, Ui_MainWindowBiblio):

    def __init__(self, parent=None):
        …
        self.modeleTableBiblio = ModeleTableBiblio()

self.comboBoxGenre.addItems(self.modeleTableBiblio.genres())
        …
```

Comme vu au chapitre Programmer par modèle-vue, `ModeleTableBiblio` gère seule ses données par accès à la base de données ; il n'y a plus lieu de lui passer une liste vide pour s'initialiser. Par ailleurs, comme les genres littéraires sont à présent chargés dans le modèle, il faut les placer nous-mêmes dans la liste déroulante `comboboxGenre` ; ceci se fait très facilement par utilisation de la méthode `addItems`, qui charge en un coup une liste de chaînes de caractères ; cette liste est obtenue par appel à la méthode `genres()` que nous avons définie plus haut, dans le modèle.

La méthode on_pushButtonSauvegarder_clicked

La seconde méthode à changer est `on_pushButtonSauvegarder_clicked` :

```
    @pyqtSlot()
    def on_pushButtonSauvegarder_clicked(self):
        livre = Livre( idLivre = None,
                       titre = self.lineEditTitre.text(),
                       …
        …
```

Comme on a ajouté l'attribut `idLivre` dans le tuple nommé `Livre`, il faut lui

donner une valeur : on se contente de lui donner la valeur `None`, sachant que l'identificateur de livre est géré complètement par le modèle (voir les méthodes `ajouteLivre` et `remplaceLivre` plus haut).

Une fois tous ces changements faits, on peut lancer l'application et constater qu'elle fonctionne comme avant, à ceci près que l'utilisateur ne doit plus ouvrir et enregistrer de fichier. On peut fermer et relancer l'application, la liste de livres sauvegardée dans la base de données réapparaît inchangée.

Astuce *> Si l'application ne fonctionne pas comme attendu, il y a lieu de vérifier la bonne exécution des requêtes SQL. À cette fin, l'instruction suivante (à placer juste après l'exécution d'une requête) peut être très utile :*

```
print (query.lastError().text())
```

3. Pour aller plus loin…

Dans l'exemple qui a été donné, nous avons choisi pour la facilité de charger la table `genre` dans un dictionnaire Python (l'attribut `idParGenre`) ; c'est fait une fois pour toutes, au démarrage de l'application et, comme nous l'avons vu, ça permet de faire facilement les `INSERT` et `UPDATE`. Cela dit, ce n'est pas forcément le meilleur choix pour des applications où les tables référencées sont mises à jour souvent. SQL, qui est un langage plus riche qu'il n'y parait, permet en fait de se passer d'un dictionnaire en faisant travailler le SGBD plutôt que le programme Python.

Voici comment les requêtes `INSERT` et `UPDATE` pourraient être transformées (sous SQLite) pour retrouver elles-mêmes l'attribut `genre_id` :

```
        query.prepare(''' INSERT INTO books
                                (book_id, title, author, publisher,
                                 genre_id, year, summary, price)
                          SELECT NULL, :titre, :auteur, :editeur,
                                    genre_id, :annee, :resume, :prix
                          FROM genres
                          WHERE genres.genre = :genre ''')
        …
        query.prepare(''' UPDATE books
                          SET title = :titre,
                              author = :auteur,
                              publisher = :editeur,
                              genre_id = (SELECT genres.genre_id
                                          FROM genres
                                          WHERE genres.genre =
:genre),
                              year = :annee,
                              summary = :resume,
                              price = :prix
                          WHERE book_id = :idLivre ''')
```

Dès lors, on pourrait faire le lien direct avec le libellé du genre littéraire :

```
        query.bindValue(":genre", livre.genre)
```

Dans cette approche, l'application PyQt n'a donc même plus à connaître les valeurs de l'attribut `genre_id`.

Terminons ce chapitre par quelques commentaires généraux. Comme déjà signalé, SQLite est un SGBD simple et de bon goût qui ne convient toutefois pas pour des

applications multiutilisateurs ou nécessitant de la sécurité. Il y a lieu de bien s'informer sur les possibilités et limitations de chaque système pour ne pas se trouver bloqué ou avoir à faire des migrations pénibles. Pour une liste comparative des SGBD relationnels, vous pouvez vous référer comme point d'entrée à l'article *Comparison of relational database management systems* sur Wikipedia.

Par ailleurs, concernant l'accès programmatique au SGBD, le choix de la classe PyQt QSqlDatabase que nous avons fait pour l'exemple n'est pas incontournable : l'écosystème Python contient plusieurs modules pour accéder aux bases de données, parfois avec une interface plus légère que celle offerte par PyQt (par exemple, les requêtes `SELECT` génèrent directement des tuples Python pour représenter les enregistrements sélectionnés). Pour canaliser les efforts et favoriser l'interopérabilité, une spécification d'API Python a été définie, sous le nom de code PEP 249. Comme modules souscrivant à cette spécification, on citera notamment le module standard Python sqlite3 — accessible sans rien installer, donc ! — et le module tiers MySQLdb qui permet de se connecter à MySQL.

Nous n'irons pas plus loin dans le cadre de cet ouvrage, mais je vous recommande vivement si vous voulez faire du développement sérieux avec une base de données de vous documenter sur le sujet et lire les manuels SQL.

Affichage 2D interactif avec les vues graphiques

Comment dessiner en PyQt ?

Dans Développement d'une application avec des widgets, nous avons vu comment créer une application GUI classique, basée sur des widgets standards (champs textuels, listes, boutons, cases à cocher, etc.). Cette approche met à la disposition du programmeur une large boîte à outils qui permet de composer des fenêtres relativement aisément, en s'aidant notamment de Qt Designer. Toutefois, certaines applications nécessitent des éléments graphiques qui n'ont aucun point commun avec les widgets proposés : un histogramme, un diagramme en camembert, une fonction trigonométrique, etc., dans les cas les plus simples ; mais aussi, pour des applications plus complexes, une vue temporelle de type agenda, des orbites de satellites, des graphes avec des nœuds et des arcs, une carte avec éléments textuels surimposés, des facilités de zoom/scrolling, des animations, des objets 3D, etc.

La technique de base pour faire ce genre de graphiques en PyQt consiste à utiliser la classe QPainter. Elle offre les primitives pour afficher des images ou dessiner des formes géométriques simples (segments de droite, polygones, ellipses, etc.) dans un système de coordonnées écran, c'est-à-dire des couples de nombres entiers représentant des positions de pixels. QPainter est accompagnée d'autres classes qui forment un système complet de dessin appelé Paint System[12]. Ce système, quoique très puissant, peut exiger toutefois beaucoup d'efforts de la part du programmeur : des fonctions comme le zoom, l'animation, le glisser/déposer, la détection de collisions, la sélection d'un élément pointé par le curseur… manquent à l'appel et doivent donc être soigneusement programmées.

Pour pallier ces difficultés, Qt a — assez logiquement — proposé de nouvelles classes qui s'appuient sur le Paint System pour apporter les fonctionnalités manquantes. Ainsi, déjà en 2001 avec Qt 3, la classe QCanvas apportait un lot important de fonctions graphiques avancées. Cette technologie a finalement abouti au framework Graphics View, apparu en 2006 avec Qt 4.2. Ce dernier a remplacé QCanvas en proposant de nouvelles abstractions, ainsi que des fonctions plus riches et plus flexibles. Depuis Qt 4.2, jusqu'aux dernières versions de Qt 5, il ne cesse d'être enrichi et amélioré.

Le framework Graphics View est destiné principalement aux graphiques en 2D (la 3D est possible, mais nécessite des techniques avancées). En 2016, le module Qt 3D a été intégré à Qt 5.7 : il offre une palette étendue de fonctions de haut niveau pour la visualisation 3D (et, en fait, *aussi* la 2D !) ; en outre, son architecture ouverte permet l'ajout d'extensions comme la simulation physique, la détection de collision et la

recherche de chemin.

Note *> Les graphiques 3D sont supportés depuis Qt4 par des modules et classes Qt dédiés qui permettent notamment l'accélération matérielle via OpenGL. Ainsi, le module Qt OpenGL dont les classes sont préfixées par QGL (par exemple : QGLWidget). Depuis Qt 5.4, ce module est rendu obsolète par un ensemble de nouvelles classes intégrées au module Qt GUI (elles ont le préfixe QOpenGL, par exemple : QOpenGLWidget) ; ces dernières classes sont à utiliser pour tout nouveau projet.*

Qu'en est-il aujourd'hui pour nous, programmeurs Python/PyQt5 qui voulons réaliser des graphiques ? Quelque part, nous avons l'embarras du choix mais chaque technique a, grosso modo, son domaine d'application et chacune propose au programmeur des abstractions très différentes. La couche graphique basée sur la classe QPainter est toujours présente et accessible ; elle permet de tout faire… pour qui veut programmer avec des primitives de bas niveau. On a par ailleurs un framework Graphics View mûr et performant, qui offre des fonctions de très haut niveau. Pour la 3D, les classes QOpenGL conviendront sans doute aux programmeurs rompus à l'API OpenGL; enfin, pour les programmeurs qui n'ont pas peur d'apprendre une nouvelle technologie, il y a le nouveau module Qt 3D, qui propose une API moderne et extensible (voir Affichage 3D avec Qt 3D).

Dans ce chapitre, nous allons nous pencher sur le framework Graphics View, qui répond efficacement aux besoins de dessins 2D.

---

[12] Nommé originellement "Arthur", même si ce nom n'apparaît presque plus dans la documentation.

10

Introduction au framework Graphics View de Qt

Niveau : débutant

Objectifs : présenter les concepts de base du framework Graphics View de Qt

Le framework Graphics View de Qt met à disposition du développeur une surface permettant de dessiner, transformer et animer un grand nombre d'éléments. Il est composé de plusieurs dizaines de classes ayant le préfixe `QGraphics` qui collaborent étroitement les unes avec les autres. Le framework définit des abstractions de haut niveau qui s'inspirent notamment du principe déjà rencontré de séparation entre modèle et vue. Il propose un catalogue d'éléments graphiques de base que vous pouvez facilement étendre pour les besoins de votre application.

Le framework Graphics View a un spectre d'utilisation très large : graphes, vues cartographiques, conception assistée par ordinateur, jeux 2D, etc. Grâce à des optimisations efficaces, il brille particulièrement lorsqu'il y a beaucoup d'éléments à afficher : ainsi, dès la première version du framework (Qt 4.2), était fournie une application d'exemple affichant 40 000 puces électroniques, chacune pouvant être sélectionnée individuellement. Par ailleurs, il est intéressant de noter que Qt Quick, dans sa version 1.0, était implémenté sur la base du framework Graphics View (voir encadré à la Section 3, Choix d'une interface).

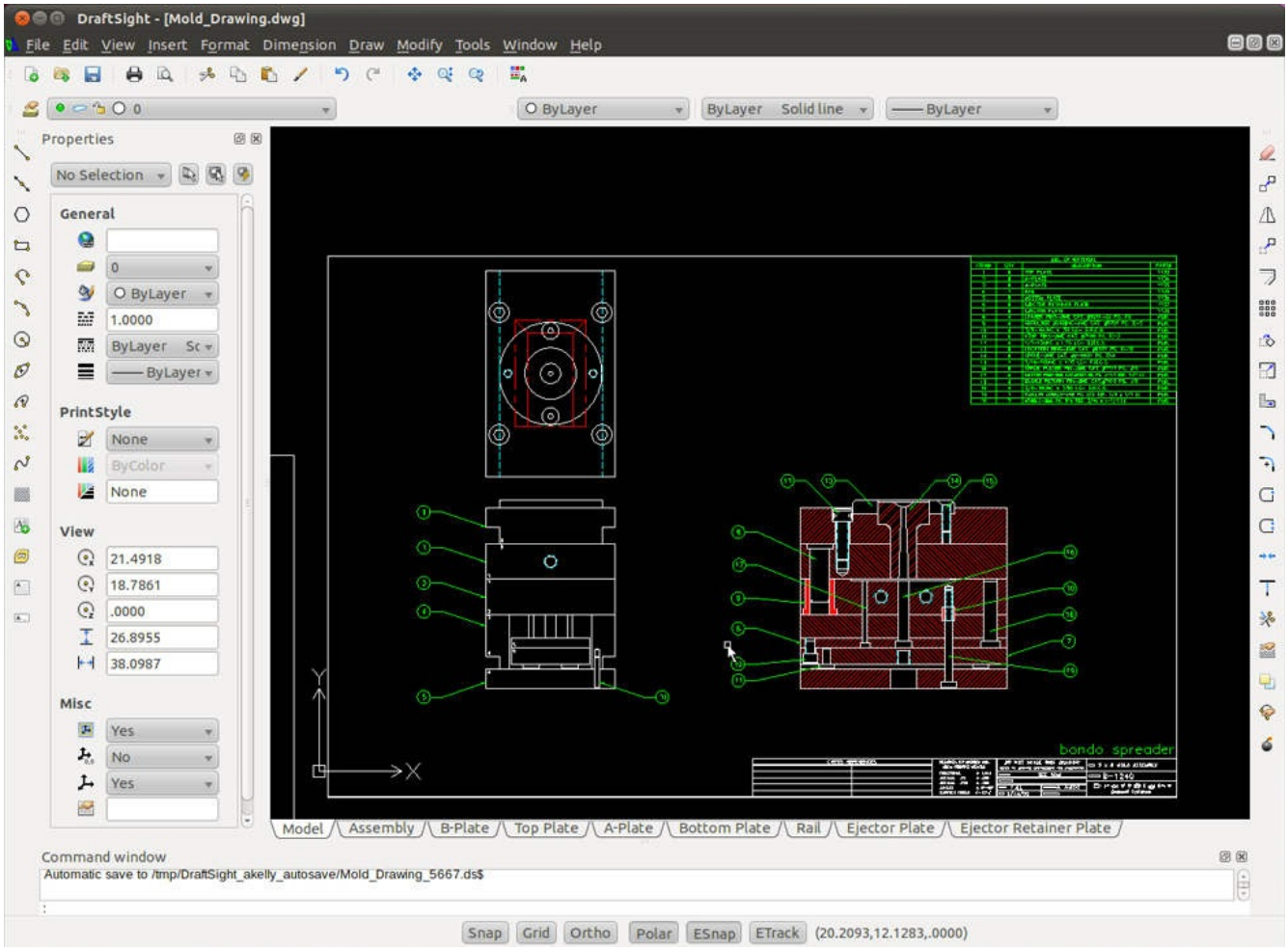
1. Quand utiliser le framework Graphics View ?

En règle générale, le framework Graphics View est recommandé (au moins) pour intégrer à une application des graphiques 2D contenant des objets sur lequel l'utilisateur peut interagir. Pour citer quelques interactions typiques : zoom, défilement, copier-coller, glisser-déposer, sélection par pointage ou lasso, etc. Voici quelques exemples d'utilisations possibles où le framework est particulièrement bien adapté :

- éditeurs de dessin vectoriel ;
- vues cartographiques ;
- plans de bâtiments/d'installations/réseaux (centres de contrôle, par exemple) ;
- conception (architecture, schémas électriques, circuits intégrés, CAO, PAO, etc.) ;
- atelier de génie logiciel (diagrammes de classes, de séquences, etc.) ;
- GUI non conventionnelles (arbres généalogiques, apprentissage pour enfants, *mind map*, etc.),
- jeux 2D.

À titre d'illustration, l'image suivante montre un exemple d'application réalisable avec le framework Graphics View : DraftSight de Dassault Systèmes. Cette application dédiée au dessin technique et à la CAO peut tirer parti de l'approche Graphics View car il y a beaucoup d'interactions avec l'utilisateur (ajout, déplacement, suppression d'éléments, zoom, etc.) ; par ailleurs, les éléments dessinés ont des attributs différents et ils peuvent chacun avoir des réactions spécifiques aux actions de l'utilisateur.

Figure 10.1 : *DraftSight — un exemple d'application Qt*



En fait, le framework peut être utilisé pour *tout* graphique 2D, du plus simple au plus compliqué. Seulement, pour des graphiques simples, statiques, avec peu d'interactions, le programmeur peut sans hésiter utiliser la technique de base avec la classe QPainter, surtout s'il se sent plus à l'aise avec cette approche. Le framework Graphics View — ne le cachons pas — exige un minimum de réflexion avant de se mettre à programmer. Dans le même ordre d'idée, signalons à ce sujet qu'il faudra abandonner l'espoir de colorier directement des pixels à partir de leurs coordonnées sur l'écran !

Pour dissiper tout malentendu, signalons que le framework Graphics View *n'est pas* une technique alternative au développement PyQt avec widgets ; il propose en fait un *super*-widget qui s'ajoute aux autres et qui permet de placer une zone graphique dans une fenêtre classique. Ainsi, dans l'exemple d'application donné, seule la partie graphique sur fond noir est réalisée avec le framework ; tout ce qui l'entoure est constitué de widgets PyQt classiques. Donc, tout ce que nous avons appris jusqu'ici reste utile et peut être mis à profit pour interagir avec une vue graphique. Par exemple, on peut programmer une case à cocher QCheckBox ou une action menu pour faire apparaître/disparaître des éléments graphiques ou encore faire afficher une boîte de

dialogue contenant des informations sur l'objet qui vient d'être cliqué. En résumé, le framework Graphics View peut donc s'intégrer harmonieusement au sein d'une application PyQt traditionnelle.

2. Scènes, éléments et vues

Pour utiliser le framework Graphics View, il est nécessaire de comprendre ses objets de base. Ils sont au nombre de trois : *scène*, *élément* et *vue*. Chacun correspond à une classe particulière du framework.

Scène

Une *scène* (classe QGraphicsScene) est un espace 2D sur lequel on va placer et déplacer différents éléments géométriques. C'est un espace muni de son propre système de coordonnées XY, qui est différent de celui de l'écran et qui peut d'ailleurs être bien plus vaste. Un point un peu surprenant quand on débute est que la scène *ne s'occupe pas directement de l'affichage* : elle ne sait pas dessiner et n'a même aucune idée de ce qu'est un écran ou un pixel ! La scène définit principalement une structure de données optimisée pour positionner un grand nombre d'éléments et accéder efficacement à ceux-ci.

Élément

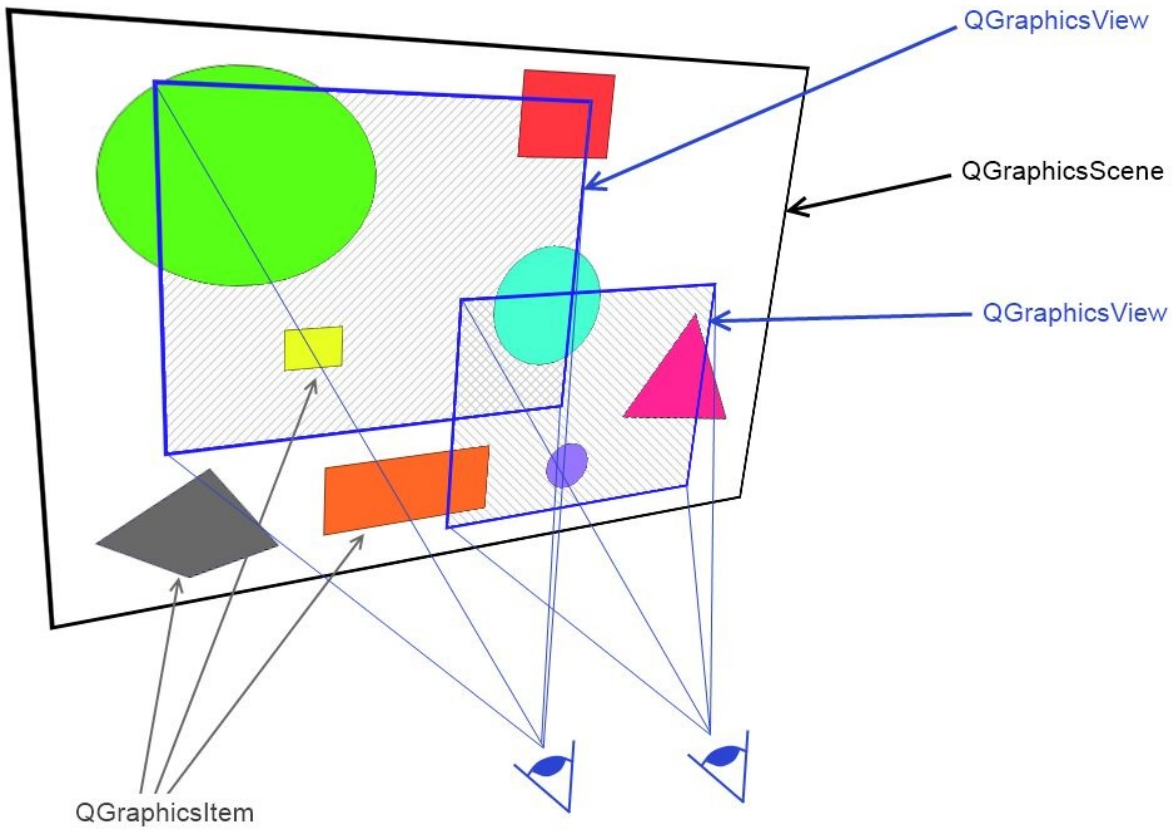
Un *élément* (classe QGraphicsItem) est un objet géométrique placé sur la scène. Il a des coordonnées (*x*,*y*) définies dans le système de coordonnées de la scène. Il définit très précisément les données et méthodes nécessaires pour dessiner un certain objet 2D : lignes, rectangles, ellipses, textes, etc. Si le cadre donné est principalement le dessin vectoriel, il est possible aussi de placer sur la scène des images matricielles (*bitmap*), lues par exemple dans des fichiers au format PNG, JPG, GIF, etc.

Vue

Une *vue* (classe QGraphicsView) est un widget qui permet de représenter une scène donnée à l'écran. Comme tout widget, la vue s'affiche dans une zone rectangulaire à l'écran qui représente la scène ou une partie de celle-ci. Plus précisément, la vue a pour charge de dessiner, dans le cadre qui lui est alloué à l'écran, les éléments de la scène visibles dans ce cadre (voir cadres hachurés sur la Figure 10.2). Pour ce faire, la vue effectue une transformation pour passer du système de coordonnées de la scène à celui de l'écran ; cette transformation basée sur le calcul matriciel est très puissante : elle permet de gérer les fonctions de zoom, de translation (barres de défilement, par exemple), voire les rotations. La vue, qui hérite de QWidget, se charge de gérer aussi toutes les interactions avec

l'utilisateur et, au besoin, les relayer aux éléments concernés : clic souris (sélection d'un élément, affichage d'un menu contextuel…), frappes clavier (copier-coller, suppression…), etc.

Figure 10.2 : *Scène, éléments et vues*



On l'aura compris : le framework utilise en fait le patron de conception modèle-vue dont nous avons déjà parlé, le modèle étant ici la scène avec tous ses éléments. Une fois qu'on a créé une scène et une vue qui lui est attachée, tout changement fait sur la scène est automatiquement répercuté sur la vue et donc visible à l'écran : ajout/suppression/déplacement d'un élément, changement d'une couleur, etc. Par ailleurs, on peut créer dans la même application plusieurs vues sur la même scène, visibles à tour de rôle ou même toutes en même temps ; ce principe permet par exemple d'avoir en vis-à-vis une vue globale fixe et une vue détaillée qui fait un zoom sur une portion particulière de la scène — un système utilisé souvent dans les applications cartographiques et… les jeux de type STR (stratégie temps-réel).

Dans la suite, nous donnons quelques détails supplémentaires sur les classes — QGraphicsScene, QGraphicsItem et QGraphicsView — qui viennent d'être citées. Il est vivement recommandé, en complément, de consulter la documentation officielle de ces

classes.

2.1. Classe QGraphicsScene

Le rôle de la classe QGraphicsScene est de définir une scène qui contient des éléments instances de QGraphicsItem. Comme expliqué plus haut, la scène ne s'occupe pas de dessiner à l'écran mais de placer, déplacer ou enlever les éléments dans un espace 2D non visible directement. Pour un philosophe réaliste, la scène serait en quelque sorte le monde réel indépendant de la manière dont on l'observe. Rien n'empêche d'ailleurs de créer une scène sans aucune vue, quoique ceci n'ait pas grand intérêt pour une application ! L'espace de la scène a par ailleurs un système de coordonnées propre utilisant des nombres en virgule flottante : ceci permet une meilleure approximation de notre monde physique par rapport à des coordonnées en nombres entiers, propres au monde pixelisé d'un écran d'ordinateur.

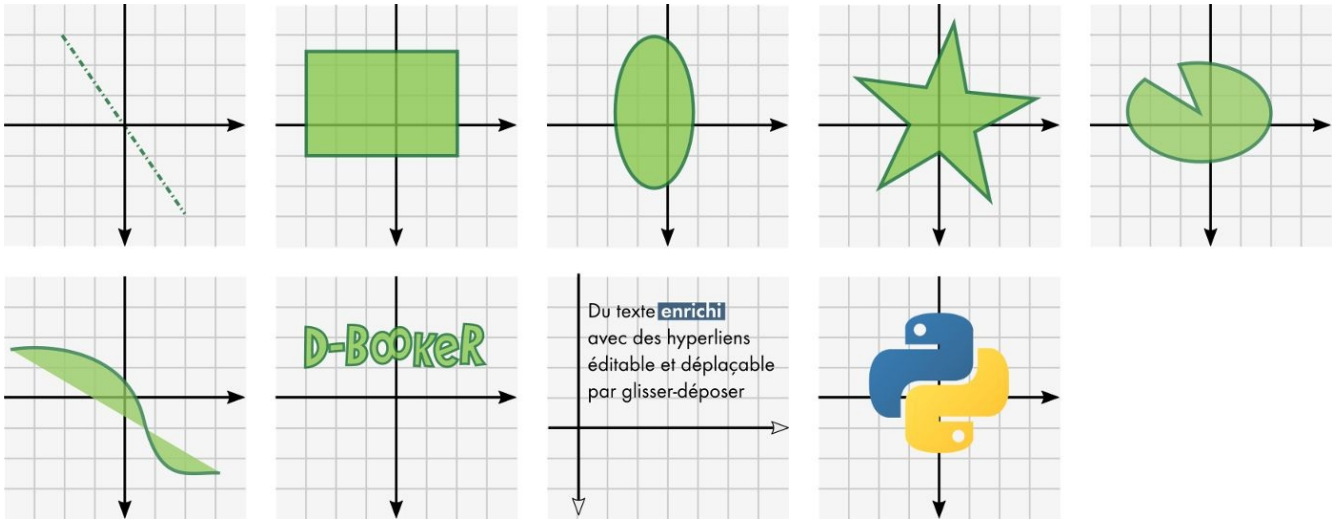
La classe QGraphicsScene a une machinerie interne complexe pour accéder efficacement aux éléments. Heureusement, tout cela est caché au programmeur. La méthode principale offerte par QGraphicsScene est très simple : `addItem(item)` ajoute le QGraphicsItem `item` donné à la scène, aux coordonnées qui lui sont assignées. Aussi, à l'appel de `addItem`, si des vues sont en train d'observer la portion de la scène où l'élément est placé, elles seront rafraîchies automatiquement. Pour faciliter la vie du programmeur, QGraphicsScene propose des méthodes de commodité pour combiner en un seul appel la création de l'élément et son ajout sur la scène (`addLine`, `addRect`, `addEllipse`, etc) ; nous aurons l'occasion d'y revenir au chapitre Première application avec une vue graphique.

2.2. Classe QGraphicsItem

La classe QGraphicsItem définit un élément géométrique à placer sur la scène, avec les données et méthodes nécessaires pour dessiner cet élément.

La classe QGraphicsItem a plusieurs classes filles qui sont directement utilisables ; elles permettent de représenter les éléments géométriques de base : lignes (QGraphicsILineItem), rectangles (QGraphicsIRectItem), polygones (QGraphicsIPolygonItem), ellipses (QGraphicsIEllipseItem), etc. ; par ailleurs, un élément peut aussi définir du texte à afficher (QGraphicsITexttem et QGraphicsISimpleTextItem) et même une image qui peut être chargée à partir d'un fichier (QGraphicsIPixmapItem).

Figure 10.3 : *Les classes filles de QGraphicsItem*



Les classes qui viennent d'être citées sont relativement simples car elles dessinent chacune un objet graphique bien particulier. À côté de cela, iI existe d'autres classes filles de QGraphicsItem qui font des choses *beaucoup* plus sophistiquées : on peut citer par exemple les classes QChart et QGraphicsProxyWidget, filles de QGraphicsWidget qui est elle-même fille de QGraphicsItem[13].

La classe QChart permet de créer facilement des graphiques mettant en relation des séries de nombres, avec des axes et des légendes : on peut dessiner des fonctions trigonométriques, l'évolution des ventes d'un produit, etc. Quant à la classe QGraphicsProxyWidget, qui est sans doute la plus surprenante, elle permet de placer un widget Qt sur la scène. Oui… Il est possible d'intégrer dans une scène donnée un bouton, une case à cocher, un champ texte…, voire un panneau contenant plusieurs widgets ; il existe d'ailleurs des démonstrations technologiques assez impressionnantes où l'on fait subir aux widgets des rotations et des déformations. Toutefois, sachant que QGraphicsView est un widget qui peut cohabiter avec les widgets traditionnels, il n'est pas du tout indispensable d'intégrer des widgets à la scène ; le QGraphicsProxyWidget est plutôt réservé à des usages avancés ou des interfaces non standards (voir l'exemple *Pad Navigator* de Qt).

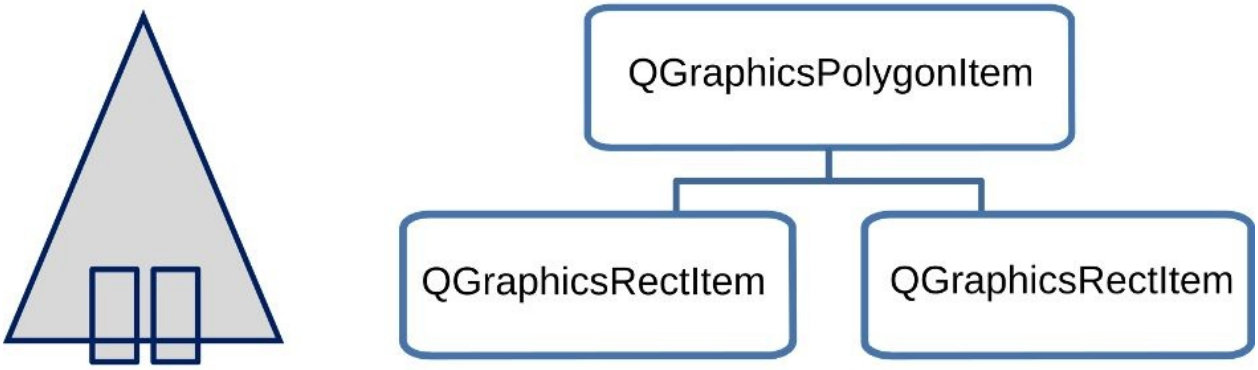
Toutes les classes filles qui viennent d'être citées ont des rôles bien spécifiques, comme dessiner un rectangle, une ellipse, etc. Pour développer les éléments graphiques nécessaires à une application, on va en général devoir étendre les QGraphicsItem pour implémenter des éléments graphiques plus élaborés. À cette fin, on utilise les procédés orientés objet habituels d'héritage et composition.

- *Héritage de QGraphicsItem* — on peut hériter d'une classe fille existante telle

que QGraphicsRectItem pour définir précisément des interactions avec l'utilisateur, c'est-à-dire ce qu'il doit se passer quand on clique sur l'élément, quand le pointeur le survole, etc ; on hérite des méthodes de dessin prédéfinies dans PyQt, mais on surcharge les méthodes liées aux interactions utilisateur. Par ailleurs, il est aussi possible de créer de nouvelles classes qui héritent directement de QGraphicsItem. Il faut alors au minimum indiquer comment elles doivent se dessiner ; pour ce faire, il y a deux méthodes à implémenter (plus précisément, à *surcharger*) : `boundedRect()` et `paint()`. Notons que le premier mode d'héritage cité (à partir d'une classe fille de QGraphicsItem) est très fréquent dans les applications utilisant le framework ; nous allons d'ailleurs l'utiliser dans notre application d'exemple. Le second mode d'héritage (directement de QGraphicsItem) est réservé à un usage avancé permettant notamment d'utiliser directement la classe QPainter, c'est-à-dire la couche graphique bas niveau utilisée par le framework.

- *Composition de QGraphicsItem* — il est possible de grouper des éléments QGraphicsItem pour les manipuler tous ensemble, comme un seul élément. On pourrait par exemple définir un élément VaisseauSpatial en assemblant un QGraphicsPolygonItem pour la coque et deux QGraphicsRectItem pour les réacteurs (voir la Figure 10.4) ; il suffit alors d'une seule instruction pour déplacer cet élément sur la scène ou lui faire subir des rotations.

Figure 10.4 : *Composition de QGraphicsItem*



Ce mode de groupement est récursif : une instance qui groupe plusieurs éléments peut être elle-même groupée dans un élément de plus haut niveau et ainsi de suite. On peut ainsi définir des éléments graphiques complexes selon une arborescence dont les éléments feuilles sont les formes géométriques simples fournies par le framework. Deux techniques distinctes sont possibles pour réaliser ces assemblages :

- un QGraphicsItem E peut être créé en spécifiant un QGraphicsItem P existant comme étant son parent (c'est ce qui est fait dans la Figure 10.4);

- la classe QGraphicsItemGroup, fille de QGraphicsItem, permet de grouper des QGraphicsItem existants.

L'intérêt respectif des deux techniques n'est pas simple à expliquer sans avoir bien compris les différents systèmes de coordonnées. Disons que la technique 1 convient bien quand l'assemblage est fixe et peut être défini à la création, par exemple, le vaisseau spatial dans un jeu ; la technique 2, utilisant QGraphicsItemGroup, est mieux adaptée pour les assemblages d'éléments déjà présents sur la scène, par exemple pour fournir à l'utilisateur une fonction *grouper*.

Parlons, pour terminer, de la superposition des éléments sur la scène. Ceci n'est pas anodin : outre l'effet visuel, la superposition d'éléments peut empêcher l'utilisateur de cliquer sur l'élément qui l'intéresse, si occulté par un autre. Comme nous l'avons vu, les QGraphicsItem sont placés sur la scène un par un, typiquement grâce à une boucle qui invoque la méthode `addItem`. Assez logiquement, lorsque la scène est dessinée sur une vue donnée, les derniers éléments ajoutés vont se dessiner *au-dessus* des éléments déjà présents sur la scène ; si les éléments sont opaques, ils vont occulter ceux qui sont en dessous d'eux, partiellement ou totalement. Toutefois, pour les besoins de l'application, le programmeur peut changer cet empilement par défaut en invoquant la méthode `setZValue(z)` sur le QGraphicsItem de son choix, où `z` est un nombre à virgule flottante positif ou négatif. En fait, même si on fait ici du dessin 2D avec coordonnées XY, chaque élément a une coordonnée Z cachée qui vaut zéro par défaut. Son rôle est simple à comprendre : un élément ayant une valeur Z donnée va toujours s'afficher au dessus de tout élément ayant un Z inférieur ; quand des éléments ont la même valeur Z, on se rabat sur l'ordre d'insertion dans la scène comme expliqué plus haut. Ainsi par exemple, quand on clique sur un élément partiellement occulté par d'autres, on peut *mettre en avant* cet élément en faisant `setZValue(zmax+1)` où `zmax` est le Z maximum de la scène (attribut à gérer par le programmeur).

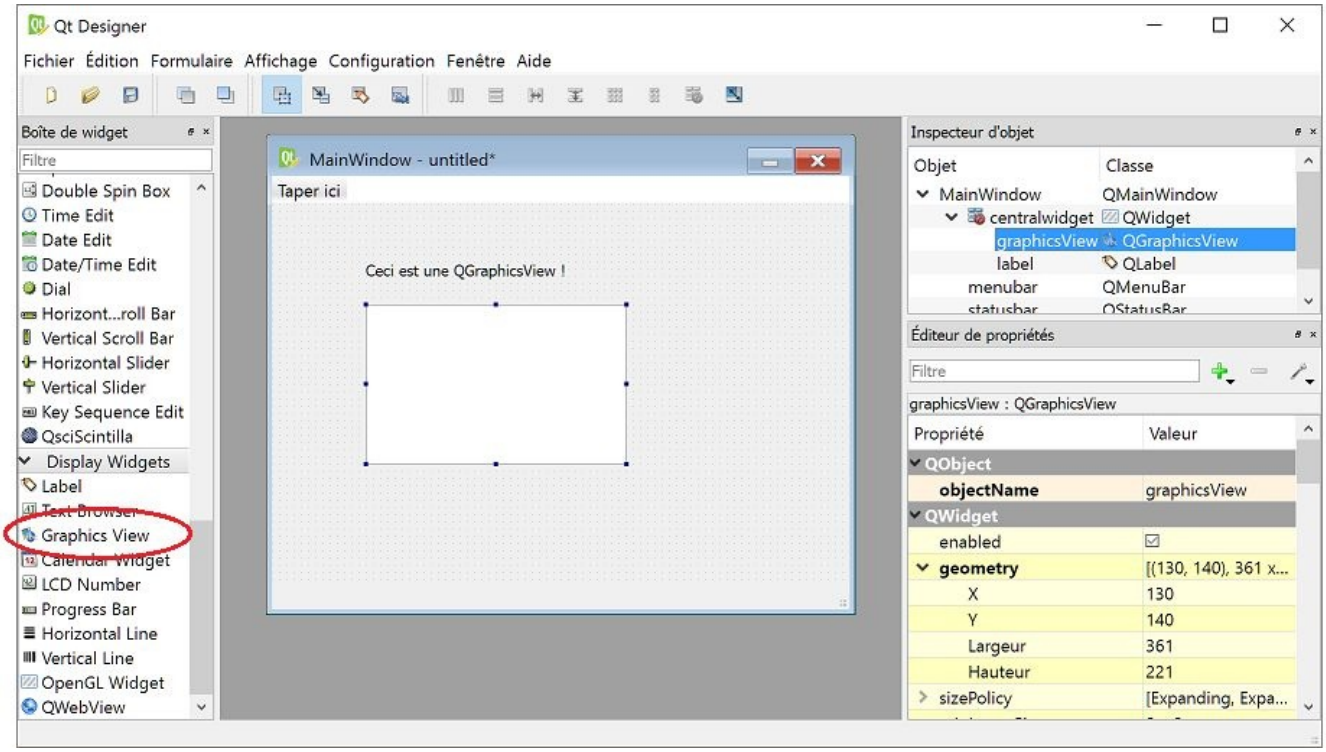
2.3. Classe QGraphicsView

La classe QGraphicsView a pour rôle d'afficher une scène donnée à l'écran. Plus précisément, on va indiquer à l'instance de QGraphcsView (généralement à sa création) une certaine scène, instance de QGraphicsScene. La vue a dès lors pour mandat d'afficher à l'écran tout ou partie de la scène dans une zone rectangulaire appelée *zone*

d'affichage (*viewport*) avec, au besoin, des barres de défilement ; ce concept vient en fait de la classe QAbstractScrollArea dont hérite QGraphicsView. Cet affichage met en jeu des transformations (notamment, pour gérer les zooms) ainsi que du *clipping*, pour n'afficher que ce qui est visible dans la zone d'affichage.

À l'inverse des classes que nous venons de voir, QGraphicsView ne nous est pas complètement étrangère puisqu'elle hérite de QWidget. C'est donc un vrai widget qu'on peut intégrer dans n'importe quelle fenêtre PyQt. Pour s'en convaincre, il suffit d'ouvrir Qt Designer : dans la Boîte de widget sur la gauche, on a, dans la catégorie Display Widgets, un Graphics View prêt à être déposé sur une fenêtre en cours de conception. On peut alors ajuster ce widget en utilisant notamment les techniques de disposition intelligente basées sur les positionneurs (layouts). Dans Qt Designer, ce widget est une instance de QGraphicsView vide, c'est-à-dire sans scène associée ; c'est compréhensible : la scène et ses éléments sont des abstractions, des *non-widgets*, qui sortent largement du cadre de Qt Designer. La création de la scène et l'association avec la (ou les) vue(s) se feront donc dans notre programme Python, après génération du code lié au fichier `.ui`.

Figure 10.5 : *QGraphicsView dans Qt Designer*



À côté des fonctions d'affichage, la classe QGraphicsView s'occupe aussi des événements utilisateur. C'est logique : en tant que widget visible à l'écran, elle est en première ligne pour recevoir les événements souris et clavier. La vue QGraphicsView

va alors propager ces événements à la scène et, au besoin, aux éléments qu'elle contient, en gérant les transformations de coordonnées entre vue et scène. Ceci a l'air anodin, mais retrouver rapidement un élément sur lequel on a cliqué n'est pas une tâche facile à programmer : il faut transformer les coordonnées (*x*,*y*) écran en coordonnées de la scène, trouver l'élément qui contient ce point[14] et, s'il y en a plusieurs superposés, garder celui qui a le plus haut indice Z. Tout ce travail est caché par le framework qui va simplement appeler la méthode `mousePressEvent` sur l'élément pointé (méthode qu'on peut bien sûr surcharger).

---

[13] On constate que les concepteurs de Qt ont quelque peu sacrifié la cohérence de leur nomenclature au bénéfice de la concision !

[14] Pour cela, la scène utilise par défaut une structure de données en arbre appelée *Binary Space Partitioning* (BSP). Elle permet de trouver rapidement le ou les élément(s) à une position (*x*,*y*) donnée.

3. Systèmes de coordonnées

Pour créer des graphiques 2D, on a bien sûr besoin d'un système de coordonnées qui permet de définir un point à partir d'un couple de nombres (*x*,*y*). Dans les systèmes graphiques classiques, on utilise un système de coordonnées écran où (*x*,*y*) sont des nombres entiers identifiant un pixel ; l'origine du repère de coordonnées peut être le coin supérieur gauche de l'écran ou, dans les systèmes à base de fenêtres, le coin supérieur gauche du widget sur lequel on dessine.

Qu'en est-il dans le framework Graphics View ? Comme déjà signalé, il y a un découplage fort entre la scène et les vues qui l'observent. Chacun de ces objets a en fait son propre système de coordonnées.

- *Coordonnées scène* — pour définir et positionner les éléments graphiques on utilise le système de coordonnées de la QGraphicsScene où (*x*,*y*) sont des nombres en virgule flottante (des `float` Python). L'origine et l'orientation du repère scène n'ont pas vraiment d'importance ; ce qui compte, ce sont les positions relatives des points les uns par rapport aux autres. Par ailleurs, les distances calculées dans ce repère n'ont pas de sens absolu : une différence d'abscisses valant 4 ne veut pas (forcément) dire 4 pixels.

- *Coordonnées vue* — les QGraphicsView sont des widgets à part entière ; à ce titre, ils ont un système de coordonnées écran classique, ayant le coin supérieur gauche de la zone d'affichage comme origine. Les coordonnées sont entières (des `int` Python) et ont pour unité le pixel.

Pour que la scène soit représentée sur une vue donnée, le framework va réaliser une *transformation* de coordonnées, avant de faire le rendu graphique à l'écran. Inversement, lorsque l'utilisateur clique sur la vue, le point (*x*,*y*) en coordonnées vue sera traduit automatiquement en coordonnées scène (*x*',*y*') en faisant la transformation inverse. Cette distinction entre les deux systèmes de coordonnées permet en fait de bien séparer les choses : la scène n'est pas du tout modifiée par les opérations de défilement ou de zoom demandées par l'utilisateur sur la vue ; cela permet notamment d'avoir plusieurs vues sur la même scène avec, typiquement, différents facteurs de zoom.

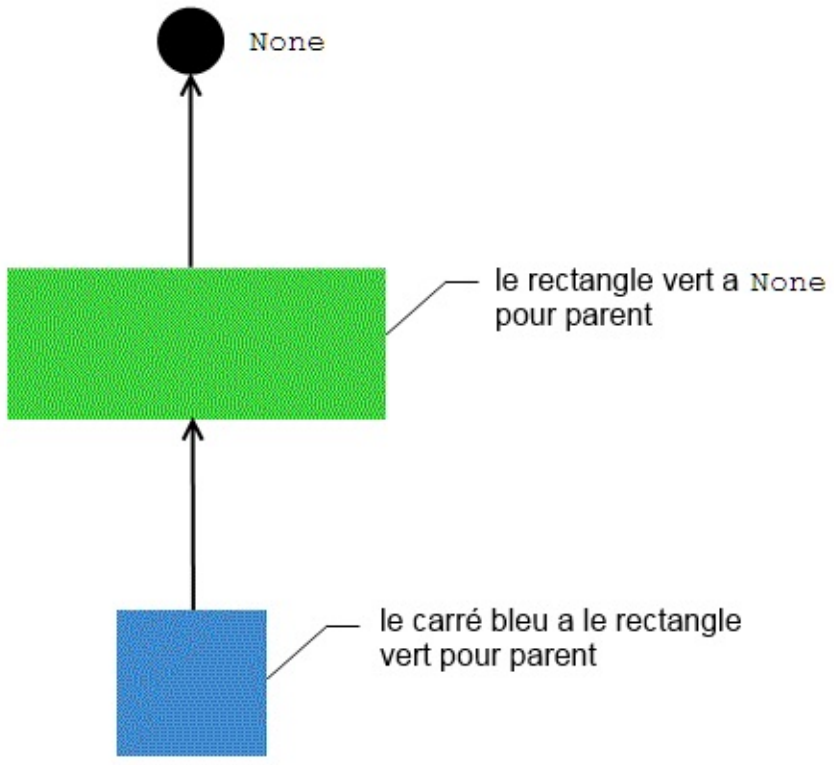
Pour bien comprendre le positionnement des éléments QGraphicsItem sur la scène, il reste un concept important à définir : celui de *coordonnées relatives*. Toute coordonnée utilisée au sein d'un élément donné s'exprime dans un repère propre à cet élément. Pour identifier ce repère, on utilise deux attributs définis par le programmeur sur tout

QGraphicsItem E :

- un *parent* : par défaut, il vaut `None`, ce qui signifie "pas de parent" ; le parent de E peut être un QGraphicsItem existant, si le programmeur définit ses éléments par composition ;
- une *position* : il s'agit de coordonnées (*x*,*y*) représentant l'origine du repère de E ; sa définition dépend de son attribut parent :
  - si E n'a pas de parent, alors (*x*,*y*) sont exprimées dans le repère de la scène,
  - si E a un parent, alors (*x*,*y*) son exprimées dans le repère de ce parent.

Prenons un exemple. On va placer sur une scène deux QGraphicsRectItem représentant un rectangle vert et un carré bleu. Pour la démonstration, l'élément carré bleu aura l'élément rectangle vert pour parent.

Figure 10.6 : *Les liens de parenté entre éléments*

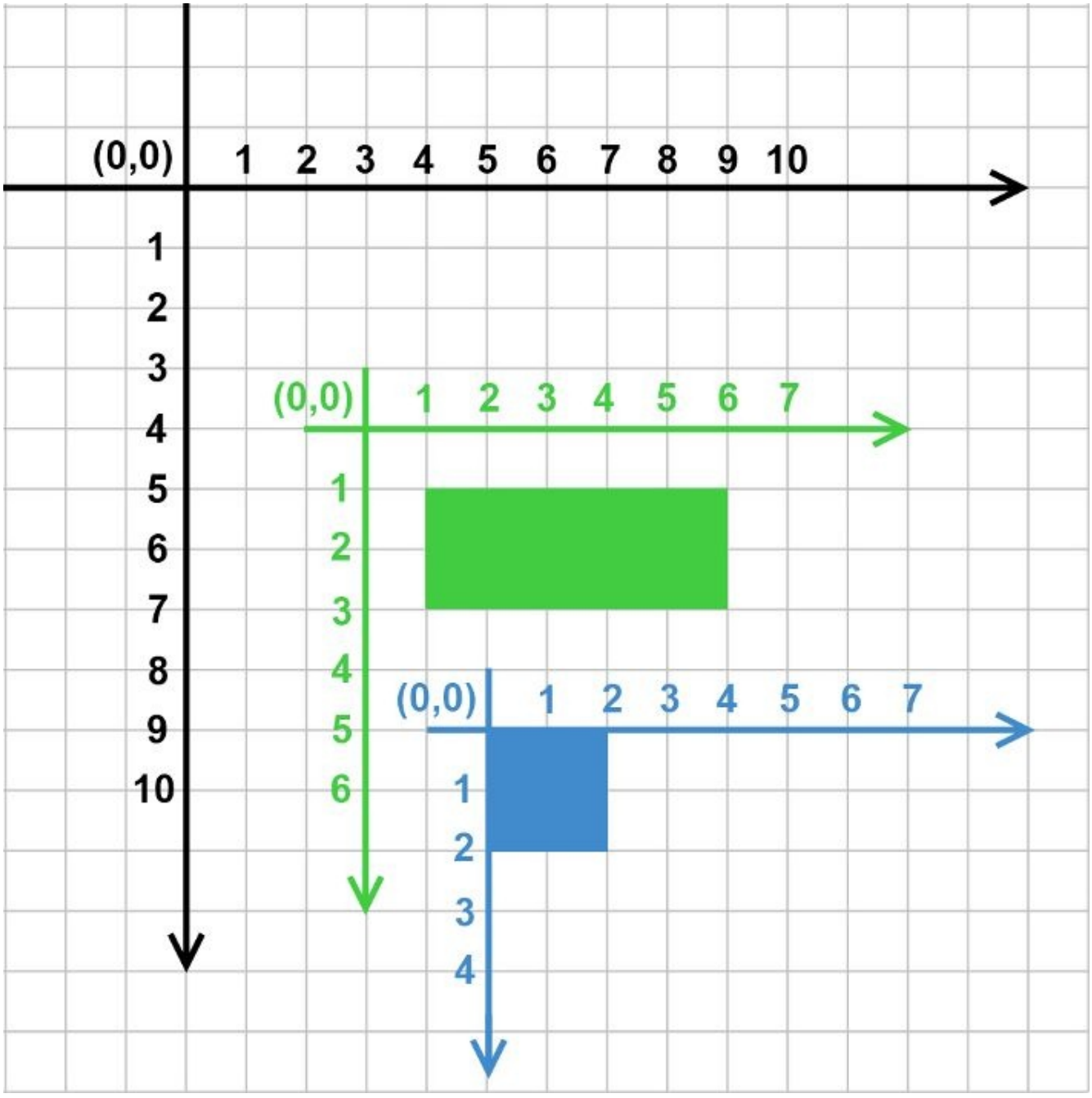


Commençons par le rectangle vert, qu'on va placer sur une scène (repère avec axes noirs sur la Figure 10.7) :

```
from PyQt5.QtWidgets import QGraphicsScene, QGraphicsRectItem

scene = QGraphicsScene()
rectangleVert = QGraphicsRectItem(1.0,1.0,5.0,2.0)
rectangleVert.setPos(3.0,4.0)
scene.addItem(rectangleVert)
```

Le rectangle vert a pour longueur 5 et pour largeur 2 (valeurs exprimées en coordonnées scène). Son coin supérieur gauche est positionné aux coordonnées (1,1) relatives à son propre repère. Quelle est l'origine de ce repère ? Cette origine est fixée par la méthode `setPos` : comme le rectangle vert n'a pas de parent, elle est aux coordonnées (3,4) de la scène (axes verts sur la Figure 10.7). Résultat des courses : le coin supérieur gauche du rectangle vert est aux coordonnées (4,5) de la scène.

Figure 10.7 : *Les éléments positionnés les uns par rapport aux autres*



Passons à présent au carré bleu :

```
carreBleu = QGraphicsRectItem(0.0,0.0,2.0,2.0,parent=rectangleVert)
carreBleu.setPos(2.0,5.0)
```

Le carré bleu est défini comme un rectangle ayant une taille de 2 unités pour longueur et pour largeur. Son coin supérieur gauche est positionné aux coordonnées (0,0) relatives à son propre repère (axes bleus sur la Figure 10.7). L'origine de ce repère est fixée par la méthode `setPos` aux coordonnées (2,5) relatives au repère de son parent, le

rectangle vert. Résultat des courses : le coin supérieur gauche du carré bleu est aux coordonnées (5,9) de la scène.

Pour résumer ces règles en deux phrases :

- un élément est toujours positionné dans le repère de son parent ou, s'il n'a pas de parent, directement dans le repère de la scène (méthode `setPos`) ;
- les coordonnées définissant l'élément sont relatives à la position de cet élément (voir constructeurs tels que `QGraphicsRectItem`).

Bien sûr, tout cela peut paraître compliqué de prime abord. Toutefois, à l'usage, ces quelques principes apparaissent en définitive assez naturels quand on pratique la composition d'éléments : il est plus aisé de positionner un boulon du vaisseau spatial dans le repère de ce vaisseau plutôt que dans le repère absolu de la scène ! Par ailleurs, tout ce qui précède n'implique pas la nécessité de se lancer dans des calculs méticuleux : en cas de besoin, les QGraphicItem proposent des méthodes de conversion d'un repère à l'autre ; par exemple, les coordonnées du coin supérieur gauche du carré bleu peuvent être vérifiées par la méthode `mapToScene` :

```
print(carreBleu.mapToScene(0.0,0.0))
# affiche: PyQt5.QtCore.QPointF(5.0, 9.0)
```

Inversement, on peut déterminer comment s'expriment des coordonnées scène en coordonnées élément par appel à `mapFromScene` :

```
print(carreBleu.mapFromScene(0.0,0.0))
# affiche: PyQt5.QtCore.QPointF(-5.0, -9.0)
```

4. Pinceaux et brosses

Beaucoup d'éléments QGraphicsItem définissent des segments de droite, des courbes, des surfaces à dessiner et à colorier. Le framework factorise toutes ces caractéristiques dans deux objets : le pinceau et la brosse.

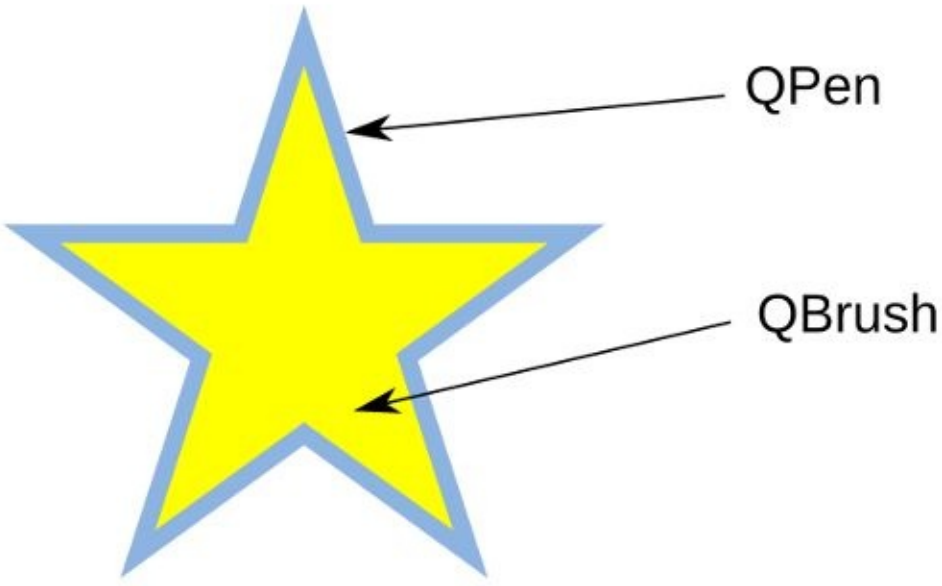
Pinceau

Un *pinceau* (classe QPen) définit toutes les caractéristiques servant à tracer des traits ou des courbes : couleur, épaisseur du trait, type continu/pointillé, etc.

Brosse

Une *brosse* (classe QBrush) définit toutes les caractéristiques servant à remplir une surface : couleur, dégradé, texture, type plein/hachuré, etc.

Figure 10.8 : *Pinceau et brosse*



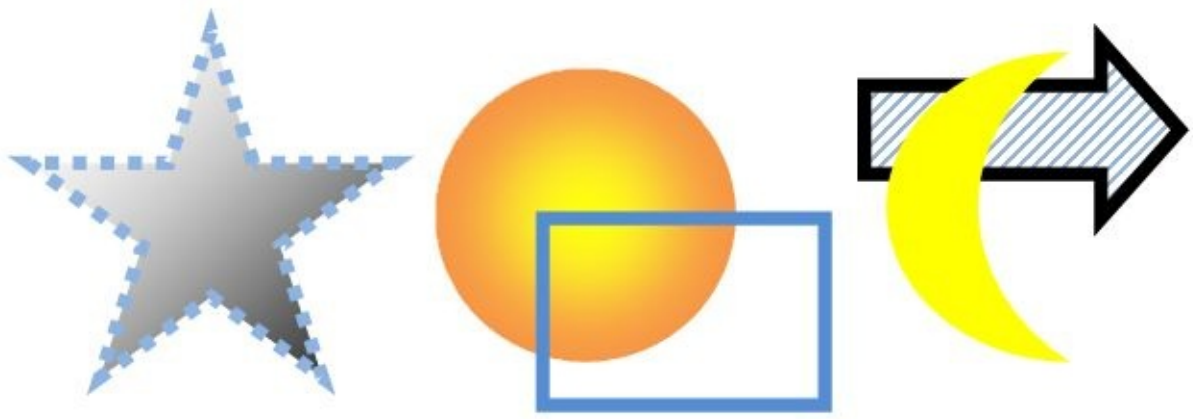
Pour être précis, les attributs pinceau et brosse sont définis au niveau de QAbstractGraphicsShapeItem, classe fille de QGraphicsItem. En effet, cette dernière est trop générale puisqu'elle recouvre tous les éléments graphiques, y compris ceux qui n'ont pas besoin de pinceau ou de brosse. Parmi les classes qui héritent de QAbstractGraphicsShapeItem, on retrouve QGraphicsEllipseItem, QGraphicsPathItem,

QGraphicsPolygonItem, QGraphicsRectItem et QGraphicsSimpleTextItem. Pour toutes les instances de ces classes, on peut donc définir un pinceau [méthode `setPen(…)`] et une brosse [méthode `setBrush(…)`]. La classe QGraphicsLineItem est un peu particulière, car elle n'a besoin que d'un pinceau : dès lors, pour ne pas se charger d'une brosse inutile, QGraphicsLineItem n'hérite *pas* de QAbstractGraphicsShapeItem, mais bien de QGraphicsItem.

***Note** > Les classes QPen et QBrush existent en fait depuis longtemps dans Qt, bien avant l'apparition du framework Graphics View. Elles sont utilisées par QPainter, la classe de base sur laquelle s'appuie le framework. La différence apparaît quant à l'utilisation de ces deux classes : dans QPainter, le pinceau et la brosse sont à définir avant toute opération de dessin ; dans le framework, ces deux objets sont assignés individuellement à chaque élément.*

Les attributs des pinceaux et brosses permettent de définir très précisément la manière dont vont être dessinés les éléments. La Figure 10.9 donne quelques exemples de combinaison d'attributs.

***Figure 10.9 :** Attributs des pinceaux et brosses*



Pour le pinceau, les attributs les plus utilisés sont la couleur (noire par défaut) et l'épaisseur (1 par défaut, en unité scène). Pour des cas spécifiques, on peut définir le style (trait continu par défaut) et même la manière dont les traits se terminent (*cap style*) et se joignent entre eux (*join style*). Certaines valeurs d'attribut permettent de définir des pinceaux spéciaux :

- *pinceau cosmétique* — on appelle ainsi un pinceau ayant une épaisseur définie avec la valeur zéro. Quel que soit le niveau de zoom, il est représenté sur la vue avec une épaisseur de 1 pixel exactement. Les pinceaux cosmétiques sont

particulièrement utiles pour les vues qui permettent de grands niveaux de zoom, telles les applications cartographiques.

- *pinceau invisible* — dans certains graphiques montrant des formes pleines (ellipses, polygones, caractères, etc.), la brosse suffit à distinguer les éléments dessinés ; on peut choisir dans ce cas de ne pas tracer les contours. À cette fin, on utilise un pinceau ayant le style `Qt.NoPen`, qui est donc en quelque sorte un pinceau invisible. À la Figure 10.9, le disque et le croissant ont un tel pinceau. Pour préciser ce qui a déjà été dit, les classes filles de QAbstractGraphicsShapeItem ont par défaut des pinceaux ayant le style `Qt.SolidLine` (trait continu), sauf la classe QGraphicsSimpleTextItem qui a un pinceau invisible par défaut ; on peut le rendre visible pour marquer le contour des caractères, ce qui peut être très joli, mais coûteux en temps d'exécution.

Pour la brosse, les attributs les plus utilisés sont le style (transparent par défaut) et la couleur. Le style transparent (`Qt.NoBrush`) pour un élément donné suppose normalement qu'un pinceau visible ait été assigné à cet élément (voir rectangle sur la Figure 10.9). Le style `Qt.SolidPattern` est l'autre style le plus utilisé (le croissant sur l'image). D'autres styles permettent des effets plus élaborés tels une texture (par exemple, `Qt.BDiagPattern` pour la flèche) ou un dégradé (par exemple, `Qt.LinearGradienPattern` pour l'étoile et `Qt.RadialGradientPattern` pour le disque). Pour définir un dégradé, il faut fournir une instance d'une classe fille de QGradient : QLinearGradient, QRadialGradient ou QConicalGradient.

Comment définir une couleur ?

Pour définir les couleurs des objets qui en ont besoin (pinceaux, brosses, dégradés, etc), Qt définit une classe QColor. Une instance de QColor possède des attributs définissant très précisément une couleur, y compris une composante alpha servant à donner un effet de transparence. Il existe de multiples méthodes pour créer une instance de QColor. En voici quelques-unes parmi les plus simples :

- couleurs énumérées : il y a vingt couleurs accessibles directement dans le module `PyQt5.QtCore.Qt` : `white`, `black`, `gray`, `red`, `green`, `blue`, etc. ;

- couleurs nommées : pour avoir un plus grand choix, on peut aussi spécifier une chaîne de caractères contenant un nom de couleur SVG 1.0, tels que `"lightskyblue"` ou `"greenyellow"` ;

- composantes RGB : on définit la couleur selon ses composantes

rouge/verte/bleue, qui sont des valeurs qui vont de 0 à 255. Par exemple, un bleu foncé se définit avec `QColor(0,0,139);`

- composantes RGB hexa : on peut aussi définir les composantes RGB en chiffres hexadécimaux dans une chaîne de caractères ayant le format "#RRGGBB". Par exemple, le bleu foncé se définit avec `QColor("#00008b").`

D'autres variantes permettent notamment de définir les couleurs en composantes HSV (*Hue/Saturation/Value*) ou CMYK (*Cyan/Magenta/Yellow/Black*).

11

Première application avec une vue graphique

Niveau : débutant

Objectifs : se familiariser avec les classes de base du framework Graphics View de PyQt

Prérequis : Introduction au framework Graphics View de Qt

Dans ce chapitre, nous allons montrer comment les concepts introduits précédemment se traduisent concrètement dans une application PyQt. Nous nous focaliserons essentiellement sur la définition des éléments sans (trop) nous occuper des interactions avec l'utilisateur. Ensuite, on verra comment créer plusieurs vues sur la même scène, avec différentes transformations. Enfin, on montrera comment créer une fenêtre principale intégrant une vue graphique via Qt Designer et comment on peut interagir avec cette vue via des widgets classiques.

Note > *Le code source final de ce chapitre est disponible dans le projet* `premiere_app`  .

1. Tous en scène !

Ce qui suit nécessite juste de créer un petit fichier Python, par exemple `test_scene.py`.

1.1. Création de la scène

Commençons par créer une scène minimaliste avec un rectangle gris.

```
#-*- coding: latin_1 -*-
# test_scene.py

from PyQt5.QtCore import Qt
from PyQt5.QtGui import QBrush
```

❶ from PyQt5.QtWidgets import QApplication, QGraphicsScene, QGraphicsView, \ QGraphicsRectItem import sys app = QApplication(sys.argv) # definition de la scène scene = QGraphicsScene() ❷ rectGris = QGraphicsRectItem(0.,0.,200.,150.) ❸ rectGris.setBrush(QBrush(Qt.lightGray)) ❹ scene.addItem(rectGris) ❺ # definition de la vue vue = QGraphicsView(scene) ❻ vue.resize(800,600) ❼ vue.fitInView(rectGris,Qt.KeepAspectRatio) ❽ vue.show() ❾ sys.exit(app.exec_())

Voici ce qu'on fait dans ce court programme.

❶ Pour les imports, on détaille ici chaque classe utilisée pour montrer le module auquel elle appartient. Pour la facilité, on pourrait utiliser `from PyQt5.… import *`, sachant que les noms des classes des différents modules ne rentrent pas en collision.

❷ On crée une scène, initialement vide. À noter que la scène a une couleur de fond qui, par défaut, est le blanc.

❸ Le rectangle QGraphicsRectItem est défini en coordonnées locales ; comme on ne définit pas une position via `setPos`, le coin supérieur gauche du rectangle est exactement aux coordonnées (0,0) de la scène. Rappelons que les dimension

données 400 × 300 sont exprimées dans le repère de la scène et ne disent encore rien sur les dimensions en pixels du rectangle affiché dans la vue.

❹ Pour colorier le rectangle, on lui associe une brosse, instance de QBrush, en lui spécifiant une couleur QColor. Une brosse est associée à chaque élément : elle définit la manière de colorier l'élément. Sans cette instruction, on aurait une brosse transparente, qui laisse donc voir le fond blanc de la scène.

❺ On ajoute l'élément rectangle à la scène. Tant qu'on n'a pas fait cela, il n'existe que "dans les limbes" et ne peut pas être visualisé !

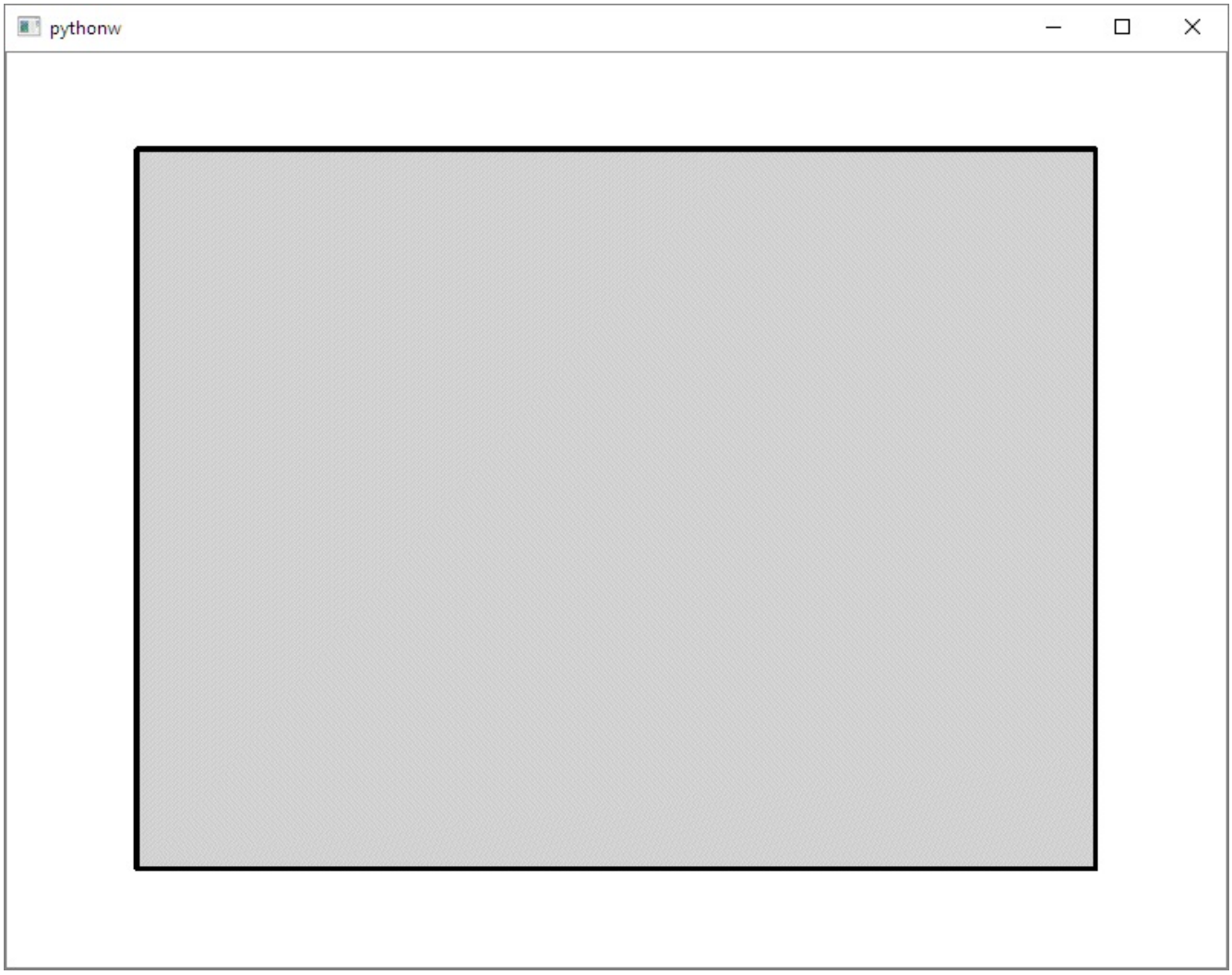
❻ On crée une vue sur la scène, en donnant l'instance de QGraphicsScene au constructeur QGraphicsView. On a à présent un widget qu'on peut visualiser à l'écran.

❼ On dimensionne la vue à 800 × 600. Comme la vue est un widget, ce sont bien des nombres entiers ici, exprimant des pixels.

❽ La vue ne sait pas initialement quelle partie de la scène (zone d'affichage) on veut observer. La méthode `fitInView` ajuste la vue de sorte à ce que le rectangle apparaisse en entier. Cette méthode utilitaire, qui se charge de calculs laborieux mettant en correspondance le système de coordonnées de la scène et celui de la vue, est très utile à l'initialisation.

❾ Pour la simplicité de l'exemple, on n'intégre pas la vue dans une fenêtre ; on la rend visible sans autre fioriture (voir Figure 11.1).

Figure 11.1 : *Exemple de scène — 1*



Notons que QGraphicsScene fournit des méthodes de commodité pour créer facilement des éléments standards. Ainsi, la création du rectangle, sa coloration et son ajout à la scène peuvent être réalisés en un seul appel :

```
rectGris = scene.addRect(0.,0.,200.,150.,brush=QBrush(Qt.lightGray))
```

1.2. Texte

À présent que nous nous sommes familiarisés avec les classes de base du framework, nous allons enrichir la scène avec du texte et un élément utilisant la composition.

Pour pouvoir définir les nouveaux éléments, on a besoin des `import` suivants :

```
from PyQt5.QtGui import QBrush, QPen, QColor, QPainter
```

```
from PyQt5.QtWidgets import QApplication, QGraphicsScene,
QGraphicsView, \
     QGraphicsItem, QGraphicsRectItem, QGraphicsTextItem,
QGraphicsEllipseItem
```

Commençons par ajouter un texte en bleu sur la scène, juste après la création du rectangle gris (donc, au-dessus !) :

```
texte = QGraphicsTextItem("Tous en scène !")
dy = rectGris.sceneBoundingRect().height() -
texte.sceneBoundingRect().height()
```

❶ texte.setPos(rectGris.x(),rectGris.y()+dy) ❷ texte.setDefaultTextColor(QColor('blue')) scene.addItem(texte)

Pour rendre l'exemple instructif, on a choisi de placer ce texte juste en bas du rectangle (voir Figure 11.2). Pour faire cela de manière générique, quelles que soient les dimensions du rectangle et celles du texte, il y a un peu de calcul de coordonnées à faire. Comme on n'a pas déclaré de parent pour le texte, il faut le positionner directement dans le repère de la scène. Les coordonnées scène du coin supérieur du rectangle sont connues : `rectGris.pos()`, ce qui revient à `(rectGris.x(),rectGris.y())`. L'appel à `setPos` part de ce point et le décale de `dy` vers le bas pour l'aligner avec le bas du rectangle ❷. La valeur de `dy` est calculée par soustraction de la hauteur du rectangle avec la hauteur du texte ❶ ; pour obtenir la hauteur de chaque élément QGraphicsItem, on doit procéder en deux temps : on invoque la méthode `sceneBoundingRect()` pour obtenir un rectangle (instance de QRectF), on invoque ensuite `height()` pour en obtenir la hauteur.

1.3. Composition

Plaçons à présent un smiley, formé d'un cercle jaune et de deux ellipses pour les yeux (on ne dessinera pas la bouche, qui est un élément un peu plus compliqué).

```
d = 48. # diametre smiley
```

❶ ox = 4. # largeur oeil oy = 6. # hauteur oeil smiley = QGraphicsEllipseItem(-d/2,-d/2,d,d) ❷ smiley.setBrush(QBrush(Qt.yellow)) yeux = [QGraphicsEllipseItem(-ox/2.,-oy/2.,ox,oy,parent=smiley) \ ❸ for _ in range(2)] yeux[0].setPos(-d/6,-d/8) ❹ yeux[1].setPos(+d/6,-d/8) brush = QBrush(Qt.black) for oeil in yeux: ❺ oeil.setBrush(brush) smiley.setPos(rectGris.mapToScene(rectGris.rect().center())) ❻ scene.addItem(smiley)

❶ Les dimensions des éléments sont définies dans des variables pour pouvoir facilement les ajuster.

❷ Le smiley est défini par un QGraphicsEllipseItem, de dimension `d × d`, un cercle donc. Attention, les deux premiers arguments n'indiquent *pas* le centre du cercle mais le coin supérieur gauche du carré qui l'entoure. Le fait d'indiquer les coordonnées `(-d/2,-d/2)` est un artifice classique pour faire en sorte que le centre du cercle soit aux coordonnées (0,0) de cet élément. Ceci va faciliter le positionnement symétrique des deux éléments yeux.

❸ Les yeux sont définis par deux QGraphicsEllipseItem, qui ont pour parent l'élément `smiley` défini. Comme ils ont tous deux des dimensions égales, on les crée — par facilité — via une liste en compréhension Python qui fait exactement la même création deux fois. Leur dimension est `ox × oy`, avec le centre en (0,0) en utilisant l'artifice vu au point ❷. À ce stade, les deux yeux sont superposés au centre du cercle `smiley` (ce qui le ferait donc apparaître comme un cyclope !).

❹ On positionne l'œil gauche (`yeux[0]`) et l'œil droit (`yeux[1]`) dans le repère de l'élément parent `smiley`. En changeant simplement le signe des coordonnées X, on positionne les deux yeux parfaitement symétriquement, puisque le centre de ce repère est le centre de `smiley`.

❺ On colorie les yeux à l'aide d'une brosse noire. Comme les boucles en Python sont très simples à écrire, il ne faut pas se priver de factoriser les propriétés communes à plusieurs éléments.

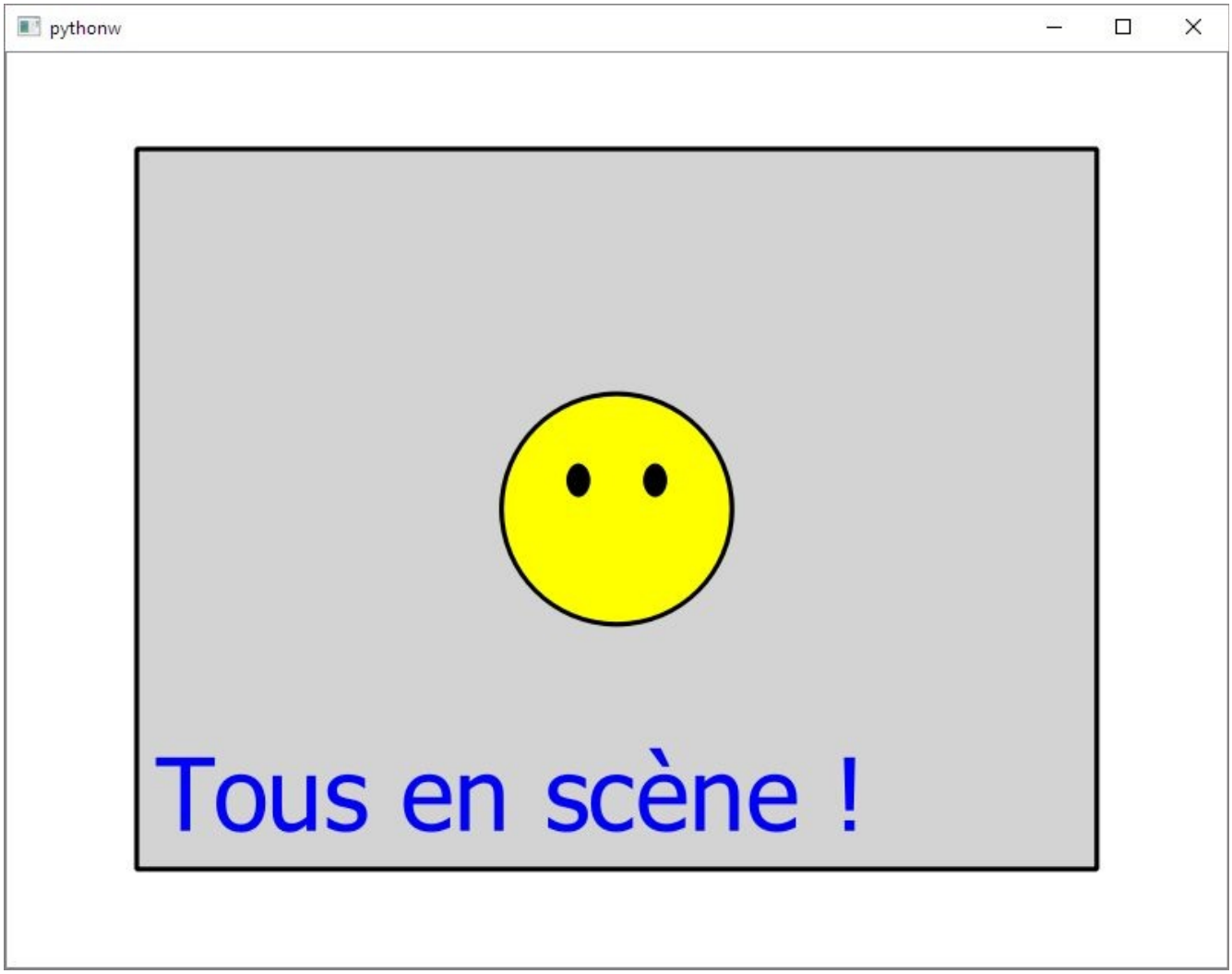
❻ On positionne le centre de `smiley` au centre du rectangle ; on utilise `mapToScene` pour obtenir les coordonnées scène à partir des coordonnées relatives au rectangle. Notons que cette instruction déplace automatiquement les deux yeux, car éléments fils de `smiley`. Le smiley est donc bien un élément composé qu'on peut manipuler comme un tout sans se soucier de ses constituants.

Pour obtenir un affichage plus agréable, il est conseillé d'activer l'*anticrénelage* sur la vue ; ceci a pour effet de lisser les traits, en atténuant l'effet d'escalier.

```
vue.setRenderHints(QPainter.Antialiasing)
```

Attention *> L'anticrénelage peut avoir un impact négatif sur la performance d'affichage si le nombre d'éléments à afficher est important. Pour une application sérieuse, il est recommandé de rendre cette option configurable — comme le proposent d'ailleurs beaucoup de jeux !*

Figure 11.2 : *Exemple de scène — 2*



1.4. Transformations

Jusqu'ici, nous n'avons pas encore bien vu l'intérêt du framework. C'est normal : nous avons une scène statique, sans interaction possible avec l'utilisateur ; comme signalé dans l'introduction, ce type de graphique aurait très bien pu être réalisé avec la

technique plus classique de QPainter.

Imaginons à présent que vous vouliez :

- agrandir le smiley de 50 % ;
- lui appliquer une rotation de 20° dans le sens des aiguilles d'une montre ;
- le déplacer avec le souris.

Dans une approche classique, vous devriez programmer des calculs de coordonnées assez laborieux, ainsi qu'une gestion sans fautes des événements souris (enfoncer un bouton, déplacer, lâcher le bouton). Avec le framework Graphics View, ces fonctions sont programmées en exactement trois instructions qu'on comprend sans explications !

```
smiley.setScale(1.5)
smiley.setRotation(20.)
smiley.setFlag(QGraphicsItem.ItemIsMovable)
```

À l'exécution, on voit que les deux transformations ont bien été opérées ; de plus, on peut déplacer la figure par glisser-déposer avec la souris.

Figure 11.3 : *Exemple de scène — 3*

On constate encore une fois l'intérêt d'avoir utilisé la composition pour les éléments du smiley : tout se positionne exactement là où l'on s'y attend, comme si les constituants étaient solidement attachés.

Le smiley ayant été créé en dernier, il se place au-dessus des autres éléments. Si on veut le placer *sous* le texte, mais toujours au-dessus du rectangle, on pourrait changer l'ordre de création dans le programme. Cette approche ne convient pas toutefois si on veut changer cet ordre dynamiquement. Comme expliqué à la Section 2.2, Classe QGraphicsItem, on peut faire cela en jouant sur l'indice Z. Actuellement, tous les éléments placés ont l'indice Z à 0. Pour faire passer le texte au-dessus du reste, il suffit de lui assigner une valeur supérieure, par exemple :

```
texte.setZValue(1)
```

La Figure 11.4 illustre ce qui se passe alors, quand on déplace le smiley.

Figure 11.4 : *Exemple de scène — 4*

2. Vues multiples

Maintenant que la scène est en place, il est aisé de créer une seconde vue sur cette scène. Pour montrer quelque chose d'intéressant, on ne va toutefois pas créer un clone de la première vue : la nouvelle vue aura pour dimension écran 300 × 400 ; de plus,

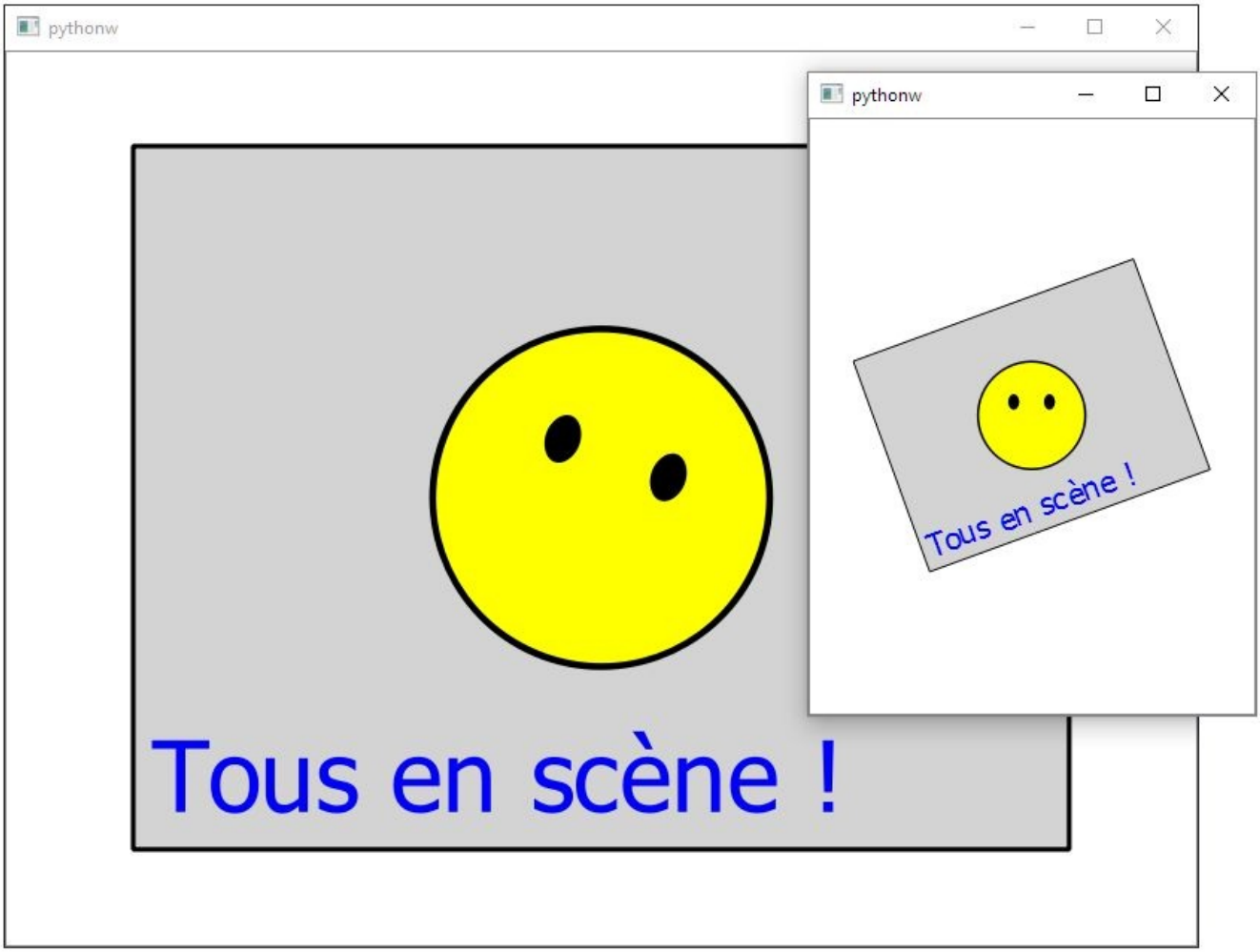
- on n'appellera *pas* `vue.fitInView` ; à l'inverse de la première vue, il n'y aura donc pas de zoom : une unité scène aura la largeur d'un pixel à l'écran ;
- on effectuera une rotation de 20° dans le sens inverse des aiguilles d'une montre ; cela revient à tourner la caméra sans rien changer à la scène.

Voici les lignes à ajouter à notre programme pour réaliser tout cela :

```
vue2 = QGraphicsView(scene)
vue2.setRenderHints(QPainter.Antialiasing)
vue2.resize(300,400)
vue2.rotate(-20)
vue2.show()
```

À l'exécution, on aura deux fenêtres vue montrant la même scène sous un angle et un zoom différent.

Figure 11.5 : *Exemple de scène — 5 (deux vues)*



On constate que la nouvelle vue a remis le smiley droit : c'est normal, la rotation de la vue (−20°) compense exactement la rotation du smiley sur la scène (+20°). Si on s'amuse à déplacer le smiley sur la première vue, on voit que la seconde vue est mise à jour automatiquement et immédiatement. Le déplacement du smiley peut aussi être opéré sur la deuxième vue : la première vue se redessine similairement. On voit donc bien à l'œuvre le principe de séparation modèle-vue : on met à jour la scène par une manipulation sur une vue et toutes les vues se rafraîchissent automatiquement… sans rien avoir à programmer !

Pour continuer la démonstration, on va créer à la volée quatre vues, qui font chacune une transformation aléatoire de la scène. À cette fin, on va combiner la méthode `rotate(angle)` déjà vue à la nouvelle méthode `scale(fx,fy)` : cette méthode fait un agrandissement d'un facteur `fx` horizontal et d'un facteur `fy` vertical. Selon que le facteur donné est plus grand ou plus petit que 1, on fait respectivement un agrandissement ou un rétrécissement. Cette fonction est bien entendu très utile pour réaliser des zooms, pour lesquels on choisira `fx = fy` ; si cette égalité n'est pas

respectée, on introduit des déformations : on va le constater dans l'exemple qui suit (techniquement, on dit qu'on ne conserve pas le *ratio* ou, en anglais, l'*aspect ratio*).

Voici les quelques lignes à ajouter.

```
from random import random
…
vuesAux = []
for _ in range(4):
    vueAux = QGraphicsView(scene)
    vuesAux.append(vueAux)
    vueAux.setRenderHints(QPainter.Antialiasing)
    vueAux.resize(400,300)
    vueAux.rotate(360*random())
    vueAux.scale(4*random(),4*random())
    vueAux.show()
```

La fonction Python `random` renvoie un nombre aléatoire entre 0 et 1. En suivant les multiplications, on voit que, pour chaque vue, on fait une rotation aléatoire entre 0° et 360° et un agrandissement entre 0 et 4, indépendamment dans les deux axes.

Voici ce que cela donne :

Figure 11.6 : *Exemple de scène — 6 (cinq vues)*

Comme on a fait des facteurs d'agrandissement aléatoires indépendants selon les axes, on obtient des effets de déformation notables (pour s'en affranchir, il suffit de placer la valeur aléatoire dans une variable `f` et appeler `vueAux.scale(f,f)`). On constate par ailleurs que des barres de défilement ont été placées automatiquement sur les vues qui ne peuvent montrer la scène en entier. En déplaçant le smiley sur une des vues, on constate à nouveau la synchronisation entre toutes les vues. Sur la base des lignes de code qui précèdent, on peut aisément pousser le framework plus loin en augmentant encore le nombre de vues à créer.

Note *> Dans la boucle `for`, on range les vues dans une liste `vuesAux` uniquement pour éviter qu'elles ne soient détruites à chaque itération. Sans cela, l'assignation de la variable `vueAux` détruit la vue précédemment assignée, car cette dernière n'est plus référencée (c'est comme cela que Python récupère la mémoire qui n'est plus utilisée). Concrètement, sans l'instruction `vuesAux.append(vueAux)`, le programme va s'exécuter sans erreur mais… on n'aura que la dernière vue créée dans la boucle.*

Bien évidemment, tout ce qui précède a uniquement vocation de démonstration : la plupart des applications n'utilisent en fait qu'une vue ou deux, font des zooms sans déformation et se passent de rotation. L'exemple montre que le principe de séparation scène et vue fonctionne parfaitement, sans causer de ralentissement. Bien sûr, l'exemple donné ici ne montre qu'une scène minimale avec juste cinq éléments ; au chapitre MosaiQ : une démo technologique ! , on verra qu'une scène peut contenir sans souffrir des dizaines de milliers d'éléments. Rappelons qu'on programme bien ici en Python, langage interprété qui a parfois une réputation de lenteur — un point qu'il faudrait grandement nuancer par ailleurs ! En fait, dans les applications PyQt, Python fait en général très peu de choses : c'est un chef de chantier qui sous-traite les opérations graphiques lourdes à des outils super-optimisés écrits en C++, voire à la carte graphique.

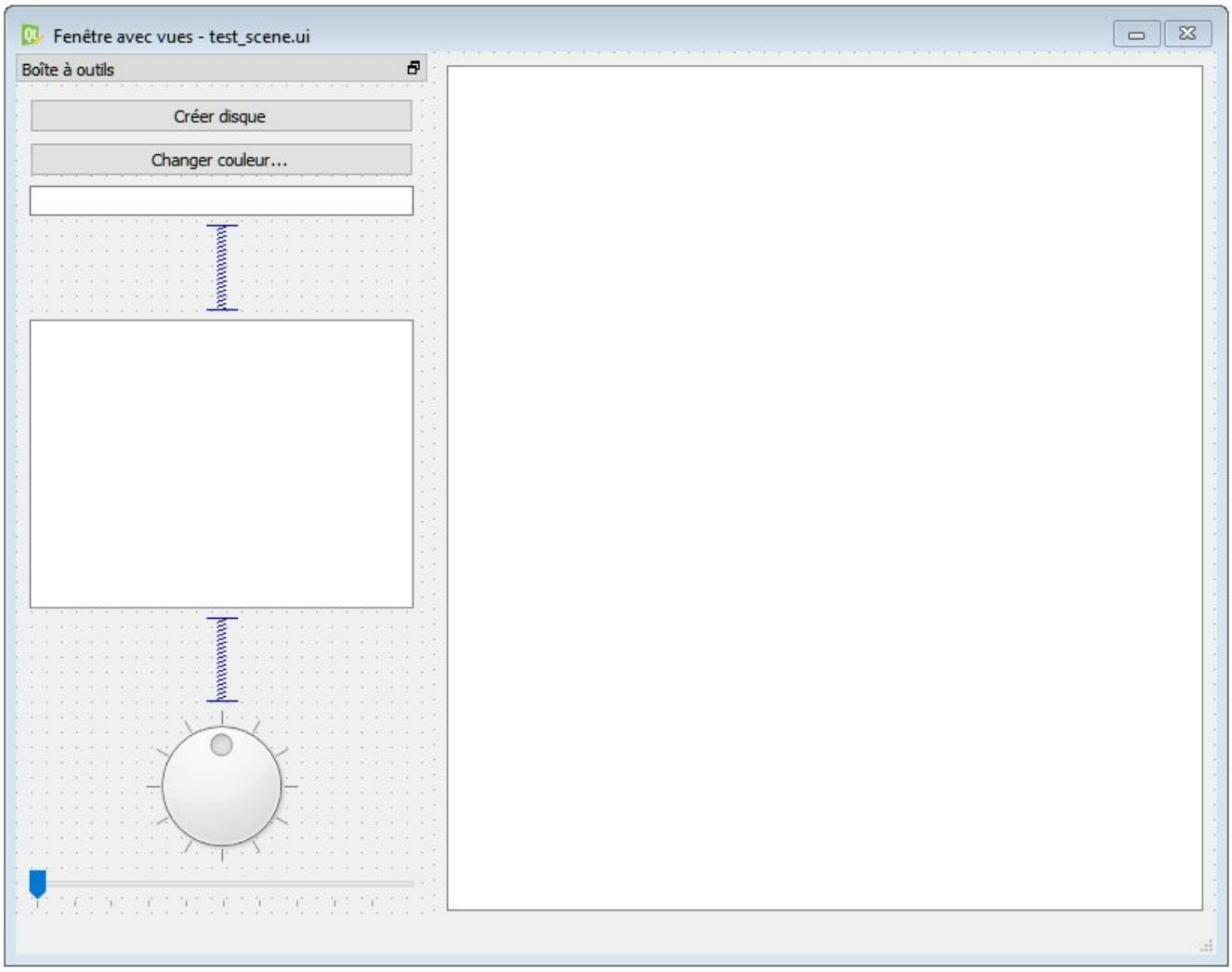
3. Intégration avec les widgets

Jusqu'ici, pour aller droit au but, on a affiché les vues instances de QGraphicsView comme des fenêtres de premier niveau. Voyons à présent comment faire les choses plus sérieusement en intégrant les vues dans une fenêtre principale et en les faisant collaborer avec les widgets qu'on connaît bien.

Pour commencer, on va créer une fenêtre principale dans Qt Designer. Dans ce qui suit, on peut bien sûr créer un projet eric6. Notre feuille s'appellera `test_scene.ui` et définira une fenêtre principale (QMainWindow) qui se présentera comme sur la Figure 11.7.

Note *> Pour plus d'infos sur les manipulations effectuées dans ce paragraphe, référez-vous au chapitre Développer avec Qt Designer.*

Figure 11.7 : *Fenêtre avec vues dans Qt Designer*



Voici les détails concernant des widgets qui doivent être disposés sur la fenêtre. La partie de droite est un Graphics View (en fait, une instance de QGraphicsView), qu'on nommera `vuePrincipale` (objectName). La partie de gauche est un Dock Widget, un *conteneur* qui a la propriété de pouvoir être déplacé ou même détaché par l'utilisateur. Ce sera notre boîte à outils dans laquelle on va placer différents widgets destinés à interagir avec la vue ou la scène. On assigne les propriétés suivantes à ce Dock Widget :

- *windowTitle* : Boîte à outils ;
- *features* : tout décocher sauf *DockWidgetMovable*, *DockWidgetFloatable* ;
- *allowedAreas* : tout décocher sauf *LeftDockWidgetArea* et *RightDockWidgetArea*.

Une fois qu'on a ces deux éléments principaux, on peut les aligner horizontalement pour qu'ils occupent toute la place disponible sur la fenêtre principale : on sélectionne la

fenêtre principale et on actionne Mettre en page horizontalement, via le menu contextuel ou la barre d'outils.

Ensuite, on place différents widgets dans le Dock Widget, de haut en bas (on fera l'alignement précis plus tard) :

- un bouton (Push Button) à nommer `pushButtonCreerDisque` (objectName) et à libeller "Créer disque" (text) ; il permettra d'ajouter un disque blanc à la scène (QGraphicsEllipseItem) ;
- un bouton (Push Button) à nommer `pushButtonChangerCouleur` (objectName) et à libeller "Changer couleur…" (text) ; il servira à changer la couleur des éléments sélectionnés ;
- un champ textuel (Line Edit) à nommer `lineEditTexte` (objectName) ; il permettra de changer le texte affiché dans le QGraphicsTextItem ;
- un espaceur vertical (Vertical Spacer) ;
- une vue graphique (Graphics View) à nommer `vueGlobale` (objectName) ; elle affichera une vue globale de la scène, sans être affectée par les zooms et rotations opérés sur la vue principale ;
- un espaceur vertical (Vertical Spacer) ;
- une molette (Dial) à nommer `dialRotation` (objectName) ; son but sera de faire tourner la vue principale ; on lui assignera les propriétés suivantes :
  - minimum : −180 (angle de rotation minimal) ;
  - maximum : +180 (angle de rotation maximal) ;
  - wrapping : coché (on peut faire le tour complet, en passant de −180° à +180°) ;
  - notchTarget : 30.0 (graduation tous les 30°) ;
  - notchesVisible : coché (affichage des graduations) ;
- un curseur (Horizontal Slider) à nommer `horizontalSliderZoom` (objectName) ; son but sera de fixer le facteur de zoom sur la vue principale ; on lui assignera les propriétés suivantes (attention : les valeurs sont entières et définissent un facteur

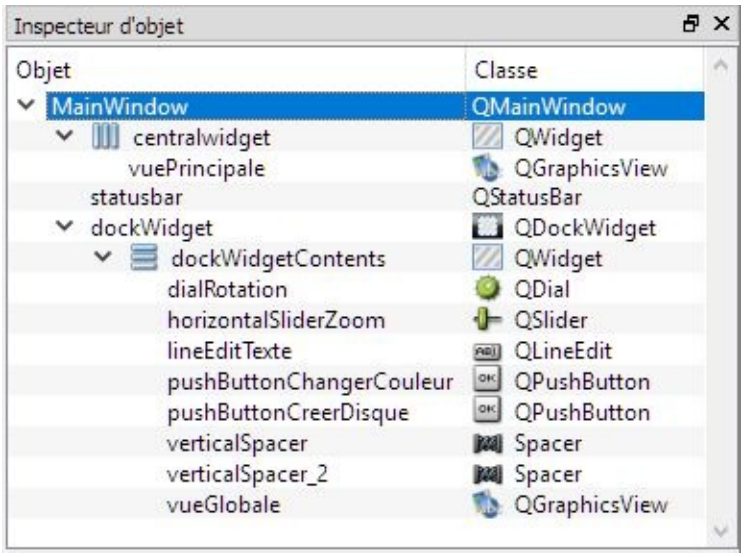
en pourcentage):

- minimum : 10 (facteur zoom minimal = 10 %) ;
- maximum : 1000 (facteur zoom maximal = 1000 %).

Une fois qu'on a sommairement positionné tous ces widgets, on peut faire l'alignement précis : on sélectionne le Dock Widget et on actionne Mettre en page verticalement, via le menu contextuel ou la barre d'outils.

Après ces opérations, la fenêtre devrait ressembler à la Figure 11.7 et l'inspecteur d'objets devrait s'afficher comme suit :

Figure 11.8 : *Fenêtre avec vues dans Qt Designer — Inspecteur d'objets*



Astuce *> Pour rappel, vous pouvez tester votre maquette en faisant une prévisualisation (Ctrl+R) ; sans surprise, les deux vues sont vides à ce stade puisqu'elles ne sont encore associées à aucune scène. On peut expérimenter la molette ainsi que le déplacement de la boîte à outils de gauche à droite de la fenêtre principale et même la possibilité de la détacher complètement.*

Nous en avons fini avec Qt Designer. Pour générer le code Python, faites Compiler la feuille sur `test_scene.ui` dans eric6. Pour rappel, sans eric6, la commande à exécuter est `pyuic5 test_scene.ui -o Ui_test_scene.py`.

Une fois notre `Ui_test_scene.py` obtenu, on peut commencer à coder. Pour la facilité, on va juste créer un fichier `test_scene.py`, qui contiendra à la fois la définition de la fenêtre et le programme principal. Commençons par créer une application qui affiche la

scène présentée dans la Section 1, Tous en scène !, sans nous préoccuper pour le moment des interactions avec les widgets placés dans notre boîte à outils.

```
# -*- coding: utf-8 -*-
# test_scene.py

import sys
from Ui_test_scene import Ui_MainWindow
from PyQt5.QtWidgets import QMainWindow
from PyQt5.QtCore import Qt
from PyQt5.QtGui import QBrush, QPainter
from PyQt5.QtWidgets import QApplication, QGraphicsScene,
QGraphicsView, \
                            QGraphicsItem, QGraphicsEllipseItem

class MainWindow(QMainWindow, Ui_MainWindow):
```

❶ def __init__(self, parent=None): super(MainWindow,self).__init__(parent) ❷ self.setupUi(self) self.scene = QGraphicsScene() ❸ self.remplirScene() self.show() ❹ for vue in (self.vuePrincipale, self.vueGlobale): ❺ vue.setScene(self.scene) vue.setRenderHints(QPainter.Antialiasing) vue.fitInView(self.rectGris, Qt.KeepAspectRatio) self.vueGlobale.setHorizontalScrollBarPolicy(Qt.ScrollBarAlwaysOff) ❻ self.vueGlobale.setVerticalScrollBarPolicy(Qt.ScrollBarAlwaysOff) self.vuePrincipale.setDragMode(QGraphicsView.RubberBandDrag) ❼ def remplirScene(self): ❽ scene = self.scene rectGris = scene.addRect(0,0,200,150,brush=QBrush(Qt.lightGray)) ❾ self.rectGris = rectGris self.texte = scene.addText("") ❿ dy = rectGris.rect().height()-self.texte.sceneBoundingRect().height() self.texte.setPos(rectGris.x(),rectGris.y() + dy) self.texte.setDefaultTextColor(Qt.cyan) scene.addItem(self.texte) d = 48. # diametre smiley ox = 4. # largeur oeil oy = 6. # hauteur oeil smiley = scene.addEllipse(-d/2,-d/2,d,d, brush=QBrush(Qt.yellow)) yeux = [QGraphicsEllipseItem(-ox/2.,-oy/2.,ox,oy,parent=smiley) \ for i in range(2)] yeux[0].setPos(-d/6,-d/8) yeux[1].setPos(+d/6,-d/8) brush = QBrush(Qt.black) for oeil in yeux: oeil.setBrush(brush) smiley.setPos(rectGris.mapToScene(rectGris.rect().center())) smiley.setRotation(20) smiley.setScale(1.5) for item in scene.items(): ⓫ item.setFlag(QGraphicsItem.ItemIsMovable) item.setFlag(QGraphicsItem.ItemIsSelectable) if __name__ == '__main__': ⓬ app = QApplication(sys.argv) mainWindow = MainWindow() sys.exit(app.exec_())

Voici quelques explications (pour la création des éléments de la scène, se référer à la Section 1, Tous en scène !).

❶ Comme expliqué à la Section 3, Génération du code, on utilise l'héritage multiple pour définir la classe MainWindow.

❷ L'initialisation de la fenêtre principale a été expliquée à la Section 3, Génération du code.

❸ On crée la scène et on l'assigne dans un attribut de la fenêtre principale ; n'importe quelle méthode peut dès lors accéder à la scène, que ce soit pour la modifier ou simplement l'observer. L'initialisation de la scène est confiée à la méthode `remplirScene` détaillée au point ❽.

❹ L'appel à `show` affiche la fenêtre principale. En principe, on fait cette opération en dernier lieu, mais, dans le cas présent, on la fait avant d'appeler `fitInView` : pour que cette méthode fonctionne bien, il faut que la vue sur laquelle elle s'applique ait sa taille définitive, ce qui est assuré par le fait de l'afficher.

❺ On fait une boucle sur les deux vues qui ont été créées dans Qt Designer pour effectuer les initialisations communes à celles-ci.

❻ Comme le but de la vue globale (sur la gauche) est de montrer toute la scène, on enlève les barres de défilement pour ne pas surcharger inutilement l'affichage. Les barres de défilement restent autorisées (par défaut) sur la vue principale.

❼ Cette instruction permet de faire une sélection par lasso (*rubber band* désigne un élastique) : on clique sur la scène et on trace un rectangle en déplaçant la souris sans lâcher le bouton ; les éléments en intersection avec le rectangle sont sélectionnés.

❽ La méthode `remplirScene` a pour tâche de garnir la scène au démarrage. Pour l'exemple, on crée des éléments en dur dans la méthode, comme déjà fait à la Section 1, Tous en scène !. Dans une application réelle, les données des éléments à créer seraient lues typiquement dans un fichier ou une base de données.

❾ On utilise ici les méthodes de commodité offertes par QGrapicsScene pour créer les éléments et les intégrer directement à la scène.

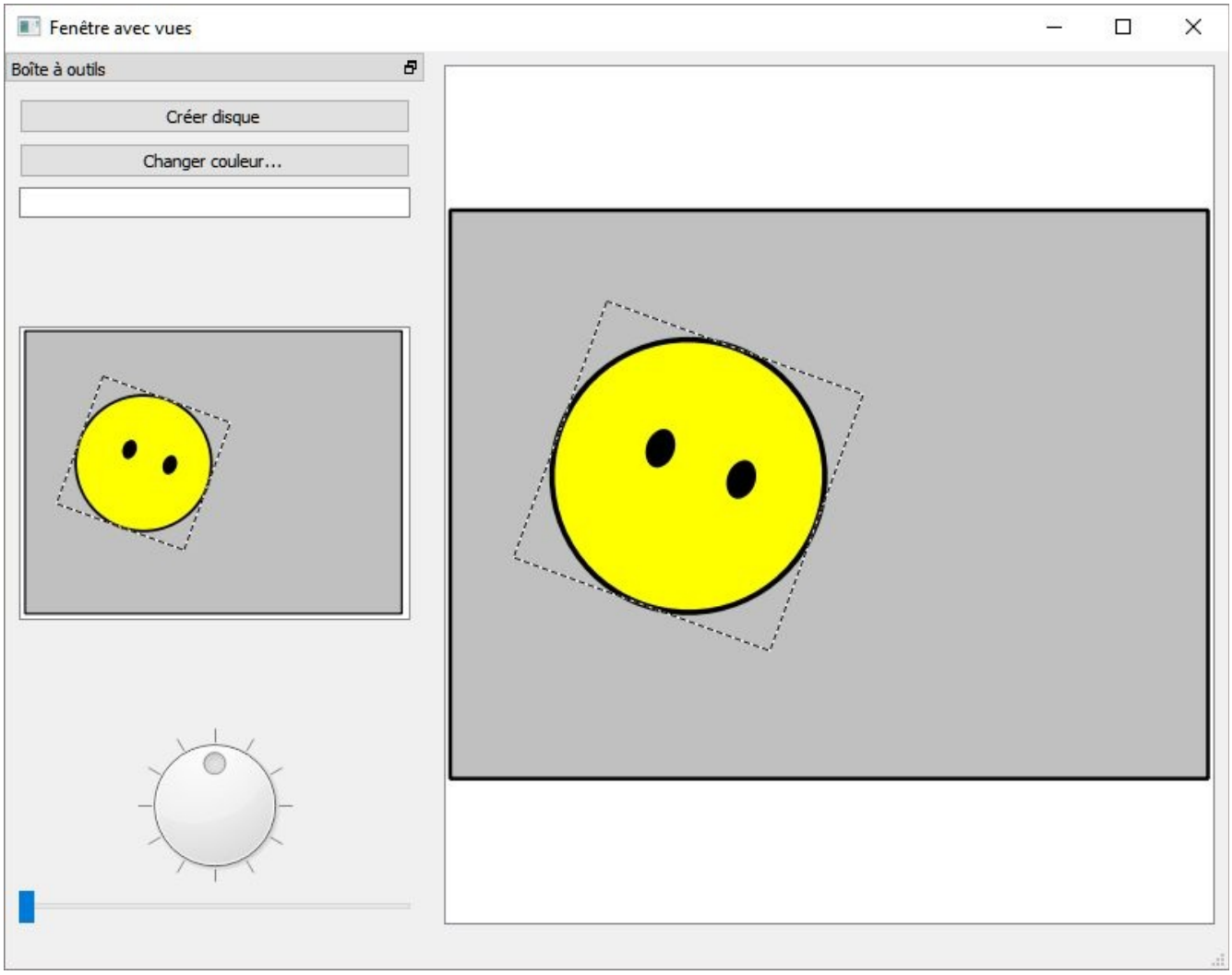
❿ L'élément texte est vide au départ, car on va l'initialiser via le champ de saisie (voir listing suivant).

⓫ La méthode `items()` renvoie la liste de tous les éléments placés sur la scène. La boucle permet de facilement assigner des propriétés communes à ces éléments en invoquant des méthodes de la classe parente QGraphicItem. Ici, on impose que tous les éléments puissent être déplacés et sélectionnés à la souris, contrairement à ce qu'on avait fait à la section précédente.

⓬ Le programme principal est classique pour une application PyQt : création de l'application et de l'instance de la fenêtre principale et démarrage de la boucle d'événements.

À l'exécution, on retrouve bien la scène vue à la section précédente, affichée dans les deux vues. Toutefois, on constate qu'à présent *tous* les éléments peuvent être déplacés, y compris chaque œil du smiley. On peut aussi sélectionner chaque élément en cliquant dessus, ce qui est dénoté par un rectangle hachuré. La multisélection et la désélection sont possibles aussi en maintenant la touche Ctrl enfoncée. On peut aussi sélectionner les éléments par lasso. On constate aussi que, grâce à la multisélection, plusieurs éléments peuvent être déplacés ensemble.

Figure 11.9 : *Fenêtre avec vues — v0.1*



Les widgets de la boîte à outils sont bien sûr parfaitement passifs à ce stade, puisqu'on n'a encore connecté aucun signal.

Nous allons nous atteler dès à présent à la dynamique de l'application, en commençant par le bouton Créer disque (dans ce qui suit, les imports sont à placer au début de `test_scene.py` et toutes les méthodes présentées sont à ajouter dans la classe `MainWindow`).

```
…
from PyQt5.QtCore import pyqtSlot
from PyQt5.QtGui import QPen

class MainWindow(QMainWindow, Ui_MainWindow):
    …
    @pyqtSlot()
    def on_pushButtonCreerDisque_clicked(self):
        disque =
```

```
self.scene.addEllipse(0,0,20,20,brush=QBrush(Qt.white),
                                                pen=QPen(Qt.NoPen))
        disque.setFlag(QGraphicsItem.ItemIsMovable)
        disque.setFlag(QGraphicsItem.ItemIsSelectable)
```

Pour exécuter la méthode lors d'un clic sur le bouton, on utilise la connexion signal-slot via le décorateur `pyqtSlot`. On crée un disque blanc à l'origine de la scène (dans le coin supérieur gauche de la vue globale). À noter : pour l'exemple, on a utilisé un pinceau invisible (type `NoPen`), ce qui a pour effet de créer une forme sans contour.

Passons à présent au champ de saisie textuel (QLineEdit) :

```
    def __init__(self, parent=None):
        …
        self.lineEditTexte.setText("Tous en scène !")
    …
    @pyqtSlot(str)
    def on_lineEditTexte_textChanged(self,msg):
        self.texte.setPlainText(msg)
```

On se connecte ici au signal `textChanged` qui est émis à chaque caractère tapé ; en assignant le texte saisi dans le QGraphicsTextItem via la méthode `setPlainText`, on obtient une mise à jour immédiate du texte sur les vues. Pour synchroniser le champ de saisie et l'élément graphique au lancement de l'application, on fait un `setText` dans le constructeur `__init__`, ce qui déclenche en réaction la méthode slot `on_lineEditTexte_textChanged`.

Passons au bouton Changer couleur… ; son but est de permettre à l'utilisateur de changer la couleur des éléments sélectionnés.

```
from PyQt5.QtWidgets import QColorDialog
…
    @pyqtSlot()
    def on_pushButtonChangerCouleur_clicked(self):
        itemsSelectionnes = self.scene.selectedItems()
        couleurInit = itemsSelectionnes[0].brush().color()
        couleur = QColorDialog.getColor(couleurInit,self,'Changer la
couleur')
        if couleur.isValid():
            brosse = QBrush(couleur)
            for item in itemsSelectionnes:
                item.setBrush(brosse)
```

Lors du clic, on retrouve les éléments sélectionnés sur la scène via la méthode `selectedItems` (rappelons que la multisélection est possible). On affiche un QColorDialog, la boîte de dialogue standard de votre OS pour choisir une couleur.

Dans cette boîte de dialogue, on fixe la couleur initiale à celle du premier élément sélectionné ; ceci n'est pas essentiel, mais c'est plus agréable pour l'utilisateur, surtout s'il n'y a qu'un seul élément sélectionné. Après avoir vérifié qu'on n'a pas quitté le dialogue par le bouton Cancel, on vérifie que la couleur choisie est valide ; si oui, on crée une nouvelle brosse avec cette couleur et on l'assigne à tous les éléments sélectionnés.

Tout cela est très bien… mais il y a un problème : si l'on clique sur le bouton sans qu'aucun élément soit sélectionné, il y aura une exception Python sur l'expression `itemsSelectionnes[0]`. On peut y remédier très facilement en protégeant les instructions avec un

```
        if len(itemsSelectionnes) > 0:
            …
```

Du point de vue ergonomique, ceci n'est clairement pas un bon choix : l'utilisateur clique sur un bouton et rien ne se passe ("Ai-je bien cliqué sur le bouton ?", "Quelque chose s'est-il passé à mon insu ?", etc.). On pourrait bien sûr afficher un message d'erreur pour ce cas, mais ce serait faire preuve d'autoritarisme. Il existe évidemment une meilleure solution : le bouton Changer couleur… ne doit être accessible que s'il y a au moins un élément sélectionné. C'est précisément ce que font les lignes qui suivent, grâce au signal `selectionChanged` émis par la scène chaque fois qu'un élément est sélectionné ou désélectionné :

```
    def __init__(self,parent=None):
        …

self.scene.selectionChanged.connect(self.onSceneSelectionChanged)
        self.onSceneSelectionChanged()
    …
    def onSceneSelectionChanged(self):
        nbElementSelectionnes = len(self.scene.selectedItems())

self.pushButtonChangerCouleur.setEnabled(nbElementSelectionnes > 0)
        msg = '%d éléments sélectionnés'%nbElementSelectionnes
        self.statusBar().showMessage(msg)
```

Dans le constructeur, on fait une connexion explicite signal-slot[15] et on invoque directement la méthode slot `onSceneSelectionChanged` pour faire les initialisations nécessaires. Dans `onSceneSelectionChanged`, on rend le bouton activable si et seulement si le nombre d'éléments est non nul. Au passage, on affiche un message dans la barre de statut, en bas de la fenêtre principale, pour indiquer à tout moment le nombre d'éléments sélectionnés.

À présent, voici comment la molette (QDial nommée `dialRotation`) va faire tourner la vue principale :

```
    def __init__(self,parent=None):
        …
        self.angleVue = 0.0
    …
    @pyqtSlot(int)
    def on_dialRotation_valueChanged(self,nouvelAngleVue):
        self.vuePrincipale.rotate(nouvelAngleVue-self.angleVue)
        self.angleVue = nouvelAngleVue
```

On a déjà vu précédemment la méthode `rotate` de QGraphicsView. Comme on a configuré la molette dans Qt Designer pour donner des valeurs entre −180 et +180, on peut directement interpréter cette valeur comme un angle de rotation. Comme on peut le voir, il y a toutefois une petite astuce à comprendre : la méthode `rotate` est cumulative, c'est-à-dire que l'angle passé en argument s'ajoute à l'angle de rotation courant de la vue. On doit donc s'arranger pour donner un angle relatif (différence entre le nouvel angle et l'angle courant). Comme QGraphicsView ne fournit pas une méthode simple pour connaître son angle courant, on maintient sa valeur dans un attribut `angleVue` initialisé à 0°.

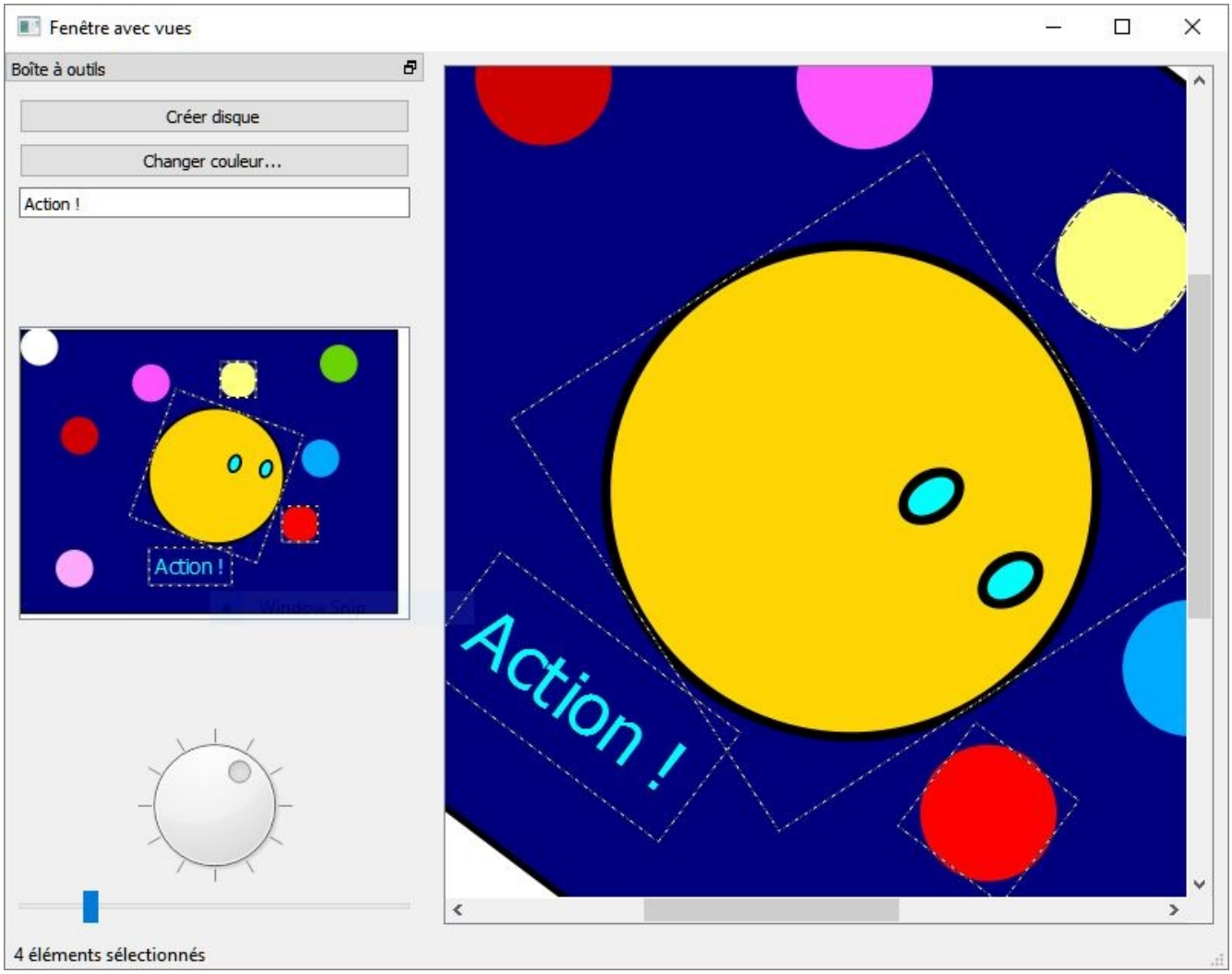
Pour terminer notre codage, on implémente le traitement du curseur (QSlider nommé `horizontalSliderZoom`) pour opérer les zooms :

```
    def __init__(self,parent=None):
        …
        self.zoomPctVue = 1.0
        self.horizontalSliderZoom.setValue(100)
    …
    @pyqtSlot(int)
    def on_horizontalSliderZoom_valueChanged(self,nouvZoomPctVue):
        f = (nouvZoomPctVue/100.0) / self.zoomPctVue
        self.vuePrincipale.scale(f,f)
        self.zoomPctVue = nouvZoomPctVue/100.0
```

Pour comprendre ces lignes, il faut se rappeler que la valeur passée par le curseur est un nombre entier entre 10 et 1000, à interpréter comme un pourcentage (voir la configuration de notre QSlider faite dans Qt Designer) ; il faut donc diviser cette valeur par 100.0 pour obtenir le facteur exprimé en virgule flottante à passer à `scale`. Par ailleurs, il faut ici aussi gérer les facteurs de zoom en relatif, car la méthode `scale`, à l'instar de `rotate`, est cumulative. Ici, on travaille bien sûr par division et pas par différence. L'instruction `setValue` dans le constructeur initialise les choses pour garantir que la valeur initiale du curseur est synchronisée avec le facteur de zoom (ceci n'était pas nécessaire avec la molette, initialisée à l'angle 0°).

Après ajout de toutes ces méthodes, on peut enfin lancer l'application et voir à l'œuvre les différentes fonctions de la boîte à outils : insertion de disques, déplacement, coloriage, message dans la barre de statut, rotation, zoom, etc. La Figure 11.10 montre ce qu'on peut obtenir après quelques manipulations.

Figure 11.10 : *Fenêtre avec vues — v1.0*



Astuce *> Une fois sélectionnée la molette, on peut l'actionner au clavier via les quatre touches directionnelles. Idem pour le curseur.*

Par cette petite application, qui ne fait qu'une centaine de lignes Python, on peut déjà voir tout le potentiel du framework Graphics View de Qt :

- on peut facilement créer une scène avec différents éléments géométriques — dont nous n'avons vu ici qu'une petite partie ;

- on peut attribuer un ensemble d'interactions standards à certains éléments (sélection, glisser-déposer) et cela de manière purement déclarative ;
- on intègre aisément une ou plusieurs vues sur cette scène dans une fenêtre classique ;
- on fait interagir les widgets de l'application avec la scène, les éléments et les vues grâce à des méthodes de haut niveau qui sont assez simples à comprendre.

On pourrait pousser l'exemple bien plus loin, en complexifiant la scène, sans perdre en fluidité. Dans le chapitre suivant, nous verrons d'ailleurs que le framework n'a aucun souci à supporter des dizaines de milliers d'éléments. Le but de ce dernier chapitre n'est toutefois pas de faire une démonstration de performance brute : on y verra surtout comment implémenter des interactions non standards en utilisant l'héritage pour les éléments et les vues.

---

[15] La connexion par décorateur pyqtSlot est possible aussi mais un peu plus difficile, car l'attribut `scene` n'a pas été défini dans Qt Designer (voir Section 4, Gestion des événements).

12

MosaiQ : une démo technologique !

Niveau : intermédiaire

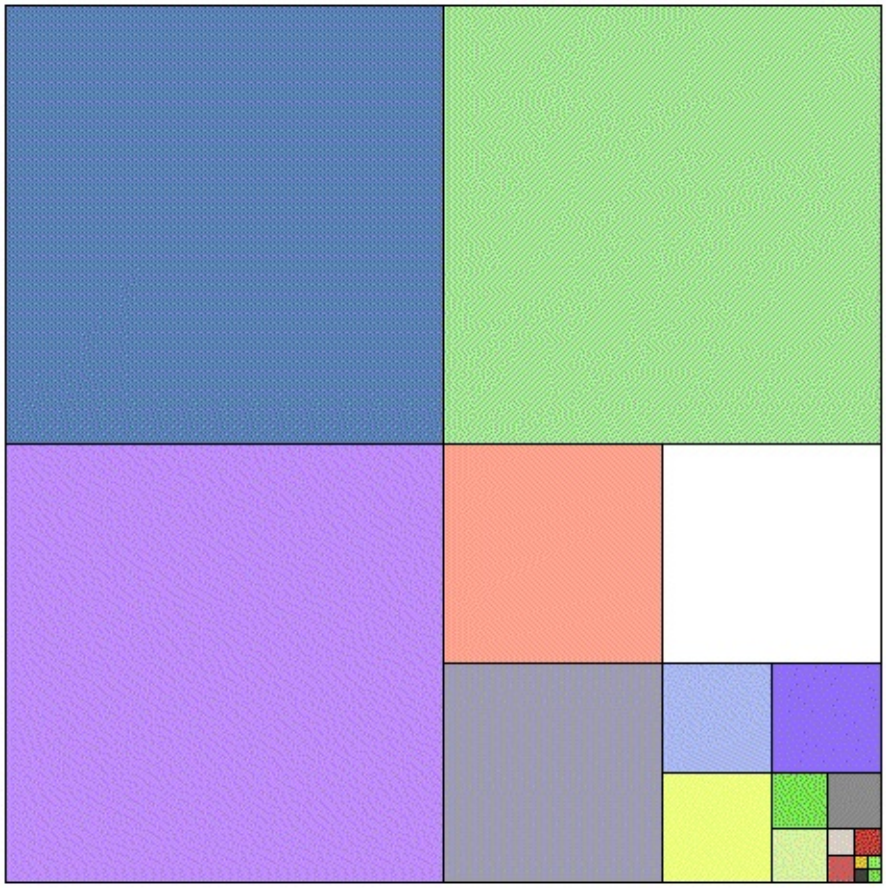
Objectifs : se familiariser avec les méthodes événementielles du framework Graphics View de Qt

Prérequis : Introduction au framework Graphics View de Qt, Première application avec une vue graphique

Dans le chapitre précédent, nous avons vu comment on pouvait assigner aux éléments des propriétés d'interaction standard héritées de QGraphicsItem. Nous avons vu aussi qu'on pouvait utiliser les connexions signaux-slots pour modifier les vues, la scène ou ses éléments à partir de widgets. Vous avez pu remarquer que pour toutes les interactions simples présentées jusqu'ici, on n'a pas fait autre chose qu'utiliser les techniques du module Développement d'une application avec des widgets. Dans une application graphique plus avancée, on voudra cependant des interactions sur mesure pour les éléments de la scène avec, si possible, un comportement différent selon le type d'élément. Dans ce chapitre, nous allons voir comment les classes du framework permettent de réaliser cela assez facilement, par héritage et surcharge de méthodes.

Pour illustrer ces concepts, nous allons nous transporter dans un univers un peu spécial… À l'origine, il n'y a dans cet univers qu'un simple carré flottant dans l'Espace. Le maître de cet univers (qu'on supposera en dehors de l'Espace !) n'a qu'un seul pouvoir : celui de fragmenter tout carré en quatre carrés plus petits ayant chacun une couleur aléatoire. Après avoir fragmenté le carré originel, le maître peut choisir un des carrés obtenus et le fragmenter, à son tour, en quatre carrés encore plus petits ; en exerçant ce pouvoir de manière répétée, il peut créer, où bon lui semble, des carrés de plus en plus petits. La Figure 12.1 illustre le concept, en supposant que le maître de l'univers ait procédé à six fragmentations, successivement sur les carrés du coin inférieur gauche.

Figure 12.1 : *Fragmentation*



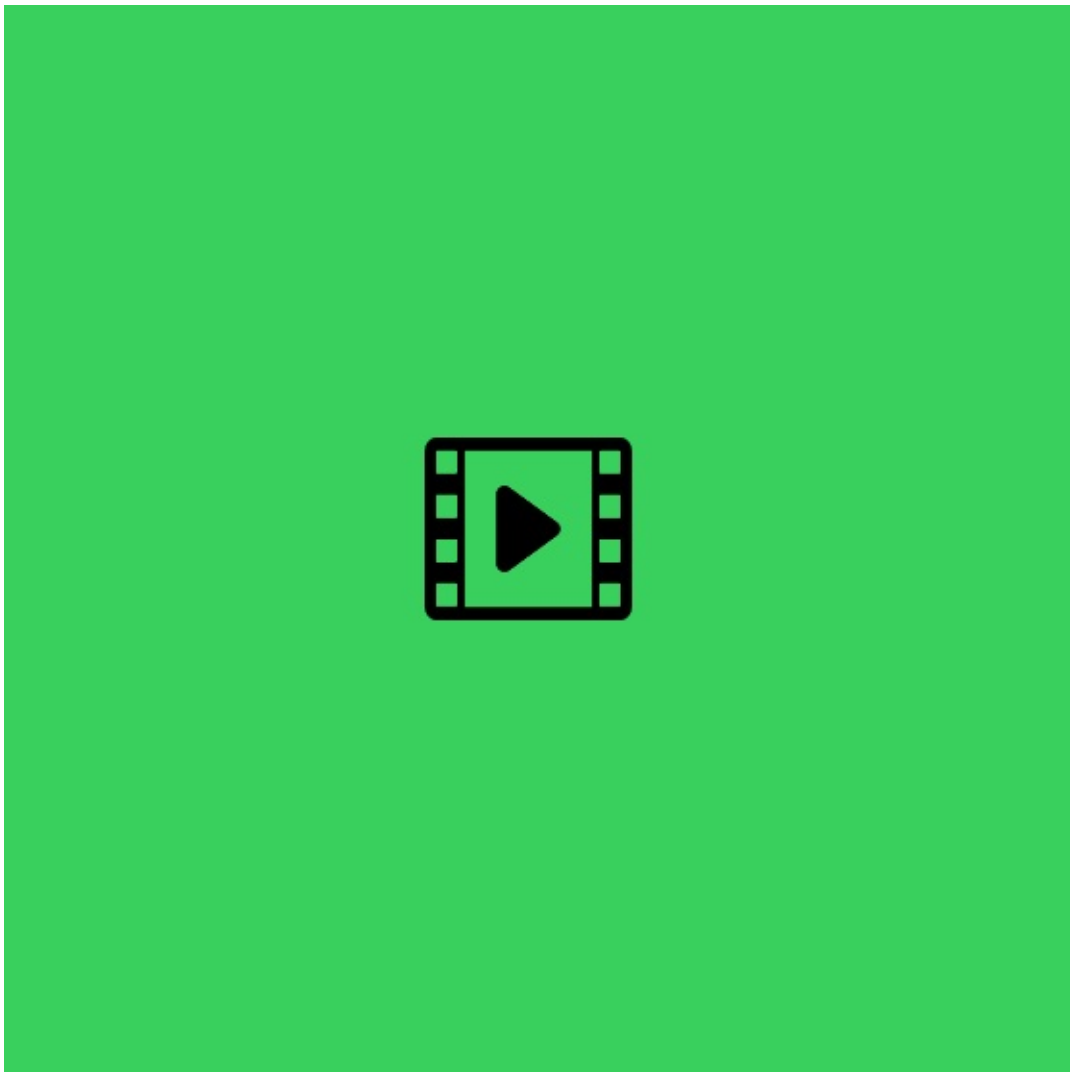
Pour voyager dans cet univers étrange, nous allons créer une application PyQt, appelée *MosaiQ*. Cette application, qui illustre le concept d'arbre quaternaire (*quadtree*), va vous montrer comment implémenter des interactions spécifiques sur les éléments graphiques et les vues du framework Graphics View. Le maître de l'univers sera bien sûr l'utilisateur ; il pourra fragmenter les carrés via de simples clics à la souris. Bien entendu, on arrivera assez vite à une fragmentation en carrés minuscules ayant la taille d'un pixel (après dix clics au même endroit, on obtient un carré ayant 1/1024 de la taille du carré initial). L'utilisateur fera alors appel à la fonction de zoom, réalisée ici avec la molette de la souris : il pourra ainsi agrandir une zone de la scène et continuer la fragmentation à un niveau microscopique.

Fidèle à notre habitude, après une petite introduction théorique, nous allons présenter d'abord une application très simplifiée qu'on enrichira progressivement avec de nouvelles fonctionnalités. Voici un résumé de ce que nous allons faire :

- surbrillance des éléments survolés ;
- fragmentation par clic ;

- zoom à la souris et au clavier ;
- mode plein écran ;
- capture d'écran ;
- fragmentation continue ;
- affichage d'un panneau d'information.

Figure 12.2 : *MosaiQ en action… (vidéo)*



Note > *Le code source final de cet exemple est disponible dans le projet* `MosaiQ` .

1. Gestion des événements par patron de méthode

Avant de nous lancer dans le développement de MosaiQ, il est nécessaire de présenter les principes de la technique qu'on va utiliser. On va utiliser un patron de conception omniprésent dans les frameworks orientés objet : le *patron de méthode* (en anglais *template method*). L'idée est assez simple à comprendre : quand un événement utilisateur se produit sur une vue donnée (déplacement souris, clic, touche clavier, etc.), le framework appelle une méthode de QGraphicsView spécifique au type d'événement ; cette méthode implémente un comportement par défaut ou même, souvent, ne fait rien. Si l'on crée une classe qui hérite de QGraphicsView, on peut dès lors redéfinir (ou, plus précisément, *surcharger*) certaines de ces méthodes pour obtenir un certain comportement. Voici quelques exemples de telles méthodes :

- `keyPressEvent` : une touche clavier a été enfoncée ;
- `keyReleaseEvent` : une touche clavier a été relâchée ;
- `mouseMoveEvent` : la souris a été déplacée ;
- `mousePressEvent` : un bouton de la souris a été enfoncé ;
- `mouseReleaseEvent` : un bouton de la souris a été relâché ;
- `mouseDoubleClickEvent` : un double-clic a été fait sur un bouton de la souris ;
- `wheelEvent` : la molette de la souris a été déplacée.

Note *> En C++, les méthodes qu'on peut surcharger sont dites virtuelles ; leurs signatures sont marquées du mot clé* `virtual`*. Dans la documentation Qt C++, ces méthodes sont regroupées dans la section Protected Functions de chaque classe. En Python, qui est un langage beaucoup moins contraint, il n'y a pas ce genre de distinction et l'on pourrait dire en fait que toutes les méthodes sont virtuelles. Pour cette raison, afin de déterminer quelles méthodes peuvent être surchargées et quels sont leurs rôles respectifs, il peut être utile de se référer à la documentation Qt C++ plutôt qu'à celle de PyQt. Toutes ces méthodes ont un argument d'une classe qui hérite de la classe abstraite QEvent (QKeyEvent, QMoveEvent, QMouseEvent, etc.) ; cet argument fournit toutes les informations sur l'événement qui s'est produit :*

touche enfoncée, coordonnées du pointeur, bouton souris enfoncé, etc.

En définissant une classe vue héritant de QGraphicsView, on a donc la possibilité de surcharger toutes ces méthodes de type événementiel. Mais le framework Graphics View va plus loin… Il propage intelligemment les événements de la vue vers la scène qui lui est associée et, surtout, vers les éléments qu'elle contient. Ainsi, toutes les méthodes précitées existent aussi au niveau des classes QGraphicsScene et QGraphicsItem, où elles pourront être surchargées. Typiquement, si on veut pouvoir cliquer sur un certain type d'élément de la scène pour faire apparaître un menu contextuel, on fera une classe qui hérite par exemple de QGraphicsRectItem et dans laquelle on surchargera la méthode `mousePressEvent`.

Il y a deux avantages majeurs à la technique qu'on vient de décrire :

- on n'a pas de connexions signaux-slots à établir ;
- on ne doit pas programmer d'algorithme de recherche des éléments ciblés par un événement donné : les éléments concernés sont directement notifiés.

Le seul prix à payer — mais il est modéré en Python ! — est qu'on doit créer des classes qui héritent des classes offertes par le framework plutôt que les utiliser directement (comme nous l'avons fait dans la section précédente).

Après cette introduction, nous sommes prêts à démarrer la programmation de l'application MosaiQ.

2. Carré initial et surbrillance

Commençons par dessiner une scène avec un seul carré. La première fonction qu'on va réaliser consiste à éclairer le carré lorsqu'il est survolé par le pointeur souris. Ceci est courant — et utile ! — dans beaucoup d'interfaces utilisateur (menus, barres d'outils, etc.) : on indique ainsi clairement l'élément qui sera affecté en cas de clic. Rappelons que la scène sera bientôt remplie de milliers de petits carrés.

À cette fin, on utilise l'héritage et le patron de méthode présenté plus haut. Lorsque le pointeur souris survole un QGraphicsItem, le framework appelle deux méthodes : `hoverEnterEvent` quand il entre dans la surface de l'élément et `hoverLeaveEvent` quand il en sort. Dès lors, on crée une classe CarreMosaiQ qui hérite de QGraphicsRectItem ; il suffira alors de surcharger ces deux méthodes pour obtenir l'effet voulu.

Voyons comment faire cela concrètement.

```
#-*- coding: latin_1 -*-
# MosaiQ v0.1

import sys
from random import randint

from PyQt5.QtCore import Qt
from PyQt5.QtGui import QBrush, QPen, QPainter, QColor
from PyQt5.QtWidgets import QApplication, QGraphicsScene,
QGraphicsView,\
                            QGraphicsRectItem

# paramètres de configuration
# couleur du fond de la scène
COULEUR_FOND    = "white"
```

❶ # couleur utilisée pour le contour des carrés COULEUR_CONTOUR = "black" # plage (min,max) pour les composantes RGB de la couleur des carrés PLAGE_COULEURS = (50,250) # pourcentage utilisé pour éclairer les carrés survolés FACTEUR_ECLAIRCISSSEMENT = 150 class CarreMosaiQ(QGraphicsRectItem): ❷ _pen = QPen(QColor(*COULEUR_CONTOUR_RGB)) ❸ _pen.setCosmetic(True) def __init__(self,x,y,c): ❹ QGraphicsRectItem.__init__(self,0,0,c,c) self.setPos(x,y) self.setPen(CarreMosaiQ._pen) couleur = QColor(*(randint(*PLAGE_COULEURS) for _ in 'RGB')) ❺ self.brush = QBrush(couleur) ❻ self.setBrush(self.brush)

self.setCursor(Qt.OpenHandCursor) ❼ self.setAcceptHoverEvents(True) ❽ def hoverEnterEvent(self,event): couleur = self.brush.color().lighter(FACTEUR_ECLAIRCISSSEMENT) ❾ self.setBrush(QBrush(couleur)) def hoverLeaveEvent(self,event): self.setBrush(self.brush) ❿ class SceneMosaiQ(QGraphicsScene): ⓫ def __init__(self): QGraphicsScene.__init__(self) size = 2.0**32 ⓬ self.carreMosaiQUnivers = CarreMosaiQ(0,0,size) ⓭ self.addItem(self.carreMosaiQUnivers) class VueMosaiQ(QGraphicsView): ⓮ def __init__(self,scene): QGraphicsView.__init__(self,scene) self.setHorizontalScrollBarPolicy(Qt.ScrollBarAlwaysOff) ⓯ self.setVerticalScrollBarPolicy(Qt.ScrollBarAlwaysOff) self.setBackgroundBrush(QColor(COULEUR_FOND)) self.setWindowTitle("MosaiQ") self.resize(800,600) self.fitInView(scene.carreMosaiQUnivers,Qt.KeepAspectRatio) if __name__ == "__main__": app = QApplication(sys.argv) sceneMosaiQ = SceneMosaiQ() ⓰ vueMosaiQ = VueMosaiQ(sceneMosaiQ) vueMosaiQ.show() sys.exit(app.exec_())

❶ Il est de bonne pratique de placer les différents paramètres importants de l'application dans des variables. On peut ainsi facilement les modifier sans parcourir le code source. On pourra, à terme, déplacer ces paramètres dans un ficher de configuration à importer.

❷ CarreMosaiQ hérite de QGraphicsRectItem car on veut dessiner un carré (pour rappel, un carré est un cas particulier de rectangle !).

❸ On veut le même pinceau pour dessiner tous les contours de nos instances de carré : on crée donc ce pinceau une fois pour toutes, en attribut de classe. Il suffira alors de l'assigner à chaque nouvelle instance créée. On définit ce pinceau comme cosmétique, car on prévoit de faire de grands zooms et on veut éviter que les contours des carrés grossissent démesurément. On pourrait tout aussi bien définir un pinceau invisible (`Qt.NoPen`), sachant que les couleurs aléatoires des brosses suffisent à distinguer les carrés ; c'est juste un choix esthétique qu'on pourrait d'ailleurs rendre configurable.

❹ Le constructeur CarreMosaiQ a juste besoin de trois arguments : position (`x`,`y`) du coin supérieur gauche sur la scène et longueur du côté `c`. Même si c'est assez rare, on a ici une signature qui a moins d'arguments que le constructeur de la classe parente : comme on veut dessiner un carré, il est inutile de demander à l'appelant de

répéter la longueur du côté ; ceci est fait une fois pour toute dans l'appel au constructeur de QGraphicsRectItem.

❺ Cette instruction condensée choisit une couleur au hasard en triant au sort chaque composante RGB dans la plage configurée. Détaillons ce qu'on fait… On utilise deux fois l'expansion des arguments Python (avec l'opérateur astérisque) ; si on pose les deux limites de la plage avec `(b,h)=PLAGE_COULEURS`, l'expression est exactement équivalente à `QColor(randint(b,h),randint(b,h),randint(b,h))`. La boucle sur la chaîne `'RGB'` est juste un artifice pour compter jusqu'à trois en rappelant ce qu'on fait.

❻ On crée une brosse avec la couleur choisie et on la sauvegarde dans un attribut dédié ; cette mémorisation a juste pour but de pouvoir restaurer la brosse après l'éclairement (voir ❿).

❼ Grâce à cette instruction, on peut assigner un pointeur de souris particulier qui remplacera le pointeur standard quand on survole le carré : ici, une petite main. C'est une manière commode et standard de signaler à l'utilisateur qu'il peut, s'il le désire, effectuer une certaine action sur l'élément.

❽ Pour que les appels aux méthodes `hoverEnterEvent` et `hoverLeaveEvent` se produisent effectivement sur un élément, il faut activer la propriété `AcceptHoverEvents` sur cet élément. Cette petite contrainte sert juste à optimiser le framework (par défaut, un élément ne sait pas qu'il est survolé).

❾ Quand le pointeur souris entre dans le carré, on récupère la couleur courante et on calcule une nouvelle couleur en l'éclaircissant avec la méthode `lighter`. On assigne au carré une nouvelle brosse avec cette couleur ; cette brosse temporaire durera tant que le pointeur est dans le carré.

❿ Quand le pointeur souris sort du carré, on restaure la brosse initiale, avec donc la couleur avant éclairement.

⓫ On définit une scène sur mesure SceneMosaiQ, en héritant de QGraphicsScene.

⓬ Comme on va faire des zooms importants, il est intéressant de définir un très grand carré, pour pouvoir aller le plus loin possible avant de tomber sous la limite du plus petit nombre représentable en virgule flottante. Rappelons qu'on est ici en coordonnées scène, pas en coordonnées écran.

⓭ On crée le premier carré sur la scène, celui par lequel tout commence… On le met dans un attribut de la scène `carreMosaiQUnivers` afin de pouvoir le référencer lorsqu'on cadrera la vue.

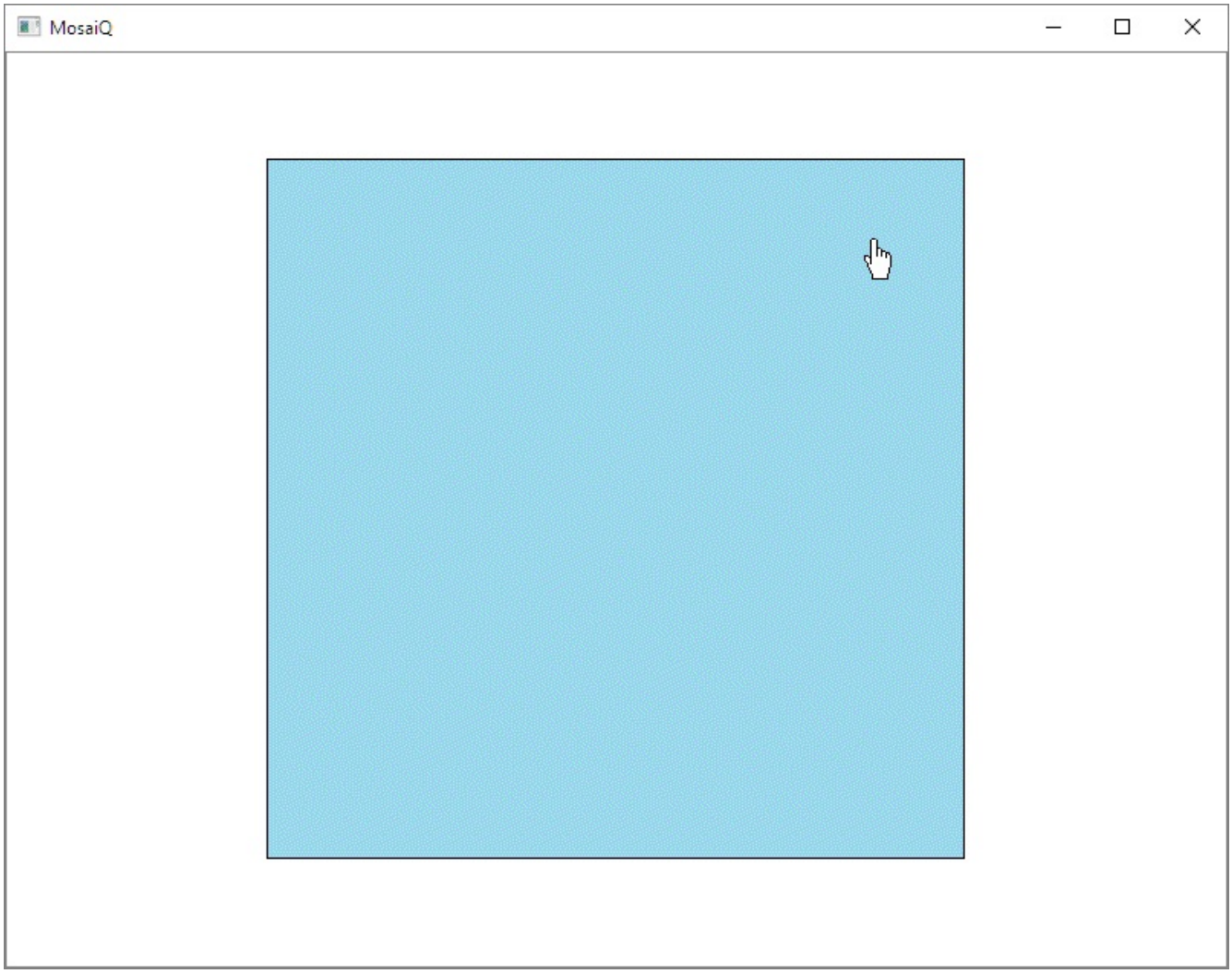
⓮ On définit une vue sur mesure VueMosaiQ, en héritant de QGraphicsView.

⓯ Dans cette application, on se déplace uniquement via des actions souris ou clavier ; pour ne pas surcharger l'affichage, on choisit de cacher les barres de défilement.

⓰ Le programme principal est sans surprise : on crée une scène et une vue en instanciant les deux classes que nous avons définie.

Quand on lance cette application, on peut voir apparaître le grand carré, coloré aléatoirement. Lorsqu'on passe le pointeur souris au-dessus du carré, on voit que sa couleur change et que le pointeur se transforme en petite main ; ces effets disparaissent lorsque le pointeur sort du carré.

Figure 12.3 : *MosaiQ v0.1 (carré initial)*


MosaiQ

3. Fragmentation par clic

Nous voilà maintenant prêts à réaliser la fonction principale de l'application : un clic sur le carré doit le fragmenter en quatre carrés plus petits. Réagir au clic sur le carré est très simple : il suffit de surcharger la méthode `mousePressEvent` de la classe CarreMosaiQ. Voici les deux méthodes à ajouter à cette classe :

```
…
class CarreMosaiQ(QGraphicsRectItem):
    …
    def mousePressEvent(self,mouseEvent):
        if mouseEvent.button() == Qt.LeftButton:
```

❶ self.fragmente() ❷ else: QGraphicsRectItem.mousePressEvent(self,mouseEvent) ❸ def fragmente(self): c = self.rect().width() / 2 ❹ x = self.x() ❺ y = self.y() scene = self.scene() ❻ scene.removeItem(self) ❼ for (dx,dy) in ((0,0),(c,0),(0,c),(c,c)): ❽ scene.addItem(CarreMosaiQ(x+dx,y+dy,c))

❶ La méthode `mousePressEvent` est invoquée quel que soit le bouton actionné. On reçoit en argument un QGraphicsSceneMouseEvent, qui donne tous les détails sur l'événement : bouton enfoncé, position sur la scène, etc. Pour se permettre d'associer plus tard de nouvelles fonctions aux autres boutons, on fait un test pour ne prendre en compte que les clics sur bouton gauche.

❷ On factorise le traitement dans la méthode `fragmente` : on fait ainsi un découplage entre événement et traitement, un peu comme avec les signaux et slots. Cela facilitera l'ajout de nouvelles fonctions qui fragmentent le carré via d'autres actions.

❸ Si le bouton cliqué n'est pas le bouton gauche, on fait le traitement par défaut de la classe parente. Il est fort probable que cet appel ne fasse rien, mais il est de bonne pratique, avec le patron de méthode, d'appeler la méthode parente comme traitement par défaut.

❹ La méthode `fragmente` a quatre nouveaux carrés à créer ; on calcule la taille de ces carrés en divisant la taille du côté du carré pointé par deux.

❺ On récupère les coordonnées (`x,y`) du coin supérieur gauche du carré pointé.

❻ On récupère la scène où se trouve le carré pointé ; dans notre cas, ce sera l'instance (unique) de SceneMosaiQ. Notons que, comme on ne fait aucune hypothèse sur la scène, on pourrait très bien réutiliser la classe CarreMosaiQ dans n'importe quelle autre scène, instance directe ou indirecte de QGraphicsScene.

❼ L'opération de fragmentation consiste en fait à supprimer le carré pointé de la scène et de le remplacer par quatre nouveaux. Il peut sembler dangereux a priori d'enlever de la scène l'élément sur lequel on est précisément en train d'agir. Ne sommes-nous pas en train de scier la branche sur laquelle nous sommes assis ? En fait, il n'y a aucun risque dans cette opération : la méthode `removeItem` ne fait que remettre en coulisse l'élément qui était sur scène ; il disparaît automatiquement de la vue (et donc de l'écran), mais il existe toujours en mémoire ; tant qu'on a une référence vers cet élément, on peut continuer à invoquer des méthodes sur lui, voire le replacer sur la scène ou le placer sur une autre scène. Dans le cas particulier de notre application, au sortir de la méthode `fragmente`, le carré enlevé de la scène ne sera plus référencé par aucun objet et sera effectivement détruit ; la seule exception est l'élément carré initial, qui n'est jamais détruit car référencé dans l'attribut `carreMosaiQUnivers` de la scène (voir `SceneMosaiQ.__init__`).

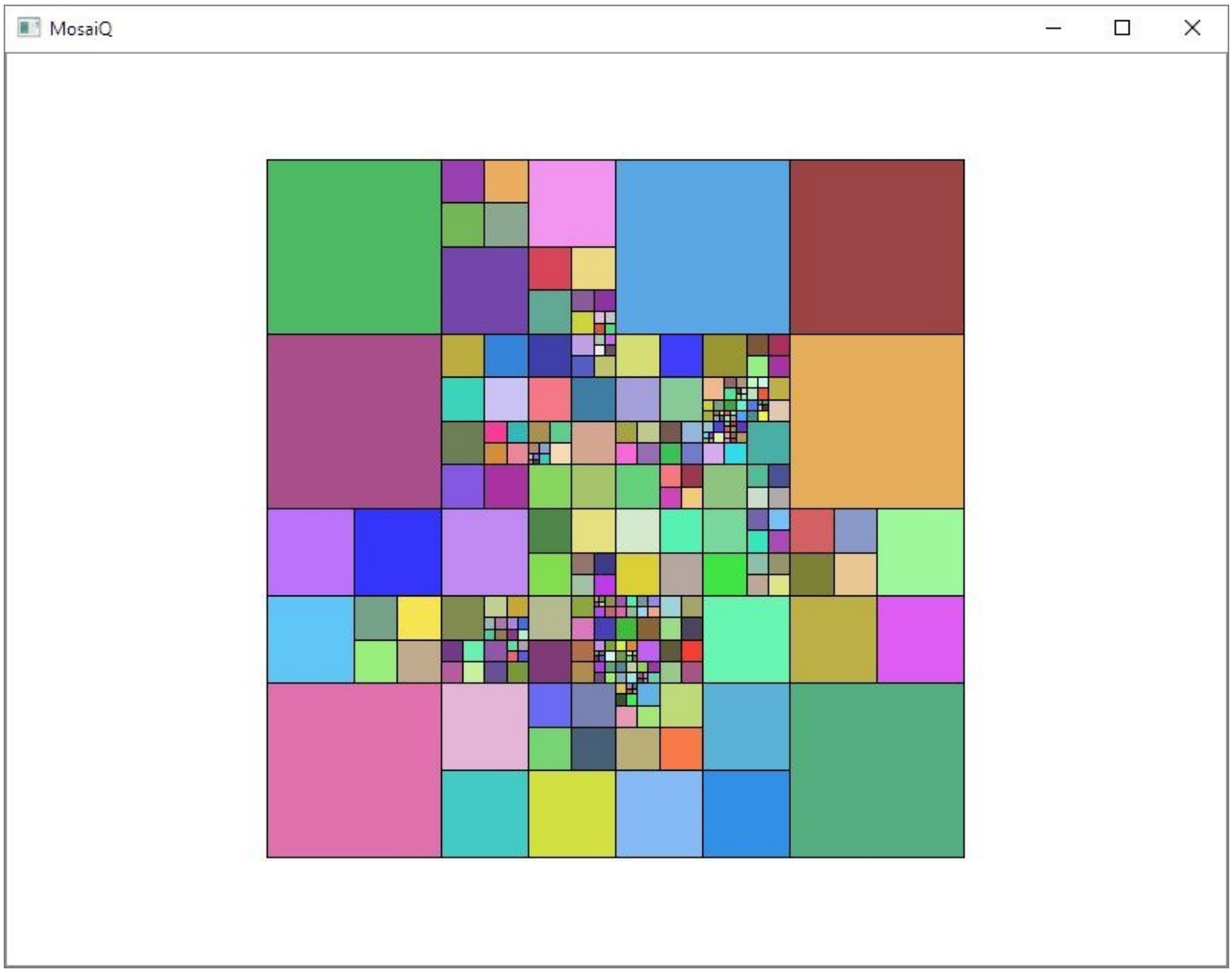
***Note** > Une autre technique possible aurait été de rendre le carré invisible, en faisant `self.setVisible(False)` : l'élément reste alors sur la scène, mais n'est plus montré sur la vue. Visuellement, le résultat est le même, mais on consomme inutilement de la mémoire. Par contre, on gagne un tout petit peu en vitesse, car la gestion interne de données par le framework est plus simple ; le gain est tout à fait imperceptible dans le cas présent mais on peut le noter avec la technique de fragmentation continue exposée plus loin. On a donc ici un cas classique de compromis entre performance et consommation mémoire.*

❽ Le cœur de l'algorithme est ici : la boucle fait quatre itérations en donnant chaque fois les coordonnées `(dx,dy)` du nouveau carré à créer, relatives au coin supérieur gauche (`x,y`) du grand carré qu'on vient de faire disparaître. À chaque itération, on crée une nouvelle instance de CarreMosaiQ, avec le coin supérieur gauche aux coordonnées scène (`x+dx,y+dy`). On aurait bien entendu pu faire tout ceci en quatre instructions distinctes, mais, comme on l'a déjà vu, la syntaxe de Python permet d'éviter beaucoup de redondances. Après cette boucle, on a quatre nouveaux carrés représentés à l'écran ; chacun a une couleur aléatoire et est prêt à réagir

individuellement aux clics et au survol du pointeur.

À l'exécution, vous pouvez à présent fragmenter le carré en cliquant et continuer ainsi sur les carrés plus petits. On constate que tous les événements souris sont bien dirigés à la bonne instance de CarreMosaiQ, qui réagit comme attendu via les trois méthodes que nous avons surchargées.

Figure 12.4 : *MosaiQ v0.2 (fragmentation par clic)*



4. Zoom à la souris

Après avoir joué un peu avec l'application, on se rend compte qu'on atteint assez vite les limites du système lorsque les carrés deviennent trop petits : le pointeur *main* occulte les détails et on ne voit plus exactement sur quel carré on clique. Une fonction zoom est donc ardemment requise.

On a vu au chapitre précédent comment commander le zoom via un widget curseur (QSlider). Au niveau de l'ergonomie, ceci n'est clairement pas la technique la plus adaptée, car on impose à l'utilisateur de déplacer le curseur souris sur un widget hors de la scène. Au lieu de cela, nous allons programmer le zoom via la molette de la souris et, en alternative, via le clavier. Dans les deux cas, on fera en sorte que le zoom soit centré sur la position du pointeur souris, ce qui est la norme dans les applications cartographiques.

Commençons par le zoom via la molette de la souris. Selon la philosophie du framework, le seul point à modifier est l'implémentation de la vue, c'est-à-dire la classe VueMosaiQ.

```
…
class VueMosaiQ(QGraphicsView):

    def __init__(self,scene):
        …
        self.setTransformationAnchor(QGraphicsView.AnchorUnderMouse)
```

❶ def wheelEvent(self,event): ❷ self.zoom(event.angleDelta().y()/100.0) ❸ def zoom(self,facteur): if facteur < 0.0: ❹ facteur = -1.0 / facteur self.scale(facteur,facteur)

❶ Cette instruction définit un *point d'ancrage* pour les transformations de la vue (le zoom notamment), c'est-à-dire un point dont les coordonnées vue ne seront pas affectées par la transformation. Par défaut, le point d'ancrage est `QGraphicsView.AnchorViewCenter`, le point de la scène représenté au centre de la vue reste au centre de la vue ; c'est ce que nous avons fait dans les exemples de zoom au chapitre précédent. Ici, la valeur `QGraphicsView.AnchorUnderMouse` impose que le point d'ancrage soit le point de la scène sous le pointeur souris. L'utilisateur pourra ainsi facilement zoomer où bon lui semble.

La méthode `wheelEvent` est appelée à chaque mouvement de la molette (ou

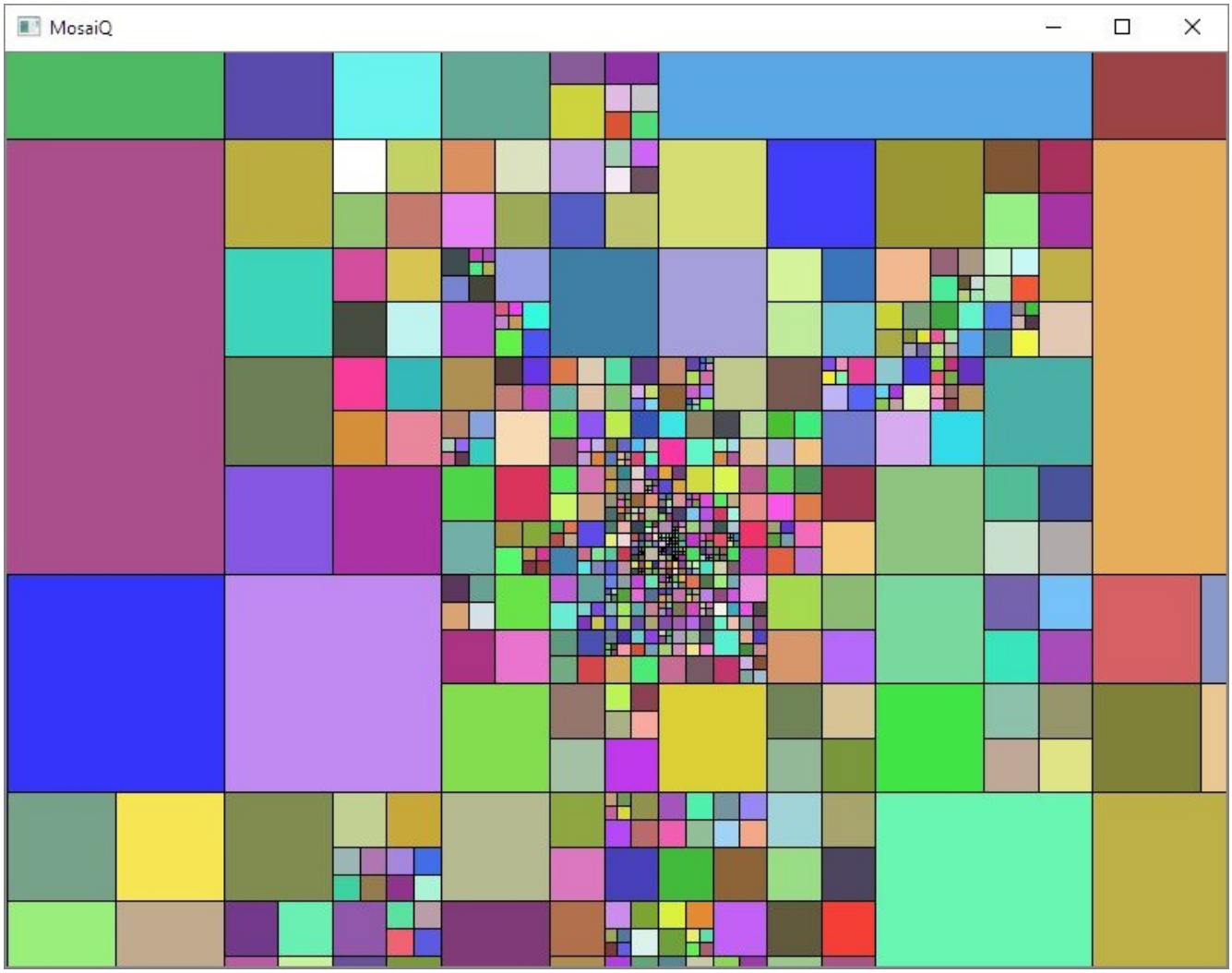
❷ équivalent — par exemple, mouvement à deux doigts sur un pavé tactile). L'argument `event` reçu est un QWheelEvent qui donne les détails de l'action effectuée.

❸ La valeur de déplacement de la molette est donnée par `event.angleDelta().y()`, une valeur qui dépend du type de souris. Elle vaut typiquement +120 pour un déplacement vers le haut et —120 pour un déplacement vers le bas ; en divisant cette valeur par 100, on obtient un zoom de +20 % ou −20 %. Ce facteur 100 pourrait bien sûr être adapté pour changer la sensibilité du mouvement ou pour s'accommoder du type de dispositif (souris ou pavé tactile) ; pour être plus propre, on pourrait définir ce facteur comme un paramètre de configuration modifiable par l'utilisateur.

❹ Pour traiter correctement les facteurs de zoom négatifs, on a créé une fonction `zoom`. La méthode `scale` attend un facteur positif (plus grand que 1 si zoom avant, plus petit que 1 si zoom arrière) ; il faut donc transformer les valeurs négatives de déplacement de la molette en facteurs positifs plus petits que 1. Pour obtenir une symétrie entre zoom avant et arrière, on applique la formule `-1.0 / facteur`. Par exemple, la valeur −1,2 (zoom arrière) est convertie en 1/1,2, c'est-à-dire 0,8333. Sans cette opération, on constate que la méthode `scale` inverse toutes les coordonnées, ce qui est très déstabilisant.

Après avoir ajouté ces quelques lignes, on peut à présent opérer des zooms dans les deux sens en tournant la molette de la souris. On peut ainsi agrandir les détails de la scène et fragmenter en carrés de plus en plus petits.

Figure 12.5 : *MosaiQ v0.3 (zooming)*



Jusqu'où peut-on zoomer ?

Assez vite, vous tenterez sans doute l'expérience du zoom extrême, tel le physicien des hautes énergies qui cherche à percer les secrets de l'infiniment petit ! Comme avec la matière tangible, il y a malheureusement une limite… Vous constaterez après avoir été trop loin dans les zooms que des phénomènes étranges se produisent : la vue change brusquement et on se retrouve sans raison dans le coin supérieur gauche du grand carré initial (ce qu'on peut vérifier en faisant des zooms arrière). Il y a donc, comme on pouvait s'en douter, une certaine limite à ne pas franchir pour garantir que l'application continue à fonctionner normalement. Ceci est dû à la représentation des coordonnées de la vue en nombres entiers ; nous verrons à la fin de ce chapitre comment calculer cette limite et comment empêcher son franchissement.

En alternative aux zooms avant/arrière pour naviguer dans la scène, signalons qu'on peut utiliser les quatre touches directionnelles du clavier pour se déplacer. Il s'agit d'une fonction héritée de QGraphicsView, elle-même héritée de QAbstractScrollArea.

Pour améliorer la précision du point d'ancrage…

Peut-être avez vous remarqué de petites anomalies lors des zooms avant ou arrière : parfois, le carré pointé change pour un carré voisin alors que le pointeur souris reste immobile. Ceci est inattendu, car on a bien spécifié un point d'ancrage `AnchorUnderMouse` dans le constructeur de la vue. En fait, il semble que cette fonction du framework souffre d'une instabilité numérique due au fait que les coordonnées scène opèrent, pour notre application un peu particulière, sur une très grande plage de valeurs. Avant de parler de bug, on peut faire l'hypothèse que les développeurs de Qt ont privilégié la performance à la précision, ce qui semble un choix acceptable pour la plupart des applications.

Le problème étant posé, il est possible de corriger cette imprécision en utilisant les méthodes primitives de QGraphicsView. Voici une recette possible (la méthode `wheelEvent` reste inchangée).

```
…
from PyQt5.QtGui import QCursor
…
class VueMosaiQ(QGraphicsView):

    def __init__(self,scene):
        …
        self.setTransformationAnchor(QGraphicsView.NoAnchor)
```

❶ self.hScrollBar = self.horizontalScrollBar() self.vScrollBar = self.verticalScrollBar() def zoom(self,facteur): if facteur < 0.0: facteur = -1.0 / facteur posVue1 = self.mapFromGlobal(QCursor.pos()) ❷ posScene = self.mapToScene(posVue1) ❸ self.scale(facteur,facteur) ❹ posVue2 = self.mapFromScene(posScene) ❺ dxVue = posVue2.x() - posVue1.x() ❻ dyVue = posVue2.y() - posVue1.y() self.hScrollBar.setValue(self.hScrollBar.value()+dxVue) ❼ self.vScrollBar.setValue(self.vScrollBar.value()+dyVue)

Cette méthode est un peu compliquée et descend clairement dans les couches basses du framework.

On commence par enlever tout traitement du point d'ancrage (`QGraphicsView.NoAnchor`) ❶, sachant qu'on va réaliser ce traitement nous-

même, dans la méthode `zoom`. Dans cette méthode, on prend les coordonnées courantes du pointeur dans la vue (`posVue1`) ❷ et on les transforme en coordonnées scène qu'on mémorise dans `posScene` ❸. Après la transformation de zoom faite par `scale` ❹, on calcule `posScene` en coordonnées de la vue transformée (`posVue2`) ❺. On sait donc que, si on ne fait rien d'autre, le point sous le curseur s'est déplacé de `posVue1` à `posVue2`, suite à la transformation opérée par `scale`. Dès lors, on opère une translation de la vue, selon le vecteur `posVue2-posVue1`, pour repositionner le point `posVue2` sous le pointeur ❻. Pour faire cette translation, on utilise les barres de défilement horizontale et verticale, même si elles ne sont pas visibles ❼.

Tout cela peut paraître un peu alambiqué, mais on obtient à présent un ancrage sans défaut : un carré pointé reste pointé tant qu'on ne bouge pas le pointeur souris.

5. Zoom au clavier

Maintenant que nous avons programmé le zoom avec point d'ancrage, il est aisé de fournir des alternatives à la molette souris (une action qui peut se relever fatigante à la longue). Nous allons utiliser le clavier : la touche A fera un zoom avant et la touche Q un zoom arrière — un choix jugé confortable pour les utilisateurs de claviers AZERTY.

Pour réaliser cela, on va — sans surprise — surcharger une nouvelle méthode de QGraphicsView : `keyPressEvent` est appelée chaque fois qu'une touche est pressée ; son argument est un QKeyEvent, qui fournit les détails sur la (ou les) touche(s) enfoncée(s). Le protocole d'appel de cette méthode nous sera très utile ici : si on garde la touche enfoncée assez longtemps, `keyPressEvent` est automatiquement appelée plusieurs fois en séquence (c'est ce qu'on a l'habitude de voir dans tous les éditeurs de texte) ; on peut ainsi obtenir à peu de frais un effet de zoom continu en maintenant la touche A ou Q enfoncée.

```
…
FACTEUR_ZOOM_FIN = 1.02
…
class VueMosaiQ(QGraphicsView):
    …
    def keyPressEvent(self,keyEvent):
        key = keyEvent.key()
        if key == Qt.Key_A:
            self.zoom(FACTEUR_ZOOM_FIN)
        elif key == Qt.Key_Q:
            self.zoom(-FACTEUR_ZOOM_FIN)
        else:
            QGraphicsView.keyPressEvent(self,keyEvent)
```

Pour retrouver quelle touche a été enfoncée, on appelle la méthode `key()` ; cette méthode renvoie un code défini dans l'énuméré Qt.Key qui donne un nom symbolique à toutes les touches du clavier. Il suffit dès lors d'intercepter les touches A/Q et d'appeler la fonction zoom avec un facteur positif ou négatif, respectivement. Pour rendre les choses configurables, on a placé ce facteur dans une variable globale. La valeur choisie est petite (2 %), car on veut un effet fluide et on s'attend à des appels en répétition, en gardant les touches enfoncées. Comme déjà pratiqué, si on tombe sur une touche non reconnue par l'application, on appelle la méthode parente `keyPressEvent` pour garder tous les comportements par défaut.

À l'exécution, on peut voir que les touches A et Q ont l'effet voulu, avec un zoom

continu si on laisse la touche enfoncée. On constate aussi qu'à tout moment la zone visualisée suit bien le pointeur de la souris, selon la technique de point d'ancrage qui a été présentée.

6. Mode plein écran

Beaucoup d'applications GUI proposent un *mode plein écran*. Ce mode, qui agrandit l'application à tout l'écran en cachant la barre de titre de la fenêtre principale, est particulièrement attrayant pour des applications comportant une vue graphique et pour les plateformes mobiles.

Contrairement à ce qu'on pourrait croire, le mode plein écran est très simple à implémenter en PyQt : la classe QWidget fournit une méthode `showFullScreen`, qu'il suffit d'appeler sans aucun argument ! On peut ainsi mettre *tout* widget en plein écran, même si cela n'a pas forcément toujours du sens. Pour quitter ce mode et restaurer la taille initiale, on appelle `showNormal()` sur le widget. Comme QGraphicsView hérite de QWidget, on a tout ce qu'il faut pour notre besoin.

En ajoutant les quelques lignes qui suivent dans `VueMosaiQ.keyPressEvent`, on utilise la touche F11 pour passer en mode plein écran et Esc pour en sortir.

```
…
class VueMosaiQ(QGraphicsView):
    …
    def keyPressEvent(self,keyEvent):
        …
        elif key == Qt.Key_F11:
            self.showFullScreen()
        elif key == Qt.Key_Escape:
            self.showNormal()
        …
```

Oui… c'est aussi simple que cela !

7. Capture d'écran

Pour ajouter une dernière fonction accessible au clavier, on se propose de faire une capture de la vue dans un fichier image, ce fichier étant choisi par l'utilisateur. Le raccourci clavier pour réaliser cette opération sera Ctrl+P.

```
…
from PyQt5.QtWidgets import QFileDialog
…
class VueMosaiQ(QGraphicsView):
    …
    def keyPressEvent(self,keyEvent):
        …
        elif keyEvent.modifiers() == Qt.ControlModifier \
```

❶ and key == Qt.Key_P: (nomFichier,filtre) = QFileDialog.getSaveFileName(self, ❷ "Sauvegarder image", filter="PNG (*.png);;JPEG (*.jpg);;BMP (*.bmp)") if nomFichier: ❸ pixmap = self.grab() ❹ pixmap.save(nomFichier) ❺ …

❶ Pour détecter la combinaison Ctrl+P, on utilise la méthode `QKeyEvent.modifiers` qui indique si une touche spéciale (Ctrl, Shift, Alt, etc.) est enfoncée au moment de l'événement principal sur le clavier (ici, action sur la touche P).

❷ La méthode statique `QFileDialog.getSaveFileName` affiche une boîte de dialogue standard permettant de choisir un fichier pour sauvegarder l'image ; on a défini des filtres pour les extensions des fichiers graphiques aux formats PNG, JPEG et BMP[16].

❸ Ce test est important pour sauter tout traitement si le bouton Annuler a été pressé.

❹ La méthode `grab()` crée en mémoire une image de la vue, telle qu'elle est affichée à l'écran, sans la barre de titre. L'objet créé est une instance de QPixmap, une classe qui permet de stocker une image en mémoire pour être transformée, affichée ou sauvegarder sur fichier.

On sauvegarde l'image dans un fichier avec le nom et le chemin donné. L'instance de

❺ QPixmap est capable de reconnaître l'extension du fichier pour écrire le fichier au format requis (PNG, JPEG ou BMP).

À l'exécution, vous pouvez à présent capturer la vue de MosaiQ dans un fichier en pressant Ctrl+P. Vous constaterez que l'image produite a exactement les mêmes dimensions en pixels que la vue affichée à l'écran ; en particulier, si vous êtes passé en mode plein écran, l'image produite aura les même dimensions que celles de votre écran.

***Note** > La technique simple exposée ici fonctionne avec tout widget car la méthode `grab` est définie au niveau de la classe QWidget (et donc héritée par QGraphicsView). Pour capturer une image de toute la scène, quelle que soit la portion affichée dans la vue, il faut faire appel à la méthode QGraphicsScene.render, qui est un peu plus compliquée à mettre en œuvre.*

---

[16] Ces trois formats sont les plus répandus et ils sont supportés par la classe QPixmap. Au besoin, on pourrait ajouter aussi les formats PPM, XBM et XPM, qui sont aussi supportés par QPixmap.

8. Fragmentation continue

Revenons sur le mode de fragmentation des carrés de notre application. Nous avions initialement annoncé qu'on allait démontrer qu'une scène pouvait facilement contenir des dizaines de milliers d'éléments. Sachant que, à chaque clic, on ajoute quatre carrés et on en supprime un, on a un résultat net de seulement trois carrés par clic. Pour éviter des crampes au doigt, il est possible d'implémenter un nouveau mode de fragmentation qui accélère considérablement le remplissage de la scène.

L'idée est d'introduire un mode de fonctionnement où l'on fragmente tout carré qu'on survole avec le pointeur, sans qu'il y ait besoin de cliquer. Pour activer ce mode, on demande à l'utilisateur de garder la touche Ctrl enfoncée. On pourra ainsi fragmenter les carrés par simples déplacements de la souris.

Il existe plusieurs manières pour programmer cela. On pourrait par exemple changer l'implémentation de `CarreMosaiQ.hoverEnterEvent`, en appelant `self.fragmente()` si la touche Ctrl est enfoncée. Toutefois, ceci ne donnerait pas tout à fait le résultat escompté : si on enfonce Ctrl alors que le pointeur est déjà au-dessus d'un carré, ce carré ne serait pas fragmenté tant que le pointeur ne l'a pas quitté et ne l'a pas survolé à nouveau.

Une meilleure technique, un rien plus compliquée, consiste à détecter les mouvements du pointeur au niveau de la vue et à agir en conséquence. À cette fin, nous introduisons une nouvelle méthode, `mouseMoveEvent`, qui sera appelée à chaque mouvement du pointeur souris dans la vue ; l'argument de cette méthode est un QMouseEvent qui donne notamment les coordonnées du pointeur dans la vue (méthode `pos()`)[17]. Voici comment implémenter cette méthode :

```
…
class VueMosaiQ(QGraphicsView):
    …
    def mouseMoveEvent(self,mouseEvent):
        QGraphicsView.mouseMoveEvent(self,mouseEvent)
```

❶ if QApplication.keyboardModifiers() == Qt.ControlModifier: ❷ item = self.itemAt(mouseEvent.pos()) ❸ if item is not None and isinstance(item,CarreMosaiQ): ❹ item.fragmente()

❶ On appelle la méthode parente pour ne pas casser le comportement par défaut.

❷ Comme on n'a pas un événement clavier ici (QKeyEvent), on utilise la méthode statique `QApplication.keyboardModifiers`, qui permet à n'importe quelle méthode de connaître l'état courant des touches spéciales du clavier ; ici, on teste que la touche Ctrl est enfoncée à l'instant précis du mouvement du pointeur. Si c'est le cas, on continue le traitement.

❸ Il faut déterminer si le pointeur survole actuellement un élément et, si oui, retrouver cet élément. Ceci est nécessaire car on est ici dans la classe vue (VueMosaiQ), pas dans la classe élément (CarreMosaiQ). La méthode `itemAt` héritée de QGraphicsView renvoie l'élément présent aux coordonnées vue données ou `None` si aucun élément n'est présent à ces coordonnées. On lui passe la position de l'événement QMouseEvent reçu, qui est bien une position en coordonnées vue.

❹ Il reste à tester qu'on survole bien un élément et qu'on a bien un carré fragmentable, c'est-à-dire une instance de CarreMosaiQ. Si oui, on fragmente cette instance ! Notons que, dans notre application actuelle, la deuxième partie du test est inutile, car on n'a pas d'autres éléments sur la scène ; il est cependant prudent de laisser ce contrôle pour se protéger lors des évolutions futures (programmation défensive).

Une fois cette méthode ajoutée, vous pouvez commencer à piler les carrés en petits morceaux, de manière industrielle ! Vous pouvez ainsi dessiner avec une sorte de pinceau magique et cacher, si le cœur vous en dit, des messages subliminaux seulement visibles en zoomant…

Figure 12.6 : *MosaiQ v0.4 (fragmentation continue)*

---

[17] En fait, le déplacement de la souris (ou son équivalent) est échantillonné en une séquence de positions, pour chacune desquelles *mouseMoveEvent* sera appelée. Plus le mouvement est rapide, plus les positions consécutives seront éloignées l'une de l'autre.

9. Panneau d'information

La nouvelle fonction de fragmentation permet de créer un grand nombre de carrés sur la scène, mais on ne connaît pas exactement ce nombre. Dans cette section, nous allons présenter comment on peut afficher un panneau d'information au-dessus de la vue (ce qu'on appelle en anglais un *overlay*) ; ce panneau présentera trois informations textuelles (voir Figure 12.7) :

- le nombre de carrés actuellement sur la scène ;
- la taille du carré sous le pointeur, en unités scène (nombre en virgule flottante) ;
- la taille de ce même carré, en unités vue (nombre entier représentant des pixels).

Ces informations seront mises à jour en temps réel, selon les actions faites (fragmentation, déplacement pointeur, zoom). Par ailleurs, pour laisser le choix à l'utilisateur, ce panneau ne sera affiché qu'après avoir pressé la touche F1 et idem pour le faire disparaître.

Cette nouvelle fonction est un peu plus compliquée à implémenter que ce qui précède, mais on va décomposer la programmation en étapes. Comme souvent, il existe plusieurs manières de procéder à ce développement. Ce qui est présenté ici tente de préserver autant que possible l'indépendance entre les différentes classes ; ainsi, on fera en sorte que *seule* la classe VueMosaiQ connaisse l'existence du panneau affiché ; donc, ni CarreMosaiQ ni SceneMosaiQ n'auront à le notifier explicitement.

Commençons par créer une nouvelle classe, PanneauInfo, qui hérite de QFrame et dont l'instance sera placée au-dessus de notre widget vue. Les informations seront affichées sur ce panneau dans trois instances de QLabel, alignées verticalement.

```
…
class PanneauInfo(QFrame):

    def __init__(self,parent,scene):
        QFrame.__init__(self,parent)
        self.vue = parent
```

❶ self.scene = scene ❷ self.setFrameStyle(QFrame.Box) ❸ self.setStyleSheet("QFrame {background-color: white;" \ + "color: darkblue}") ❹ self.resize(180,75) ❺ self.labelNbCarres = QLabel(self) self.labelTailleCarreScene =

QLabel(self) self.labelTailleCarreVue = QLabel(self) self.vBoxLayout = QVBoxLayout(self) self.vBoxLayout.addWidget(self.labelNbCarres) self.vBoxLayout.addWidget(self.labelTailleCarreScene) self.vBoxLayout.addWidget(self.labelTailleCarreVue)

❶ Comme on le voit, le parent du panneau d'information est la vue, dont on aura besoin d'appeler certaines méthodes ; pour rendre l'intention explicite, on crée un attribut appelé `vue`.

❷ Le panneau aura besoin aussi de connaître la scène, qu'on récupère de l'argument donné au constructeur.

❸ On définit l'apparence du QFrame pour avoir un cadre noir via le style `QFrame.Box`.

❹ Pour définir la couleur de fond du panneau et la couleur du texte, la méthode recommandée est d'utiliser la syntaxe des feuilles de style de Qt (*style sheets*), qui est très proche de celle des CSS. Ce style est hérité par les trois QLabel enfants. On peut omettre cette instruction, mais, alors, le fond du panneau est transparent, ce qui rend l'information moins lisible, sachant que les carrés colorés de la vue seront en arrière-fond.

❺ Ce qui suit utilise les techniques présentées au chapitre Créer une première application pour créer les trois QLabel et les aligner verticalement sur le panneau.

Pour afficher les informations nécessaires sur le panneau, on a besoin à chaque instant de répondre à deux questions : combien y a-t-il de carrés sur la scène ? et quel est le carré actuellement sous le pointeur souris ? Il existe plusieurs techniques pour y répondre. L'une d'elle, simple et fiable, consiste à définir deux attributs de classe à CarreMosaiQ (donc communs à toutes les instances) et à laisser les méthodes de cette classe s'occuper de la gestion de ces attributs.

```
…
class CarreMosaiQ(QGraphicsRectItem):
    …
    nbInstances = 0
```

❶ instanceSurvolee = None ❷ def __init__(self,x,y,c): … CarreMosaiQ.nbInstances += 1 ❸ def hoverEnterEvent(self,event): … CarreMosaiQ.instanceSurvolee = self ❹ def hoverLeaveEvent(self,event): … CarreMosaiQ.instanceSurvolee = None ❺ def fragmente(self): … CarreMosaiQ.nbInstances -= 1 ❻

❶ `nbInstances` est un attribut de classe qui contiendra à tout moment le nombre d'instances de CarreMosaiQ (zéro au début).

`instanceSurvolee` est un attribut de classe qui contiendra à tout moment l'instance de CarreMosaiQ actuellement sous le pointeur souris (`None` si aucune).

❷ ***Note** > Comme un attribut de classe est, par définition, unique au sein d'un programme, la présente approche fait l'hypothèse qu'on ne crée les instances que sur une seule scène ; dans le cas contraire, il faudrait placer `nbInstance` et `instanceSurvolee` en attributs d'instance de SceneMosaiQ (ce qui nécessiterait une gestion un peu moins simple).*

❸ Le constructeur construit une nouvelle instance, on incrémente donc le compteur.

❹ L'appel de la méthode `hoverEnterEvent` signale que le pointeur vient de rentrer dans la zone au-dessus du carré courant : on mémorise cette instance.

❺ L'appel de la méthode `hoverLeaveEvent` signale que le pointeur vient de quitter la zone au-dessus du carré courant : on met `None` pour indiquer qu'on n'a momentanément aucun carré sous le pointeur.

❻ Comme la fragmentation supprime le carré courant, on décrémente le compteur d'une unité. Notons qu'il ne faut pas compter les quatre instances créées, car elles sont déjà prises en compte dans le constructeur `CarreMosaiQ.__init__`.

On peut à présent à tout moment retrouver le nombre de carrés sur la scène via l'attribut `CarreMosaiQ.nbInstances` et le carré sous le pointeur via `CarreMosaiQ.instanceSurvolee`. Fort de ce mécanisme, nous pouvons compléter l'implémentation de PanneauInfo pour exploiter ces deux attributs et afficher l'information attendue.

```
…
class PanneauInfo(QFrame):

    def __init__(self,parent,scene):
        …
        scene.changed.connect(self.sceneChanged)
```

❶ def sceneChanged(self,regions): txtNbCarres = "%5d" % CarreMosaiQ.nbInstances ❷ if CarreMosaiQ.instanceSurvolee is None: ❸ txtTailleScene = '---' txtTailleVue = '---' else: tailleCarreScene = CarreMosaiQ.instanceSurvolee.taille() ❹ txtTailleScene = '%e' % tailleCarreScene tailleCarreVue = CarreMosaiQ.instanceSurvolee \ .tailleDansVue(self.vue) ❺ txtTailleVue = '%4d pixels' % tailleCarreVue self.labelNbCarres.setText("# carrés: "+txtNbCarres) ❻ self.labelTailleCarreScene.setText("taille carré scène: " \ + txtTailleScene) self.labelTailleCarreVue.setText("taille carré vue: "+txtTailleVue) class CarreMosaiQ(QGraphicsRectItem): … def taille(self): return self.rect().width() ❼ def tailleDansVue(self,vue): return vue.mapFromScene(self.taille(),0).x() - vue.mapFromScene(0,0).x() ❽

❶ Pour activer la mise à jour du panneau, on connecte le signal `changed` de la scène à une nouvelle méthode slot. Ceci permet de réagir à deux événements qui modifient la scène : la fragmentation (suppression/création de carrés) et le changement de carré sous pointeur, puisqu'on a programmé une surbrillance des carrés par changement de brosse. Le seul souci est qu'on ne capte pas les zooms, qui sont des événements sur la vue (donc, pas sur la scène) ; ces événements sont importants, car ils nécessitent de changer le champ taille carré vue, donnant la dimension du carré en pixels. Nous verrons un peu plus bas une petite astuce pour arranger cela dans VueMosaiQ.

❷ On formate le nombre d'instances comme un nombre entier à cinq chiffres.

❸ Si aucun carré n'est survolé, on définit des symboles signifiant l'absence de donnée.

❹ Sinon, on récupère la taille du carré survolé en unités scène (`float` Python), en utilisant la nouvelle méthode `taille` ajoutée à la classe `CarreMosaiQ` (voir ❼) : on formate ce nombre, qui peut être très grand en notation scientifique.

On demande au carré survolé par le pointeur de calculer sa taille en pixels. À cette

❺ fin, on a créé une nouvelle méthode `tailleDansVue` dans la classe `CarreMosaiQ` (voir ❽) ; comme `CarreMosaiQ` ne connaît que les coordonnées scène, on passe en argument la vue pour laquelle on veut obtenir la taille.

❻ Une fois qu'on a les trois informations bien formatées dans des chaînes de caractères, on les affiche dans les QLabel correspondants. Idéalement, on aurait pu créer trois QLabel supplémentaires avec les parties statiques des messages ; nous ne l'avons pas fait pour ne pas compliquer l'exemple (besoin de layouts supplémentaires).

❼ C'est comme cela qu'on récupère la taille du carré dans le repère de la scène ; la valeur obtenue est celle de l'argument c passé au constructeur.

❽ Pour calculer la taille du carré en unités vue (pixels), on utilise la méthode `mapFromScene` pour faire des conversions de coordonnées scène en coordonnées vue ; on mesure la distance vue sur l'axe des X entre le point origine et un point situé à sa droite, à la distance scène égale à la dimension du carré ; le résultat est un nombre entier donnant la distance en pixels. Attention : ceci ne marche que s'il n'y pas de rotation sur la vue, ce qui est le cas ici.

On a à présent une classe PanneauInfo prête à l'emploi. On l'instancie comme enfant de VueMosaiQ : elle apparaîtra donc au-dessus de cette vue, toujours à la même position relative. Voici les changements à faire dans la classe VueMosaiQ :

```
…
class VueMosaiQ(QGraphicsView):

    def __init__(self,scene):
        …
        self.panneauInfo = PanneauInfo(self,scene)
```

❶ self.panneauInfo.move(20,20) ❷ self.panneauInfo.setVisible(False) ❸ def zoom(self,facteur): … self.scene().changed.emit(None) ❹ def keyPressEvent(self,keyEvent): … elif key == Qt.Key_F1: ❺ self.panneauInfo.setVisible(not self.panneauInfo.isVisible()) …

❶ On crée le panneau en tant qu'enfant de la vue ; on lui passe la scène en argument, comme requis par le constructeur.

❷ Le panneau est placé aux coordonnées (20,20) de la vue, donc dans le coin supérieur gauche. Il y restera fixé indéfiniment et suivra les mouvements de la fenêtre vue.

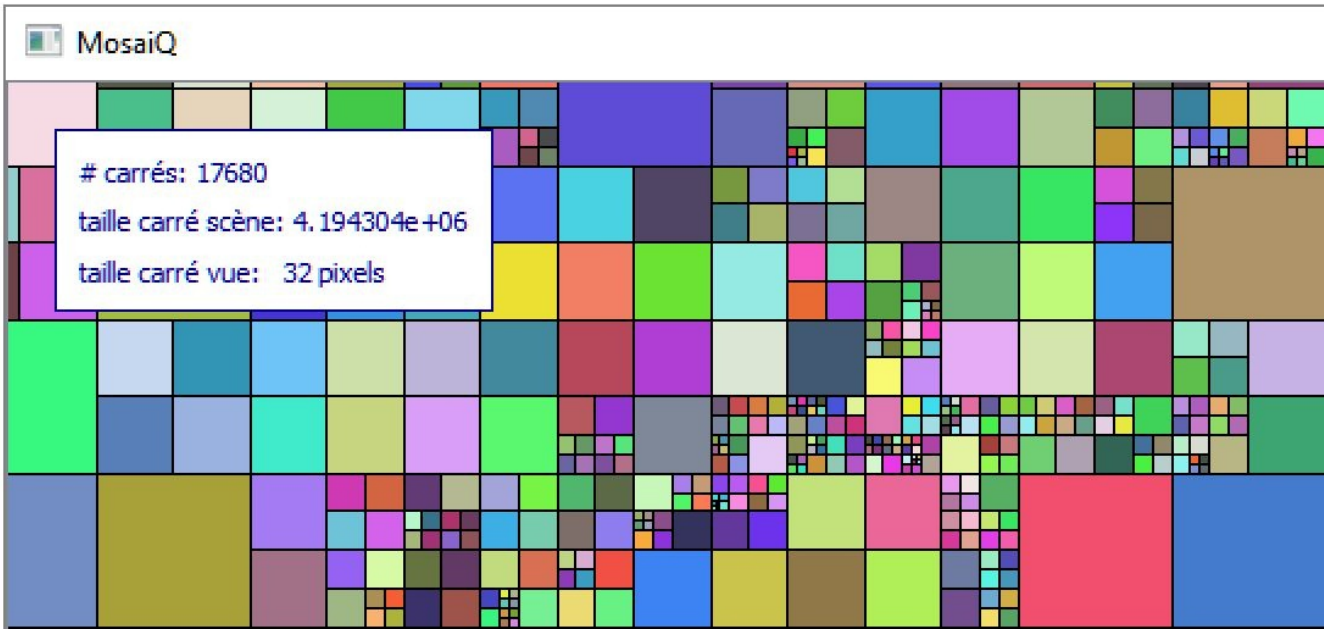
❸ On cache initialement le panneau, car on veut le faire apparaître seulement à la demande, en pressant la touche F1.

❹ Voici l'astuce annoncée un peu plus haut, dans la description de PanneauInfo : si un zoom est opéré, on fait émettre à la scène le signal `changed` ; ceci aura pour effet d'appeler `PanneauInfo.sceneChanged` pour rafraîchir les informations sur le panneau.

❺ On réagit à la touche F1 comme une bascule : si le panneau est invisible, on le rend visible et inversement.

Vous pouvez à présent exécuter l'application et activer le panneau via F1. Vous pouvez constater que le panneau se met bien à jour à chacune de vos actions, y compris la mise à jour de taille carré vue lors des zooms.

Figure 12.7 : *MosaiQ v1.0 (panneau d'information)*



Vous voyez à présent des informations intéressantes sur la scène et la vue ; vous

observez entre autres que le nombre de carrés augmente très vite grâce à la fragmentation continue (touche Ctrl) ; vous pouvez aussi vous rendre compte des différences d'échelle entre coordonnées scène et vue.

***Note** > La technique utilisée ici (construction du panneau une fois pour toute et affichage via `setVisible`) est bien adaptée dans ce contexte car le panneau est très simple. Pour un panneau plus complexe, qui aurait par ailleurs une bonne probabilité de ne pas être affiché au cours d'une session, vous pourriez choisir de construire le panneau seulement quand l'utilisateur le requiert. Par ailleurs, avec cette approche, il serait éventuellement opportun d'utiliser le patron de conception Singleton pour ne faire la construction qu'une seule fois.*

10. Limitation du zoom

Comme signalé lors de l'implémentation de la fonction zoom, il existe une limite au-delà de laquelle l'application réagit bizarrement — on pourrait même dire qu'elle devient *instable*. Ce n'est pas très étonnant sachant que, dans MosaiQ, on balaie plusieurs ordres de grandeur ; si les nombres ont une représentation de taille fixe en mémoire, il y a forcément des limites aux coordonnées qu'on peut manipuler.

Ce qui serait souhaitable, ce serait de détecter le problème *avant* qu'il n'arrive et, le cas échéant, bloquer la fonction zoom. Pour faire cela sans devoir trop réfléchir, on pourrait suivre une approche empirique : par essais, on évalue le facteur de zoom à partir duquel on constate que les problèmes surviennent ; puis, à partir de cette constante, on code un test qui bloque la fonction zoom.

Cette approche empirique n'est pas la meilleure, car la constante obtenue est imprécise et pourrait même changer d'un environnement d'exécution à l'autre. Heureusement, il existe une autre approche : elle exige un peu de réflexion, mais elle est plus fiable et aussi plus simple à implémenter. L'élément principal à comprendre est que la limitation provient de la classe QGraphicsView : les coordonnées entières sont limitées à la plage `[INT_MIN, INT_MAX]`, où `INT_MIN` et `INT_MAX` sont des constantes dépendant du système. Ces constantes sont en fait définies au niveau du C++, le langage natif de Qt ; `INT_MAX` vaut typiquement 65535 ($2^{15}-1$) ou 2147483647 ($2^{31}-1$). Pour s'assurer qu'il n'y ait pas de débordement, il suffit de veiller à ce qu'aucune coordonnée vue n'excède cette valeur `INT_MAX`. À cette fin, on peut se référer à la taille du plus grand carré — celle du carré initial, donc — pour obtenir une coordonnée maximale. Partant de ce principe, il est possible de détecter précisément quand la limite est atteinte et bloquer le zoom si nécessaire. Voici les quelques lignes à ajouter dans VueMosaiQ (on opère bien sûr au niveau de la méthode `zoom`, pour capter aussi bien le zoom à la souris que le zoom au clavier) :

```
…
import struct
INT_MAX = 2**(struct.Struct('i').size*8-1) - 1
```

❶ … class VueMosaiQ(QGraphicsView): … def zoom(self,facteur): if facteur < 0.0: facteur = -1.0 / facteur else: ❷ coordMax = self.scene().carreMosaiQUnivers.tailleDansVue(self) ❸ if coordMax*facteur >= INT_MAX: ❹ QApplication.beep() ❺ return …

❶ Comme on n'a pas accès aux constantes C++, on utilise une astuce pour obtenir la valeur de `INT_MAX` : le module Python `struct` permet de calculer la taille occupée par un nombre entier, en octets. De là, on peut déduire la valeur cherchée et la stocker dans une variable globale `INT_MAX`. Attention, cette valeur est différente de la constante Python `sys.maxsize`.

❷ On entre dans la partie de contrôle. Le `else` assure qu'on est bien dans le cas *zoom avant* (facteur donné positif).

❸ On calcule la coordonnée maximale dans la vue courante en appelant la méthode `tailleDansVue` (voir ❽) sur le carré initial ; ce dernier est mémorisé au niveau de la scène, dans l'attribut `carreMosaiQUnivers`. D'un appel à l'autre, le résultat sera différent, sachant que la transformation de coordonnées opérée par la vue sera différente.

❹ Avec l'expression `coordMax*facteur`, on anticipe quelle serait la coordonnée maximale sur la vue après zoom. Si cette valeur est supérieure à `INT_MAX`, on sait que le zoom opéré avec la fonction `scale` va provoquer un débordement.

❺ Si un débordement est anticipé, on émet un signal sonore (dépendant de l'OS) pour avertir l'utilisateur qu'on a atteint la limite ; on appelle `return` pour sortir de la fonction `zoom` sans faire de traitement normal.

11. Conclusion

Dans ce chapitre et dans les deux précédents, vous avez pu découvrir comment programmer avec le framework Graphics View en PyQt. Vous avez appris les concepts de scène, élément et vue ; vous avez utilisé les classes QGraphicsScene, QGraphicsItem et QGraphicsView comme briques de base pour construire des applications graphiques simples. Vous avez constaté que beaucoup de fonctions standards pouvaient s'implémenter facilement, parfois même en une seule ligne de code. Enfin, si vous étiez soucieux des performances et des temps de réponse, vous avez pu constater que le couple Python/PyQt s'en sort finalement très bien.

Pour terminer, voici quelques conseils pour aller plus loin.

- Avant de programmer une fonction, lisez la documentation ou consultez un moteur de recherche pour voir si cette fonction n'est pas déjà proposée par une classe du framework. C'est particulièrement vrai si la fonction est simple ou standard : nous l'avons vu notamment avec le glisser/déposer et la sélection par lasso.

- On a présenté les différents systèmes de coordonnées (scène/vue) ainsi que les coordonnées relatives parent/enfants. Si cela peut sembler compliqué au début, on constate qu'à l'usage ces différents systèmes *simplifient* l'écriture des programmes. Il est très rare de devoir programmer des transformations de coordonnées à la main : il existe un grand choix de méthodes préfixées par `map` (`mapFromScene`, `mapToScene`, etc.) qui font ce travail.

- Pour une application avancée, avec des interactions utilisateur complexes, il est essentiel de bien réfléchir à la conception. Beaucoup de problèmes peuvent avoir une solution simple en identifiant bien les classes ainsi que les relations d'héritage et de composition. Dans une application bien conçue, les corps des méthodes événementielles surchargées (`mousePressEvent`, `mouseMoveEvent`, etc) sont très simples et comportent peu de branches `if`. Typiquement, pour les éléments de la scène, on définira plusieurs classes qui héritent de QGraphicsItem ; chaque élément réagira à sa manière aux actions de l'utilisateur. Le framework est conçu pour être étendu ; il ne faut pas s'en priver, surtout en Python où les mécanismes orientés objet sont simples à maîtriser.

Développement d'une application avec Qt Quick

13

Premiers pas avec Qt Quick

Niveau : débutant

Objectifs : développer une interface graphique Qt Quick simple avec un formulaire

Qt Quick est le dernier arrivé dans la famille Qt pour la réalisation d'interfaces graphiques. Par rapport aux technologies existantes, il change complètement de paradigme : il *déclare* des liens entre des composants, alors que les autres associent généralement des actions à des stimuli. Pour ce faire, il recourt à un autre langage, au QML, spécifiquement prévu pour cette utilisation ; il ressemble beaucoup au CSS et à d'autres langages de feuilles de styles comme Sass ou Less.

Parmi ses avantages, il est très adapté pour la réalisation rapide de prototypes, sans nécessiter de grandes connaissances en programmation ; de plus, sa proximité avec les technologies web facilite sa prise en main par des profils plus orientés vers la conception d'interfaces que la programmation proprement dite. Avec les années de maturité, Qt Quick peut maintenant être utilisé pour des applications de bien plus grande taille que des prototypes : l'outil est notamment conseillé pour le développement d'applications avec ESRI ArcGIS, pour tout ce qui concerne la cartographie et les systèmes d'informations géographiques au sens large.

Cet avantage est aussi très probablement son très grand inconvénient : il nécessite d'apprendre un nouveau langage de programmation pour décrire les interfaces graphiques. Outre la barrière à l'apprentissage, cela complique les communications avec la partie Python de toute application : il devient nécessaire d'écrire du code pour faire le lien.

Contrairement à Python, QML n'a pas une syntaxe très évoluée : principalement, il s'agit d'imbriquer des composants (un rectangle, un bouton, etc.) et de définir leurs propriétés (couleur, texte affiché, position, etc.). Les valeurs de ces propriétés sont données par du code JavaScript, entièrement intégré à QML.

L'objectif de ce module sera de développer une application de gestion d'une bibliothèque, extrêmement similaire à celle développée dans le module Qt Widgets. Toutefois, pour expliquer la syntaxe et les concepts de base de Qt Quick, nous allons commencer par quelques exemples bien plus simples.

1. Découverte de Qt Creator

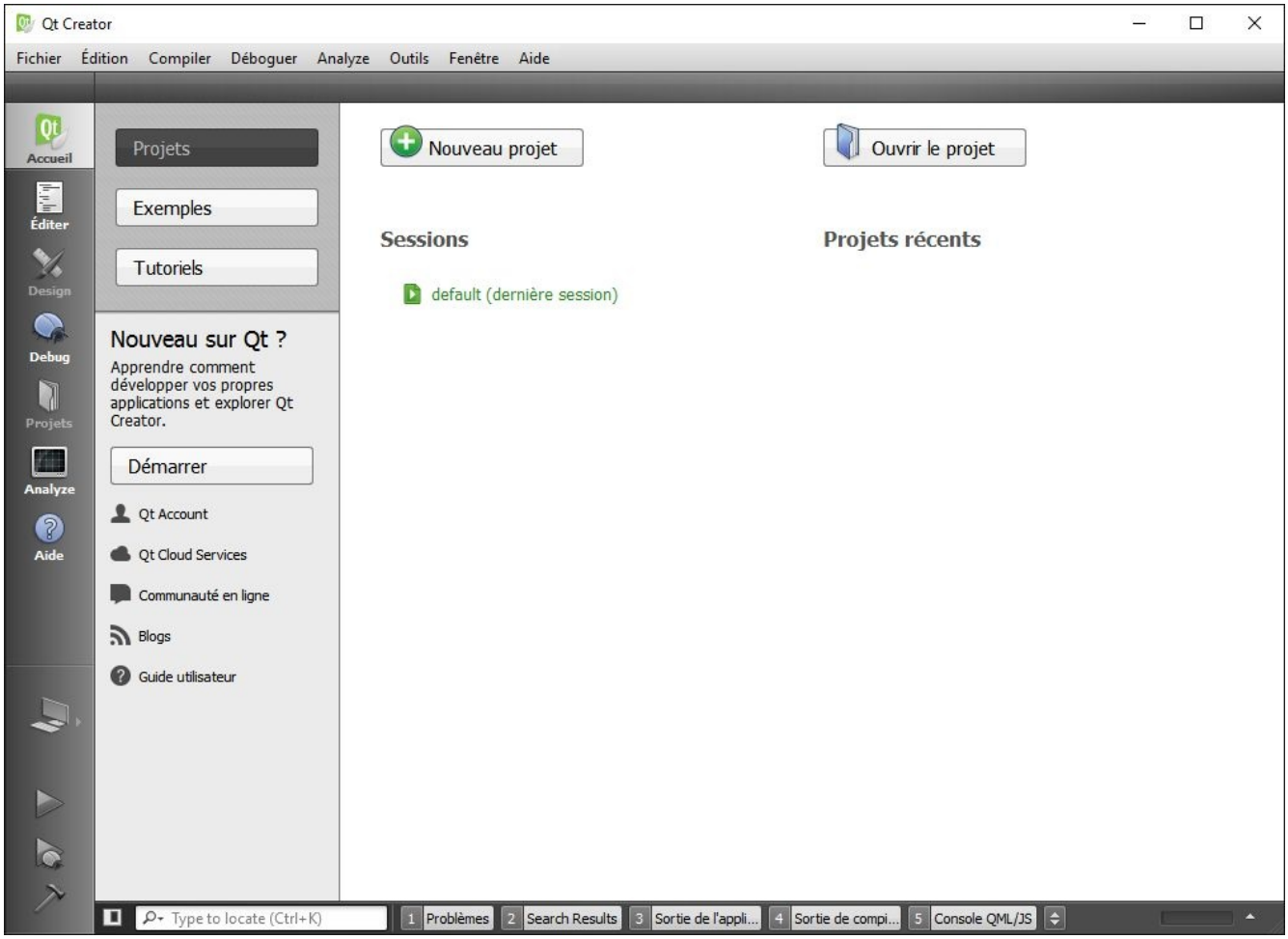
La principale difficulté pour utiliser Qt Quick avec PyQt réside dans l'absence d'environnement de développement intégré (comme eric6) qui soit bon à la fois pour le côté Python et pour le côté Qt Quick. Actuellement, seul Qt Creator, l'environnement de référence pour Qt en C++, propose un éditeur de code QML performant, une intégration des mécanismes de lancement et de débogage des applications Qt Quick et un éditeur visuel intégré. Cependant, il n'est pas adapté au développement Python, même si, au besoin, il peut dépanner avec la configuration nécessaire. Les environnements de développement Python, quant à eux, ne disposent d'aucunes fonctionnalités de base pour Qt Quick (pas même la coloration syntaxique).

Toute la partie Qt Quick utilisera donc Qt Creator comme environnement de référence. Il est installé en même temps que Qt par les installeurs officiels (donc pas par PyQt ou un gestionnaire de paquets), disponibles gratuitement sur le site de Qt.

1.1. Aperçu de l'interface

Une fois Qt installé, Qt Creator est disponible comme application dans l'environnement graphique. Au premier lancement, il peut paraître lourd et peu intuitif ; cependant, avec un peu d'accompagnement, il devient simple à utiliser.

Figure 13.1 : *Premier lancement de Qt Creator : l'interface*



Qt Creator est organisé autour d'une série de *modes* : à sa première ouverture, il lance le mode Accueil, pourvu d'un ensemble de liens notamment vers des tutoriels et la documentation ; il fournit également un accès rapide aux derniers projets ouverts. Parmi les autres modes, deux sont plus particulièrement utiles : Éditer, qui présente une vue classique d'éditeur de code avec coloration syntaxique, et Aide, donnant accès directement à toute la documentation de Qt, notamment celle de Qt Quick. La barre de gauche permet de passer d'un mode à l'autre, simplement en cliquant sur son icône.

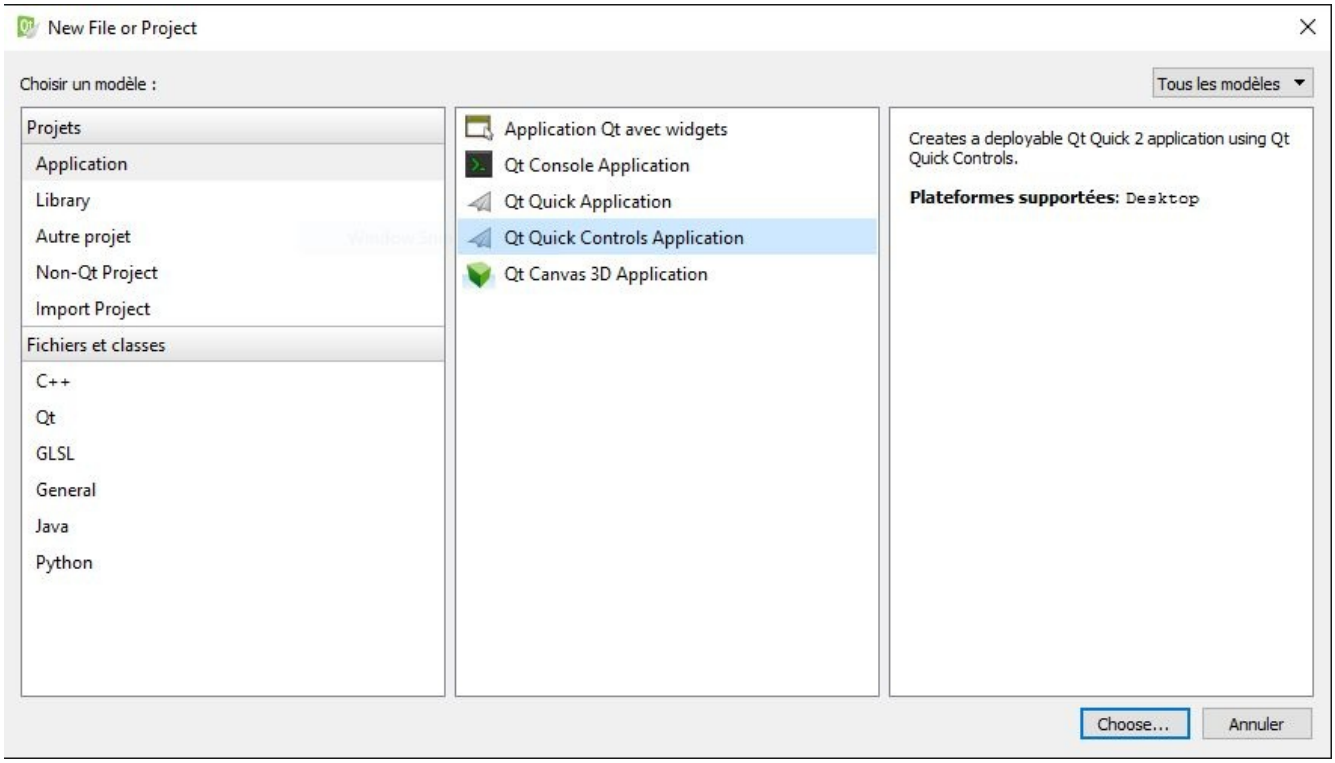
1.2. Cycle de vie d'un projet Qt Quick

L'environnement de développement s'oriente, comme la plupart des EDI, autour de la notion de *projet*, c'est-à-dire un ensemble de fichiers formant une application. La création d'un projet se fait par le menu Fichier > Nouveau fichier ou projet (Ctrl+N).

Dans la fenêtre qui s'ouvre, les types de projets qui correspondent à une application

Qt Quick sont Qt Quick Application (une application Qt Quick 2 simple), Qt Quick Controls Application (avec des contrôles de base très utiles) et Qt Canvas 3D Application (pour utiliser le module Qt Canvas 3D). Pour le moment, nous choisirons Qt Quick Controls Application.

Figure 13.2 : *Fenêtre de création d'un nouveau projet*



Ensuite, Qt Creator demande le nom du projet (prendre le nom de l'application est une bonne idée pour s'y retrouver) et l'emplacement des fichiers à créer. Ces choix n'auront aucun impact sur le développement, si ce n'est l'emplacement physique des fichiers.

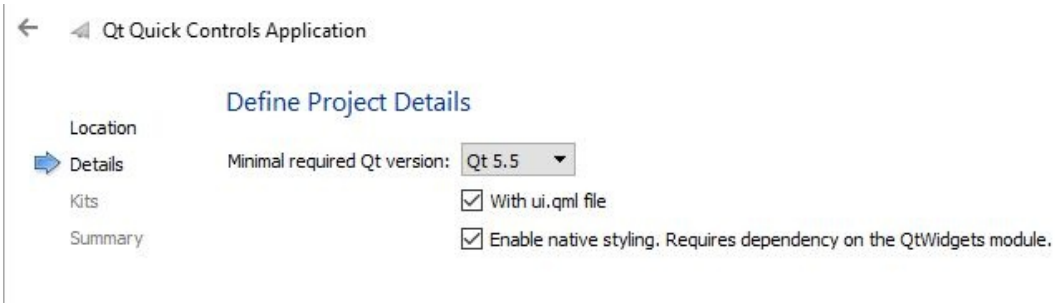
L'écran suivant demande la version minimale de Qt à utiliser pour le projet : la valeur par défaut convient à la plupart des cas, à moins que les fonctionnalités les plus récentes soient utilisées[18].

Note *> Notamment, dès la version 5.4, l'assistant propose par défaut de créer un fichier* `.ui.qml` *: il s'agit d'un modèle de code qui peut directement être modifié par l'éditeur visuel, accessible dans le mode Design. Ces fonctionnalités ne seront pas utilisées par la suite.*

Attention *> Ne confondez pas le numéro de version de Qt avec celui de PyQt : les numéros de version de PyQt ne correspondent pas toujours à ceux de Qt, même si c'est très souvent le cas. En cas de doute, la correspondance est indiquée dans le*

nom du fichier wheel ou de l'installateur de PyQt (par exemple, `PyQt5-5.8-5.8.0-cp34.cp35.cp36.cp37-abi3-macosx_10_6_intel.whl` ou `PyQt5-5.5.1-gpl-Py3.4-Qt5.5.1-x64.exe`).

Figure 13.3 : *Sélection de la version*



Le dernier écran concerne la sélection d'un *kit*. Il s'agit de l'installation de Qt qui sera utilisée ; généralement, une seule installation de Qt sera utilisée dans un projet. Dans tous les cas, les valeurs par défaut sont recommandées.

Figure 13.4 : *Sélection du kit*

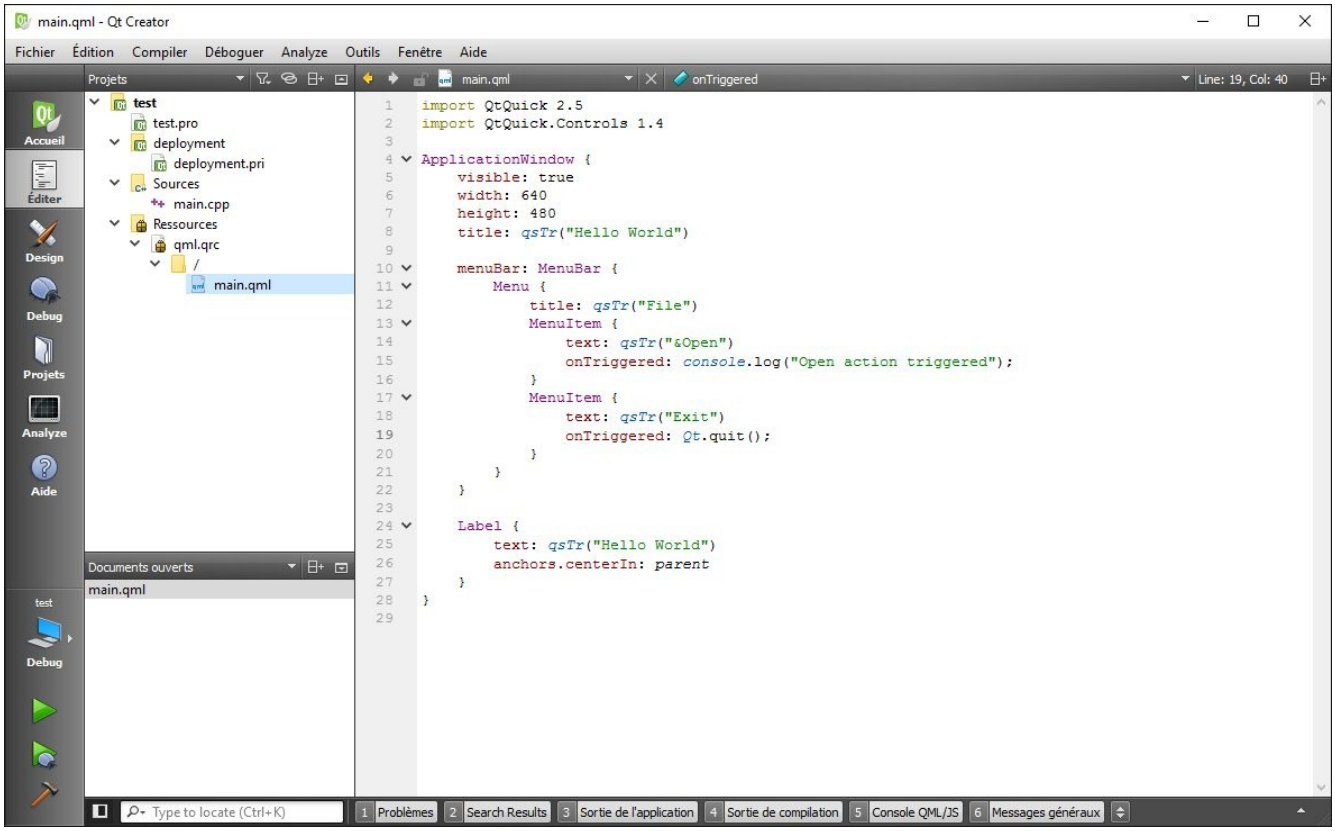
Finalement, en guise de récapitulatif, le dernier écran liste les fichiers qui seront créés et leur emplacement, il propose aussi de les ajouter dans un gestionnaire de versions, comme Git ou Mercurial. Un clic sur Terminer lance la création du projet.

In fine, le projet est bien créé, avec le document QML principal tout juste constitué ouvert dans le mode Éditer. L'explorateur de projets sur la gauche montre une série de fichiers qui ne sont pas très intéressants et dont la seule utilité est de lancer l'application. Le code QML est contenu dans le fichier `main.qml` et est relativement simple. Il est disponible dans la section `Ressources` de l'arborescence.

Note *> Si vous avez demandé la création d'un fichier `.ui.qml` dans l'assistant de création de projet, cet exemple sera plus compliqué et comprendra un fichier*

`MainForm.ui.qml`, exploité dans le fichier `main.qml`.

Figure 13.5 : *Le projet une fois créé dans Qt Creator*



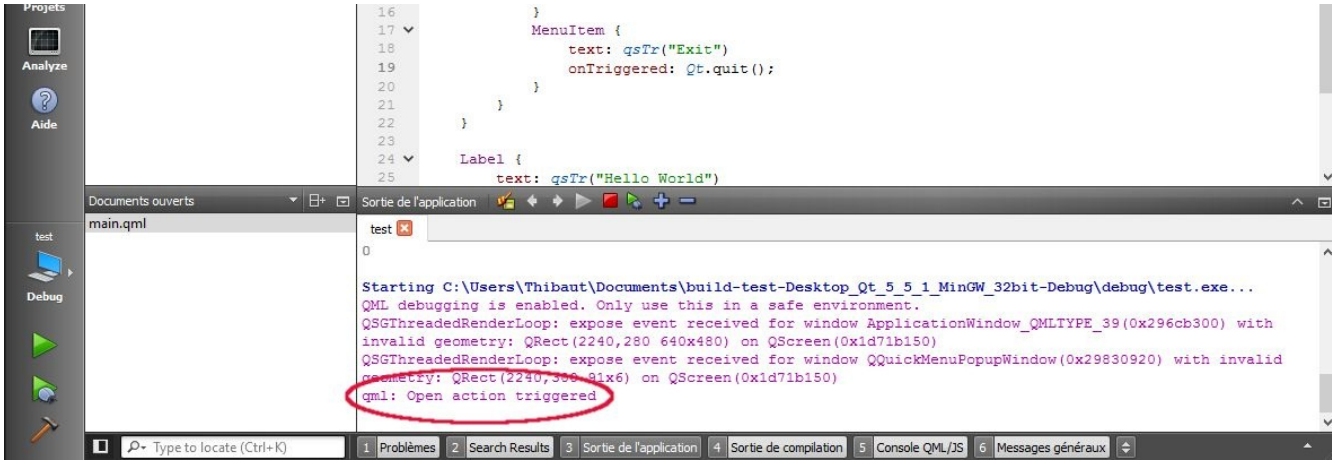
Pour lancer cet exemple, il suffit de cliquer sur la flèche verte en bas à gauche. La première exécution prend un certain temps, vu qu'il est nécessaire de compiler le code C++ qui lance l'interface Qt Quick. Une fenêtre assez rustique s'ouvre, avec une barre de menus présentant deux entrées et l'habituel texte *Hello World* dans la partie centrale.

***Note** > Pour lancer une application Qt Quick n'exploitant que les capacités de base de l'environnement (c'est-à-dire tous les exemples jusqu'au chapitre d'intégration avec Python), il est aussi possible d'utiliser l'exécutable `qmlscene`, livré avec Qt et PyQt.*

Figure 13.6 : *L'application d'exemple*

Le code indique que l'action `console.log()` doit être initiée si l'entrée File > Open est sélectionnée. Lorsque l'utilisateur clique sur cette entrée du menu, le texte indiqué en paramètre (dans les parenthèses après `console.log`) s'affiche dans la console de Qt Creator, en bas de la fenêtre. Il s'agit d'une technique de débogage très simple à mettre en œuvre quand une application ne réagit pas exactement comme attendu.

Figure 13.7 : *La console de Qt Creator et un message affiché depuis le code QML*



L'ajout de fichiers au projet peut se faire par les menus (Fichier > Nouveau fichier ou projet…), par un raccourci clavier (Ctrl+N) ou directement par un clic droit dans l'arborescence des fichiers. Pour un développement Qt Quick, deux types sont particulièrement intéressants : les fichiers QML QML File (Qt Quick 2) et les fichiers JavaScript JS File.

[18] Pour Qt 5.5 et Qt Creator 3.6, les versions affichées ne montent pas plus haut que Qt 5.3. Ce n'est pas le cas avec les versions plus récentes de Qt Creator.

2. Document QML

Pour débuter avec QML et comprendre les principes généraux, l'idéal est de partir d'un code plus simple que celui de l'exemple fourni par Qt Creator. Ce premier exemple ne montre qu'un rectangle contenant du texte, mais il illustre déjà un principe essentiel de QML : *un document QML forme une hiérarchie d'objets*, ce qui correspond au concept de *composition* en programmation orientée objet.

De manière générale, Qt Quick s'organise autour de deux concepts orientés objet : un nouveau composant *hérite* d'un autre (dans l'exemple qui suit, l'interface hérite d'un rectangle) ; il peut agréger une série d'autres composants par *composition*. Par l'héritage, l'interface aura toutes les propriétés d'un rectangle, en en adaptant quelques propriétés (taille, couleur). La composition remplit le contenu du rectangle.

***Note** > Les codes sources de ce chapitre sont disponibles dans le dossier `bases`. Ce premier exemple se trouve dans le sous-dossier `rectangle`*  .

```
import QtQuick 2.5

❶ Rectangle { ❷ width: 200 ❸ height: 100 color: "lightblue" Text { ❹
anchors.centerIn: parent ❺ text: "Hello World" color: "darkblue" } }
```

❶ La première étape consiste à importer les blocs de base qui seront utilisés pour construire l'application, tout comme l'instruction `import` de Python. Ici, l'importation se fait sur tous les éléments de base de Qt Quick, contenus dans le module `QtQuick`. Généralement, les codes sources présenteront deux directives d'importation : les composants Qt Quick de base, plus `QtQuick.Controls` pour grandement faciliter la création d'une interface.

***Note** > Contrairement à Python, pour Qt Quick, les modules disposent d'un numéro de version. Ainsi, il est possible de gérer des évolutions incompatibles avec les versions précédentes d'un module sans casser le code existant.*

Le document QML présenté a pour racine une instance du composant `Rectangle`, déjà défini dans le module `QtQuick`. Ici, le rectangle est la racine du document QML : il correspondra directement à la fenêtre ; ainsi, les dimensions du rectangle seront celles de la fenêtre.

Note *> Avec Qt Quick, la convention est que les composants ont toujours un nom qui commence par une capitale. Cette remarque sera cruciale pour la définition de nouveaux composants.*

Le fonctionnement du composant Rectancle est simple : il demande l'affichage d'un rectangle à l'écran, selon des modalités qui sont spécifiées après l'accolade. Ces modalités sont principalement de deux types : des propriétés et sa composition, c'est-à-dire d'autres éléments.

Le composant Rectangle définit deux propriétés principales, qui sont sa hauteur `height` et sa largeur `width`. Il peut aussi avoir une couleur `color`, mais également une bordure colorée d'une largeur donnée, des bords arrondis, etc. ; toutes les propriétés prédéfinies sont répertoriées dans la documentation.

Par rapport à Python, la syntaxe de ces propriétés est particulière : elles correspondent approximativement aux variables de Python, mais l'attribution de valeur s'indique avec le symbole deux-points `:` et non d'égalité `=`.

Le rectangle ne contient ici qu'un seul composant enfant, qui est une instance de Text : celui-ci affiche simplement du texte, donné par sa propriété `text`. Cet affichage peut être paramétré à l'aide de propriétés, comme `color` pour la couleur (tout comme un rectangle). D'autres propriétés sont bien évidemment définies, comme la police de caractères ou la taille de fonte. D'ailleurs, le texte lui-même peut être défini de manière plus riche qu'actuellement à l'aide de balises HTML ; seul un sous-ensemble de balises est implémenté, mais il permet de réaliser la plupart des variations voulues.

Contrairement au rectangle, le composant Text définit lui-même sa taille à l'aide de son contenu. D'autres composants sont dans le même cas, comme une fenêtre principale ApplicationWindow. Définir la taille du composant de texte est toujours possible, à l'aide des propriétés `height` et `width`, mais le comportement est plus difficile à comprendre.

Lorsqu'une instance d'un composant est incluse dans une autre, il faut indiquer leur position l'une par rapport à l'autre. Une première manière de procéder consiste à utiliser des positions absolues, avec les propriétés `x` et `y` ; il vaut cependant souvent

❺ mieux utiliser le système d'ancres défini par Qt Quick : cette propriété indique que le centre du composant Text devra correspondre au centre de son composant parent (ici, Rectangle).

L'un des grands avantages de Qt Quick est que *cette simple définition s'assure que le texte restera centré dans la fenêtre, peu importe son redimensionnement*.

Figure 13.8 : *Premier exemple : un rectangle et du texte forment déjà une hiérarchie*

Note *> Tout en haut de la hiérarchie Qt Quick des composants affichés à l'écran, on retrouve deux composants. L'un d'entre eux est Item, qui ne définit rien d'autre que les propriétés de base nécessaires à tout affichage (comme les coordonnées et les dimensions). L'autre est Component, équivalent à un document QML instanciable (un fichier QML réutilisable, défini avec les techniques exposées au chapitre Créer un composant réutilisable).*

Pour entrer dans les détails techniques, tant Item que Component héritent de QtObject, à rapprocher de la classe QObject en Python. Ses responsabilités se limitent aux liens entre propriétés (voir plus bas). Cependant, QtObject est rarement utile dans des applications, mais peut servir à créer une nouvelle hiérarchie de composants qui n'ont aucune apparence graphique.

Il est possible d'aller un peu plus loin et d'associer une action à la fenêtre ainsi créée : lors d'un clic n'importe où, fermer la fenêtre. Pour réagir à un clic, le composant de bas niveau à utiliser est MouseArea. Il dispose d'une "propriété" un peu spéciale : `onClicked`. En réalité, il ne s'agit pas d'une propriété, mais d'un *gestionnaire de signal* : quand un événement se passe au niveau du composant (ici, quand il reçoit un clic), le code associé à ce gestionnaire de signal est exécuté. Ce code est écrit en JavaScript et respecte la syntaxe de ce langage, présentée au chapitre Présentation de JavaScript.

Même si l'apparence ne change pas d'un iota par rapport à l'exemple précédent, la différence est que cette application a une certaine interactivité avec l'utilisateur, même

si elle est minime. Cette zone sensible aux clics doit avoir deux caractéristiques : remplir complètement le parent (ce qui se fait également avec le système d'ancres de Qt Quick et la propriété `anchors.fill`) et réagir lors d'un clic (en associant du code au gestionnaire de signal `onClicked`).

***Note** > Cet exemple est disponible dans le dossier `rectangle-fermable` .*

```
import QtQuick 2.5

Rectangle {
    width: 200
    height: 100
    color: "lightblue"

    Text {
        anchors.centerIn: parent
        text: "Hello World"
        color: "darkblue"
    }

    MouseArea {
        anchors.fill: parent
        onClicked: Qt.quit()
    }
}
```

Comment charger un document QML depuis Python ?

Qt Creator n'est pas la seule manière de lancer des documents QML : PyQt peut faire de même, bien évidemment. Le plus simple est d'utiliser ce bout de code :

```
import sys

from PyQt5.QtWidgets import QApplication
from PyQt5.QtQml import QQmlApplicationEngine


if __name__ == "__main__":
    app = QApplication(sys.argv)
    engine = QQmlApplicationEngine("main.qml")
    engine.quit.connect(app.quit))
    sys.exit(app.exec_())
```

Il crée, comme tout code PyQt, une instance de QApplication, puis lance un moteur d'exécution Qt Quick (à travers la classe QQmlApplicationEngine).

L'interaction entre Python et Qt Quick dépasse largement le simple chargement d'une interface Qt Quick non intégrée à l'application PyQt : les techniques correspondantes sont présentées au chapitre Communiquer avec Python.

Note *> Ce code ne fonctionnera que si les documents Qt Quick à charger sont de type fenêtre. C'est la raison pour laquelle, dès à présent, tous les exemples exécutables auront, comme composant racine, ApplicationWindow : sinon, rien ne s'afficherait, tout simplement.*

Le composant ApplicationWindow est approfondi au chapitre Créer une fenêtre principale. Window remplirait exactement le même rôle dans bon nombre d'exemples (voir la Section 1.4, Fenêtre générique de ce même chapitre).

Note *> Dans certains cas (relativement rares), il peut être utile d'employer le composant Connections au lieu d'écrire les gestionnaires de signal au niveau du composant.*

3. Identifiants et interactivité

Les interfaces graphiques ont beaucoup à apporter à l'utilisateur grâce à l'interactivité qu'elles proposent. Cette notion regroupe bon nombre de réactions possibles aux stimuli de l'utilisateur et d'événements extérieurs : les menus déroulants, les réactions aux boutons, le redimensionnement de la fenêtre et de son contenu, la mise à jour automatique d'une liste à partir d'un site web, etc. Pour ce faire, Qt Quick propose deux mécanismes : le plus élémentaire est celui de lien entre propriétés ; le plus évolué traite directement avec l'émission de signaux, tout comme PyQt.

3.1. Définition d'identifiants

Une bonne partie de l'interactivité avec Qt Quick est implémentée en liant des éléments les uns aux autres. Pour ce faire, la première étape est de définir des identifiants pour les éléments intéressants : préciser un identifiant n'est jamais nécessaire dans un document QML, sauf pour y faire référence, comme dans les exemples précédents.

Les identifiants sont donnés par la propriété `id`. Une fois défini, l'identifiant d'un élément ne peut plus changer, tout comme le nom d'une variable. Par exemple :

```
import QtQuick 2.5

Rectangle {
    id: myRectangle
    width: 200
    height: 100
    // …
}
```

Cet identifiant se comporte comme une variable, notamment dans les autres éléments pour faire référence au composant ainsi identifié. Le mot clé `parent` fait référence à l'élément qui contient l'élément courant.

Note *> Les commentaires sur une ligne débutent par `//` (et non `#`, contrairement à Python). Il est aussi possible de définir des commentaires sur plusieurs lignes, en les entourant de `/*` et `*/`*

```
Rectangle {
    id: myRectangle // Commentaire sur une ligne.

    /* Commentaire sur plusieurs lignes.
```

```
    Commentaire sur plusieurs lignes.
    */
}
```

3.2. Liens entre propriétés

Par exemple, le bout de code suivant affiche deux champs de texte qui contrôlent l'affichage d'une chaîne de caractères : le premier champ détermine le texte à afficher, le deuxième sa couleur. Le champ de texte est repris du module Qt Quick Controls, ce qui permet notamment d'avoir un texte fictif (`placeholderText`), le temps que l'utilisateur saisisse quelque chose.

Les éléments de l'interface peuvent être positionnés à l'aide de coordonnées absolues selon les axes `x` et `y` : l'axe `x` est horizontal (il correspond à la largeur des composants), `y` est vertical (hauteur). `(0, 0)` correspond au coin supérieur gauche de la fenêtre. Ici, l'emplacement des différents widgets est déterminé par rapport à la largeur de l'élément parent (`parent.width`) et à celle de l'élément courant (`width` : implicitement, la propriété est celle de l'élément courant), mais aussi à la position d'autres éléments de l'interface (comme `textField.y`). Le lien entre l'affichage du texte et les deux champs se fait par leur identifiant : `colourField.text` et `textField.text`.

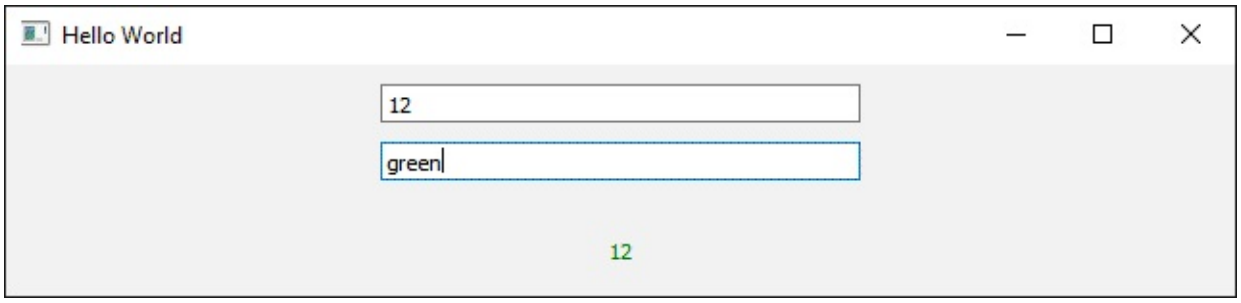
La valeur donnée à chaque coordonnée est en réalité une expression, un calcul qui est réévalué à chaque changement dans une de ses composantes : si la position d'un élément évolue, toute l'expression est réévaluée et la position de l'élément courant change. Il s'agit en réalité de code JavaScript, limité à sa partie expressions. Ce fonctionnement s'oppose à ce qui se passe dans la majorité des langages de programmation, où les expressions sont évaluées une et une seule fois.

***Note** > Cet exemple est disponible dans le dossier `lien-proprietes`  .*

```
import QtQuick 2.5
import QtQuick.Controls 1.4

ApplicationWindow {
    visible: true
    width: 640
    height: 120
    title: qsTr("Hello World")

    TextField {
        id: textField
        width: 250
        x: parent.width / 2 - width / 2
```

```
        y: parent.y + 10
        placeholderText: "Entrez du texte"
    }

    TextField {
        id: colourField
        width: 250
        x: parent.width / 2 - width / 2
        y: textField.y + 30
        placeholderText: "Saisissez une couleur"
    }

    Label {
        color: colourField.text
        text: textField.text
        x: parent.width / 2 - width / 2
        y: colourField.y + 50
    }
}
```

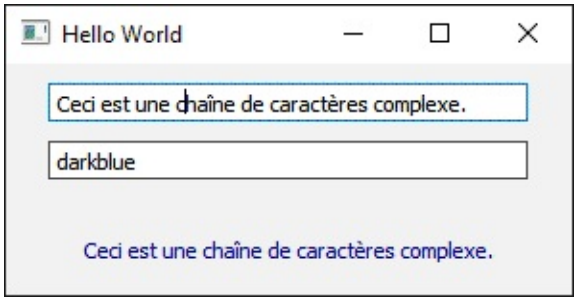
Figure 13.9 : *Utilisation d'identifiants pour lier l'affichage d'une chaîne de caractères à deux champs*



Cet exemple n'a rien de transcendant, mais il donne déjà une certaine forme d'interactivité : si l'utilisateur change le contenu des deux champs, l'affichage s'adapte immédiatement ; s'il redimensionne la fenêtre, malgré un positionnement en coordonnées absolues, tous les éléments se déplacent pour à nouveau être centrés dans la fenêtre.

Note *> Pendant la saisie de la couleur, l'utilisateur devra passer par toute une série de noms invalides (à moins d'employer du copier-coller) : il saisira d'abord un* `g`*, puis un* `r`*, etc., jusqu'à arriver à* `green`*. Aucune erreur ne sera affichée pendant ce temps (même si* `gre` *n'est pas un nom de couleur acceptable) : Qt Quick prendra une couleur par défaut.*

Figure 13.10 : *Identifiants et interactivité de base*

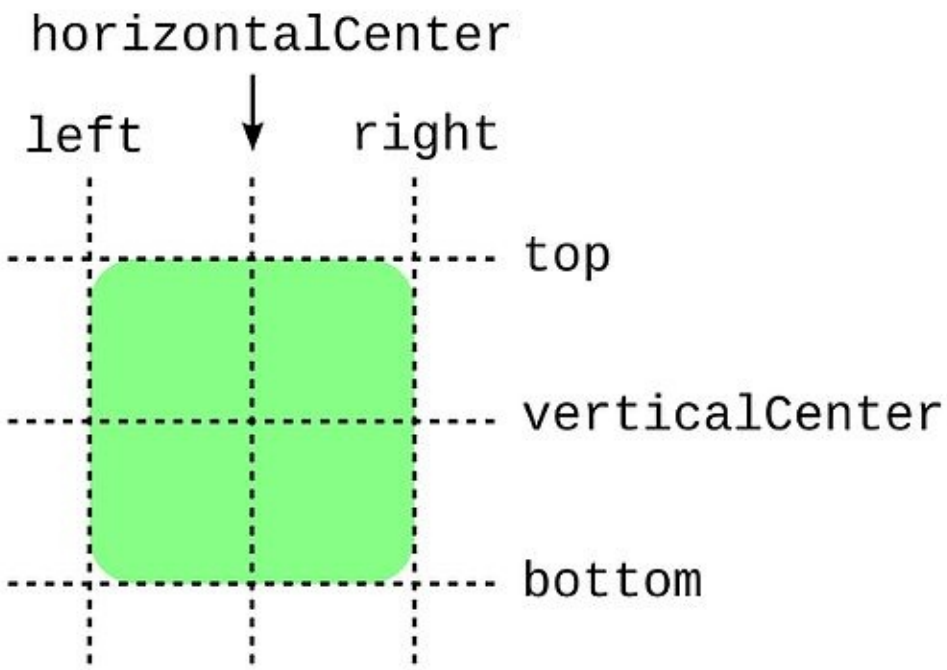


3.3. Système d'ancres

Une meilleure manière de positionner des éléments les uns par rapport aux autres est d'utiliser le système d'ancres prévu dans Qt Quick. Il exploite également le système d'identifiants et évite de manipuler des coordonnées absolues.

Pour chaque élément, six lignes particulières sont définies (voir Figure 13.11). Vous pouvez ainsi définir la position d'un composant en donnant la position de l'une de ces lignes. Par exemple, on peut ancrer sa ligne du dessus (propriété `anchors.top`) à la même hauteur que celle du bas d'un autre (`autre.bottom`), puis celle de gauche du premier (`anchors.left`) à celle de droite du second (`autre.right`).

Figure 13.11 : *Six emplacements définis pour les ancres*



Le code précédent peut ainsi être réécrit à l'aide des ancres. Il sera bien plus lisible et plus facile à composer : pour centrer les éléments, il n'est plus nécessaire de réfléchir à un calcul un peu particulier.

Note > *Cet exemple est disponible dans le dossier* `ancres`  .

```
import QtQuick 2.5
import QtQuick.Controls 1.4

ApplicationWindow {
    visible: true
    width: 640
    height: 120
    title: qsTr("Hello World")

    TextField {
        id: textField
        width: 250
        anchors.horizontalCenter: parent.horizontalCenter
```

❶ y: 10 ❷ placeholderText: qsTr("Enter text") } TextField { id: colourField width: 250 anchors.top: textField.bottom ❸ anchors.topMargin: 10 ❹ anchors.horizontalCenter: textField.horizontalCenter placeholderText: qsTr("Enter colour") } Label { color: colourField.text text: textField.text anchors.top: colourField.bottom anchors.topMargin: 10 anchors.horizontalCenter: colourField.horizontalCenter } }

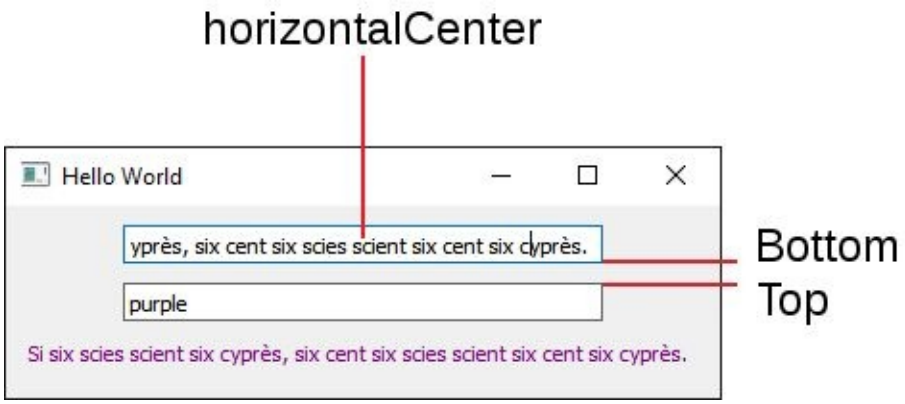
❶ La première ancre fait correspondre, sur l'axe horizontal, le centre du premier champ avec le centre (selon l'axe horizontal) de la fenêtre. Tout comme précédemment, lors du moindre redimensionnement de la fenêtre, le lien est préservé et le champ sera toujours centré.

❷ L'emploi des ancres n'interdit pas un positionnement absolu pour l'axe qui n'est pas fixé, ici, sur la hauteur de la fenêtre.

❸ Le dessus du deuxième champ correspondra au bas du premier, c'est-à-dire que le champ de couleur sera en dessous du champ de texte.

❹ Pour éviter que ces deux champs soient collés, cependant, il faut définir une certaine marge, soit en dessous du premier, soit au-dessus du second. Ici, la marge correspondra au-dessus du deuxième champ.

Figure 13.12 : *Interface identique avec des ancres*



3.4. Réactions au clic sur un bouton

Nous pouvons très facilement ajouter à cette interface des réactions à des stimuli plus poussés, par exemple avec des boutons pour augmenter ou diminuer la taille du texte affiché. Ces boutons sont aussi disponibles dans Qt Quick Controls.

Conceptuellement, la réaction de l'application est identique : à un moment donné, un signal est émis (ici, l'appui sur un bouton ou, dans la section précédente, le changement d'une valeur), un certain bout de code est exécuté (qu'il effectue directement une action ou, plus indirectement, qu'il change la valeur d'une propriété par réévaluation de l'expression). Le code exécuté lors de l'émission du signal est indiqué au niveau d'un gestionnaire de signal, ici `onClicked`, qui a la même forme qu'une propriété, sauf que la valeur associée est un morceau de code JavaScript.

Note *> Cet exemple est disponible dans le dossier* `boutons` .

```
import QtQuick 2.5
import QtQuick.Controls 1.4

ApplicationWindow {
    // …

    Button {
        id: increaseButton
        text: qsTr("Increase size")
        // …
        onClicked: text.font.pointSize += 1
```

```
    }

    Button {
        id: decreaseButton
        text: qsTr("Decrease size")
        // …
        onClicked: text.font.pointSize -= 1
    }

    Label {
        id: text
        color: colourField.text
        text: textField.text
        // …
    }
}
```

Figure 13.13 : *Augmenter la taille du texte par des boutons*

Note *> Les plus curieux ont probablement déjà tenté de descendre la taille de police sous zéro, en appuyant frénétiquement sur le bouton qui la diminue. Cependant, Qt Quick ne le leur permettra pas : il est impossible de faire baisser la taille sous zéro, l'opération de décrémentation est tout simplement ignorée.*

3.5. Gestionnaires de signal

Les liens entre propriétés sont très pratiques pour le développement d'applications, mais ils ne suffisent pas dans tous les cas, comme pour la réaction aux clics de souris. Le précédent exemple montrait sans le nommer un exemple de *signal*, la notion à la base de l'interactivité dans Qt Quick.

Ces signaux sont émis lorsqu'une action se produit dans l'interface : par exemple,

l'utilisateur a appuyé sur une touche, sur un bouton, des données sont disponibles depuis le réseau, etc. Il se transmet dans l'architecture interne de PyQt et finit par déclencher, côté Qt Quick, l'appel d'une fonction particulière : le *gestionnaire de signal* (en anglais, *signal handler*). Le programmeur peut associer ce bout de code directement à un signal donné.

***Note** > En réalité, ce système est bien plus général que celui des liaisons de propriétés : ce dernier réévalue des expressions lors de l'émission de signaux qui indiquent le changement de valeur d'une variable, alors que les gestionnaires de signal sont nettement plus flexibles, permettant d'effectuer n'importe quelle action lors de l'émission d'un signal (notamment définir la valeur d'une propriété). En réalité, il serait possible de se passer complètement du système de liaison de propriétés, mais le code deviendrait bien plus difficile à écrire.*

De manière générale, un gestionnaire de signal est défini lors de l'émission d'un signal : si le signal est nommé *`signal`*, alors le gestionnaire de signal correspondant sera `on`*`Signal`* (remarquer la différence de capitale). Lors d'un changement de valeur d'une propriété, le signal *`property`*`Changed` est émis : le gestionnaire de signal sera alors `on`*`Property`*`Changed`. Peu importe la propriété, ce signal sera toujours émis automatiquement, sans besoin de configuration. Cette utilisation de base a déjà été décrite à la Section 3.4, Réactions au clic sur un bouton.

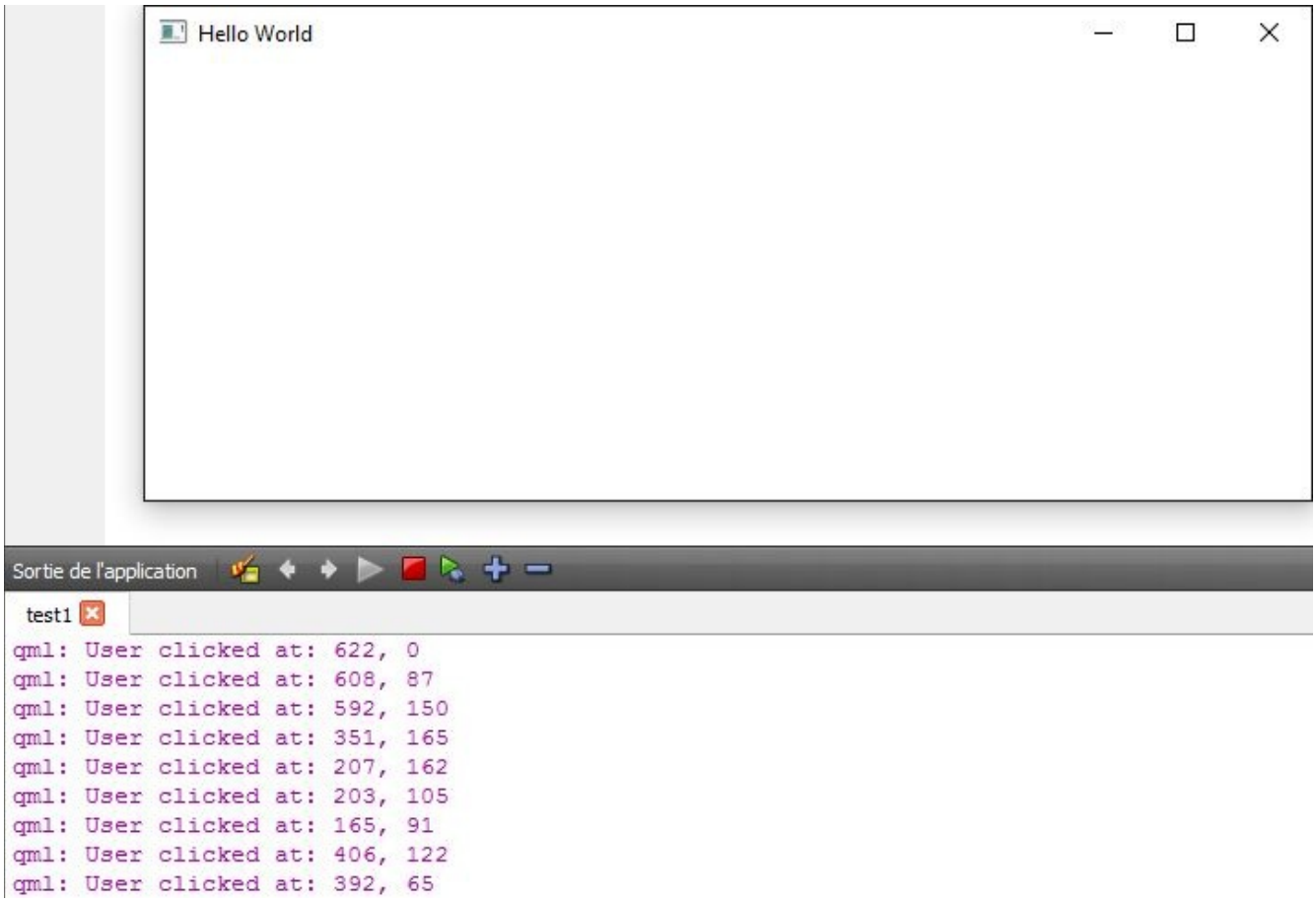
Dans la documentation de Qt Quick, les composants QML sont toujours accompagnés de leurs signaux, comme MouseArea, un composant qui sert à recevoir les événements en provenance de la souris. Certains possèdent également des paramètres, à l'instar de fonctions, comme `clicked(MouseEvent mouse)` : ils indiquent que le signal transporte une certaine information, qui sera disponible au niveau du gestionnaire de signal. Ici, le signal `clicked` offrira au code connecté une variable `mouse`, dont le type est `MouseEvent` : à travers cet objet, le code pourra déterminer, par exemple, l'endroit exact où le clic a eu lieu.

***Note** > Cet exemple est disponible dans le dossier `mousearea`  .*

```
import QtQuick 2.5
import QtQuick.Controls 1.4

ApplicationWindow {
    visible: true
    width: 640
    height: 240
    title: qsTr("Hello World")

    Rectangle {
```

```
        MouseArea {
            anchors.fill: parent
            onClicked: {
                console.log("User clicked at: " +
                            mouse.x + ", " + mouse.y)
            }
        }
    }
}
```

Figure 13.14 : *Utilisation des paramètres d'un gestionnaire de signal*



Note > *Un signal est émis lorsqu'un composant a fini d'être chargé en mémoire et avant que les liaisons entre propriétés soient calculées. Il est accessible par* `Component.onCompleted`*, sur n'importe quel composant Qt Quick. Il permet notamment d'initialiser certaines propriétés. De même,* `Component.onDestruction` *permet d'exécuter du code lors de la fermeture de l'application, par exemple pour enregistrer les données modifiées. Ces deux signaux seront régulièrement utilisés dans les exemples des chapitres plus avancés.*

14

Présentation de JavaScript

Niveau : débutant

Objectifs : écrire du code JavaScript dans le contexte d'une interface Qt Quick

Qt Quick utilise énormément de JavaScript, principalement pour gérer toutes les formes d'interactivité avec l'utilisateur ou l'environnement. Qu'il s'agisse d'une propriété d'un élément ou du code à exécuter lors d'un événement, JavaScript se présente comme incontournable pour le développement d'applications — même s'il est toujours possible de le remplacer par du code Python, la manière de procéder est plus compliquée (elle sera détaillée dans le chapitre Communiquer avec Python).

Ce chapitre présente les bases de JavaScript, un langage relativement simple à apprendre : malgré des différences syntaxiques avec Python, il lui ressemble en certains points. Notamment, tous deux sont des langages dynamiques principalement impératifs avec des accents de programmation fonctionnelle, leur typage est faible. Seules les parties intéressantes pour une utilisation dans Qt Quick seront montrées.

Un peu d'histoire…

JavaScript est un langage d'abord créé pour le Web, dans les années 1990, par les développeurs de Netscape. En peu de temps, il s'est répandu dans tous les autres navigateurs pour devenir un élément central des sites web modernes. Il a rapidement été normalisé par ECMA, sous le nom d'ECMAScript (qui correspond à la norme ECMA-262, régulièrement révisée). Grâce à Node.js, il s'est répandu au niveau des serveurs. De manière générale, JavaScript s'exporte de plus en plus en dehors de ses plates-bandes initiales et ce mouvement n'est pas près de s'arrêter.

1. Expressions

Jusqu'à présent, l'utilisation principale de JavaScript dans des fichiers QML était au niveau d'expressions, c'est-à-dire des valeurs données aux propriétés, par exemple pour copier le texte entré par l'utilisateur dans un champ. De manière générale, elles peuvent s'écrire de manière identique à Python, à l'exception de l'exponentiation et des connecteurs logiques.

```
2 + 4 * 8 - 2 // Opérations arithmétiques comme en Python
16 + Math.pow(9, 2) // L'exponentiation se fait avec la fonction
                    // pow(base, exposant), il n'y a pas d'opérateur
**

2 < 4 // Comparaisons comme en Python
8 != 9 && 81 == Math.pow(9, 2) // Les connecteurs logiques s'écrivent
                               // && pour and et || pour or
(2 < 4) == true // Aussi, les booléens s'écrivent true et false
                // (au lieu de True et False avec Python)
true == ! false // La négation s'écrit ! (et non not)
```

Une expression peut aussi faire intervenir la valeur des propriétés d'éléments identifiés. Dans ce cas, l'identifiant est séparé de la propriété par un point `.` — tout comme l'accès aux membres d'un objet en Python. Par exemple, le chapitre d'introduction montrait ceci :

```
TextField {
    x: parent.width / 2 - width / 2
    y: textField.y + 30
}
```

Tout comme Python, JavaScript propose d'écrire une condition sous la forme d'une expression, avec la syntaxe d'un opérateur ternaire :

```
x = (a > 1) ? b : c // En Python, équivalent à x = b if a > 1 else c
```

Ces expressions peuvent également contenir des appels à des fonctions, qui s'écrivent de la même manière qu'en Python, à l'exception que les paramètres nommés n'existent pas. Certaines fonctions préprogrammées peuvent se révéler utiles, notamment pour transformer le type d'une expression :

```
parseInt("42") // Convertit la chaîne de caractères donnée
               // en nombre entier
parseInt("2a", 16) // Même opération en spécifiant la base
```

```
                    //(ici, un nombre hexadécimal)
parseFloat("5.67") // Même opération pour un nombre
                    // à virgule flottante
String(false) // Opération inverse : convertir une expression
              // en chaîne de caractères

parseFloat("5.67aa") // Les caractères qui ne sont pas des chiffres
                     // sont ignorés…
parseFloat("µµ5.67") // … à moins d'être avant le nombre à analyser.
                     // Cet exemple renverra une erreur,
                     // sous la forme d'une valeur NaN.
```

Quand les expressions sont - elles évaluées ?

Contrairement à du code Python traditionnel, les expressions JavaScript *peuvent* être réévaluées automatiquement *dans un contexte Qt Quick*. Ce n'est le cas que lorsqu'elles définissent la valeur d'une propriété, quand un (ou plusieurs) des éléments utilisés dans l'expression change.

Par exemple, la première manière de centrer un champ de texte utilisait cette caractéristique de Qt Quick :

```
TextField {
    id: textField
    x: parent.width / 2 - width / 2
    // …
}
```

La valeur de la propriété `x` est définie par l'expression `parent.width / 2 - width / 2` : dès qu'il y a un changement dans la valeur de `parent.width`, toute l'expression est réévaluée. Dans ce cas, cela n'arrive que lors d'un redimensionnement de la fenêtre.

***Note** > Un tableau comparatif d'expressions JavaScript et Python est disponible dans le Tableau .2.*

2. Blocs de code

Les gestionnaires de signal tels que `onClicked`, utilisé pour traiter le clic sur un bouton à la Section 3.4, Réactions au clic sur un bouton, ne prennent pas une expression, mais bien un véritable bout de code : une suite d'instructions, pas simplement une valeur. Le cas présenté ne montre qu'une seule instruction. Pour en présenter plusieurs, il faut définir un bloc de code. Là où Python utilise l'indentation, JavaScript utilise des accolades : le début du bloc est indiqué par `{` , la fin par `}`.

Ainsi, pour afficher dans la console la nouvelle taille suite à l'appui sur un bouton de l'exemple de la Section 3.4, Réactions au clic sur un bouton, le code peut être modifié comme suit :

```
ApplicationWindow {
    Button {
        id: increaseButton
        text: qsTr("Increase size")
        // …
        onClicked: {
            text.font.pointSize += 1;
            console.log(text.font.pointSize);
        }
    }

    // …
}
```

La fonction `console.log` sert à afficher du texte dans la console, dont la sortie est recopiée par Qt Creator. Puisqu'il s'agit d'une deuxième instruction à effectuer lorsque le signal de clic arrive, il est nécessaire de définir un bloc de code, entre les accolades.

Attention *> Comme en Python, les points-virgules en fin d'instruction ne sont pas obligatoires. Par contre, en JavaScript, il est recommandé d'en mettre, afin d'éviter certains problèmes quand une ligne commence par une parenthèse.*

De tels blocs peuvent contenir des variables, qui ont la même signification qu'en Python. La différence est qu'elles doivent être déclarées avec le mot clé `var` avant toute utilisation, par exemple comme ici pour définir un tableau de trois éléments :

```
{
    var array = [1, 2, 3];
```

```
    console.log(array);
}
```

***Note** > Si vous tentez de définir une variable sans le mot clé `var`, vous aurez droit à un message d'erreur pas toujours explicite : `Error: Invalid write to global property "array"`. Sans ce mot clé, Qt Quick considère que l'écriture se fait dans une propriété d'un composant particulier, l'objet global, d'où le message d'erreur. L'erreur est facile à faire : bon nombre d'exemples de code sur Internet omettent ce mot clé, puisqu'ils sont prévus pour être exécutés dans un navigateur (et non dans Qt Quick).*

Ces blocs de code comprennent bien évidemment les conditions et les boucles. La syntaxe des conditions est aussi similaire à celle de Python, toujours en remplaçant l'indentation par des accolades. Par exemple, pour n'afficher la nouvelle taille que si elle est supérieure à cinq unités :

```
onClicked: {
    text.font.pointSize += 1;
    if (text.font.pointSize >= 5) {
        console.log(text.font.pointSize);
    }
}
```

JavaScript propose également une construction très similaire à cette condition, mais qui est une expression, tout comme l'opérateur ternaire : par exemple, il est possible de définir la valeur d'une propriété ou d'une variable avec l'opérateur `if`.

```
Text {
    text:
        if(x >= 0)
            "x est grand";
        else
            "x est petit";
    // Cette construction est équivalente à :
    // text: (x >= 0) ? "x est grand" : "x est petit"
}
```

Au contraire des conditions, les boucles sont inspirées du C et ne partagent pas beaucoup de points communs avec Python. La syntaxe générale présente trois parties, séparées par des points-virgules `;` :

```
for (var index = 0; index < M; i++) {
    // …
}
```

***Attention** > Contrairement aux points-virgules de fin d'instruction, ceux des boucles ne peuvent jamais être omis.*

La première partie initialise la valeur d'une variable de compteur de boucle (généralement nommée `i`, pour des raisons historiques), souvent à zéro ; la deuxième donne une condition : une fois qu'elle n'est plus satisfaite, la boucle s'arrête ; la troisième et dernière est une instruction effectuée à chaque fin d'itération, qui incrémente souvent la valeur du compteur. Aucune de ces parties n'est requise, mais les points-virgules doivent rester. Ainsi, la forme générale précédente est strictement équivalente à celle-ci :

```
var index = 0;
for (;;) {
    if (! (index < M)) {
        break; // Ce mot clé arrête l'exécution de la boucle.
    }

    // …

    i++
}
```

Ces boucles sont notamment utiles pour itérer sur tous les éléments d'un tableau. Le compteur de boucle correspondra alors à la position dans le tableau et la condition sera de garder un indice inférieur au dernier dans le tableau (dont la taille est indiquée par `array.length`).

```
var array = [1, 2, 3];
for (var i = 0; i < array.length; i++) {
    // L'élément courant est accessible par array[i].
    // …
}
```

Attention *> La syntaxe* `for (variable in object)` *fait penser à celle de Python, mais ne lui correspond pas vraiment : en JavaScript, elle itère sur les propriétés énumérables de l'objet. En particulier, pour un tableau, il n'y a aucune garantie d'itérer sur les indices dans un certain ordre… ni de n'itérer que sur les indices ! Il vaut donc mieux l'éviter à moins de savoir ce que l'on fait.*

Note *> Un tableau comparatif de constructions JavaScript et Python est disponible dans le Tableau .4.*

3. Fonctions

Les fonctions en JavaScript sont, en quelque sorte, une version simplifiée de celles disponibles en Python : il n'est pas possible de définir de paramètre avec mot clé ou valeur par défaut. Syntaxiquement, la principale différence est l'utilisation du mot clé `function` au lieu de `def`. Par exemple, pour renvoyer la somme des deux arguments :

```
function sum(x, y) {
    return x + y;
}
```

Fonctions anonymes

Tout comme en Python, il est possible de définir des fonctions anonymes, qui sont alors traitées comme n'importe quelle valeur dans le langage (booléen, nombre, chaîne de caractères, etc.) : elles peuvent être stockées dans des variables ou passées en argument à des fonctions. En revanche, les fonctions anonymes en JavaScript utilisent le même mot clé (alors que, en Python, on utilisera `lambda`) et peuvent contenir plusieurs instructions. Par exemple, le code précédent est équivalent à celui-ci :

```
var sum = function(x, y) {
    return x + y;
}
```

Une boucle sur un tableau peut alors se réécrire avec la fonction `forEach` (associée à chaque tableau) et une fonction anonyme appelée à chaque itération. Ses arguments sont alors la valeur actuelle, son indice et le tableau complet, ce qui donne la même information qu'une boucle standard.

```
var array = [1, 2, 3];
array.forEach(function(v, i, a) {
    // … code à appeler sur chaque élément du tableau
    // (Par définition, v == a[i].)
});
```

La fonction `map()` sert à transformer un tableau en un autre tableau, chaque élément étant transformé. Ce genre d'opération est relativement fréquent, par exemple pour transformer une liste de chaînes de caractères en bas de casse. Cette fonction `map()` s'applique sur un tableau et prend en argument une fonction anonyme, qui prend en

argument un élément du tableau et renvoie la valeur dans le nouveau tableau.

```
var array = [1, 2, 3];
var newArray = array.map(function(val) {
    return val + 2;
}
// newArray vaut [3, 4, 5].
```

La question la plus importante est cependant l'endroit où ces fonctions peuvent être définies dans du code Qt Quick. La manière la plus simple, mais probablement la moins propre, de procéder est de les écrire directement au niveau des composants, au même niveau qu'une propriété ou un élément imbriqué. Ainsi, l'exemple de la Section 3.4, Réactions au clic sur un bouton peut se réécrire avec une fonction pour remplacer l'instruction du gestionnaire de signal :

Note *> Le code de cet exemple est disponible dans le projet* `fonctions-position`  .

```
ApplicationWindow {
    function increaseSize() {
        text.font.pointSize += 1;
    }

    Button {
        id: increaseButton
        text: qsTr("Increase size")
        // …
        onClicked: increaseSize()
    }

    // …
}
```

Fonctions anonymes et liaisons entre propriétés

Ces fonctions anonymes sont notamment utiles pour créer des liaisons entre propriétés *à la main*. Cette manière de procéder est assez rare et répond à des besoins spécifiques.

Dans l'exemple du chapitre précédent, le lien entre le champ de texte pour la couleur et la couleur proprement dite se fait immédiatement. Pour qu'il n'existe qu'après un clic sur un bouton, le code pourrait être le suivant :

```
ApplicationWindow {
    // …

    Label {
        id: text
        color: "black" // Pas de lien créé !
        text: textField.text
        // …
    }

    Button {
        // …
        text: qsTr("Enable colour tweaking")

        // Le lien entre la couleur du texte et le champ
correspondant
        // est créé avec ce code.
        onClicked: text.color = Qt.binding(function() {
            return colourField.text;
        })
    }
}
```

Au contraire, si on avait simplement écrit la liaison de propriétés comme si elle était effectuée au niveau du composant Label, le code correspondant ne serait appelé que lors de chaque appui sur le bouton : la liaison entre propriétés ne serait pas créée.

```
ApplicationWindow {
    // …

    Label {
        id: text
        color: "black"
        // …
    }

    Button {
        // …
        // Le gestionnaire de signal n'est appelé que lors d'un
clic
        // sur le bouton : la couleur n'est mise à jour qu'à ce
moment,
        // pas en continu comme avec une liaison de propriétés.
        onClicked: text.color = colourField.text
    }
}
```

4. Séparation du code JavaScript

Insérer des fonctions JavaScript directement dans le code QML n'est pas vraiment une bonne pratique, surtout si ces fonctions sont nombreuses. De même, pour utiliser une même fonction dans plusieurs fichiers QML, le copier-coller n'est pas un choix acceptable. C'est pourquoi Qt Quick permet d'*inclure* des fichiers JavaScript, grâce à l'instruction `import`, placée en tout début de fichier ; par exemple, pour inclure le fichier `code.js` :

```
import QtQuick 2.5
import "code.js" as Code
```

Dans ce cas, toutes les fonctions de `code.js` seront disponibles dans l'identifiant `Code` : si une fonction `factorial()` y est définie, elle sera accessible côté QML comme `Code.factorial()`.

Ce même mécanisme d'inclusion est disponible entre fichiers JavaScript, grâce à une extension à la syntaxe de base de JavaScript (qui n'est donc pas disponible dans un navigateur) : la fonction `Qt.include("file.js")` inclut le fichier `file.js`. Toutes les fonctions de `file.js` seront alors directement disponibles dans le fichier JavaScript, mais aussi côté QML.

***Note** > Lors de la création d'un fichier JavaScript, Qt Creator demandera s'il doit créer un fichier sans état. Les fonctions contenues dans ces fichiers ne peuvent pas manipuler des composants Qt Quick : elles ne peuvent que calculer des résultats en fonction de la valeur de leurs entrées. Cette différence se marque au niveau de la première ligne du fichier JavaScript :*

```
.pragma library
```

L'exemple précédent peut se refactoriser pour déplacer les deux fonctions de changement de taille du texte affiché. Pour plus de propreté, elles ne travaillent plus forcément sur un seul composant bien nommé, mais prennent en argument le composant dont la taille doit varier. Ainsi, le fichier JavaScript `code.js` s'écrira comme ceci :

***Note** > Le code de cet exemple est disponible dans le projet `fichier-js` .*

```
function increaseSize(textField) {
    textField.pointSize += 1;
}
```

```
function decreaseSize(textField) {
    textField.pointSize -= 1;
}
```

Ensuite, le fichier QML `main.qml` sera légèrement modifié pour d'abord importer ce nouveau fichier, puis pour les appels de fonction :

```
import QtQuick 2.5
import QtQuick.Controls 1.4
import "code.js" as Code

ApplicationWindow {
    // …

    Button {
        // …
        onClicked: Code.increaseSize(textField)
    }
```

5. Structures de données

JavaScript dispose principalement de deux types de structures de données : les tableaux et les objets. Les premiers servent à stocker une suite d'objets indexée et ont déjà été présentés à la Section 2, Blocs de code, ils s'utilisent de la même manière que les listes de Python. Les objets, quant à eux, sont des structures plus évoluées, très similaires aux dictionnaires de Python. De manière très simple, ces objets correspondent à des tableaux dont l'indexation peut se faire avec une chaîne de caractères. Par exemple, pour représenter un livre :

```
var book = {
    "title": "Commentarii de Bello Gallico",
    "author": "Caius Iulius Caesar",
    "publisher": "Les Belles Lettres",
    "year": 1926
}
```

Chaque clé du dictionnaire, chaque propriété de l'objet est indiquée entre guillemets (optionnels) ; la valeur en est séparée par un deux-points `:`.

Pour récupérer les attributs de ce dictionnaire, en modifier la valeur ou en ajouter, on utilise une syntaxe très similaire aux tableaux, à l'exception qu'une chaîne de caractères remplace la position :

```
a["title"] // Correspond à 'Commentarii de Bello Gallico'
```

Les notations de tableau et d'objet peuvent s'imbriquer à l'infini, sans problème. Par exemple, voici une liste de livres : il s'agit d'un tableau (d'où les crochets à l'extérieur) d'objets (d'où les accolades à l'intérieur).

```
var books = [
    {
        "title": "Commentarii de Bello Gallico",
        "author": "Caius Iulius Caesar",
        "publisher": "Les Belles Lettres",
        "year": 1926
    },
    {
        "title": "Προμηθεύς δεσμώτης",
        "author": "Αἰσχύλος",
        "publisher": "Les Solitaires Intempestifs",
        "year": 2010
    }
```

```
]
```

JSON est souvent présenté comme une norme d'échange des données. En réalité, il s'agit de structures de données JavaScript : tout fichier JSON est syntaxiquement valable en JavaScript[19]. L'acronyme JSON signifie d'ailleurs *JavaScript object notation*.

La fonction `",".join()` de Python sert à concaténer les chaînes de caractères d'une liste en insérant un caractère (ou plusieurs) entre deux éléments. Par exemple, pour la liste `["chiens", "chats", "poulets"]`, la fonction `", ".join(l)` renvoie la chaîne de caractères `"chiens, chats, poulets"`. La même fonctionnalité est accessible directement en JavaScript, toujours avec une fonction `join()`, mais avec une syntaxe inversée : cette fonction s'applique sur une liste et prend en argument le séparateur.

```
["chiens", "chats", "poulets"].join(", ")
```

Programmation orientée objet

Si ces dictionnaires portent le nom d'objets, c'est qu'ils correspondent à cette notion, dans le même sens que la programmation orientée objet. Chaque propriété peut ainsi correspondre à une fonction. À l'intérieur de cette fonction, le mot clé `this` correspond à l'objet courant (tout comme `self` en Python, la différence étant qu'il est ici implicite). La syntaxe d'appel aux méthodes est identique à Python : *`variable.méthode`*`(`*`arguments`*`)`.

```
var book = {
    title: "Commentarii de Bello Gallico", // Propriété de
l'objet.
    printTitle: function() {               // Méthode de l'objet.
        console.log(this.title);
    }
}
book.printTitle(); // Appel d'une méthode de l'objet book.
```

Les propriétés de l'objet sont accessibles de la même manière qu'en Python :

```
book.title == book['title']
```

Cette syntaxe est une alternative à celle utilisant des crochets.

Attention *> JavaScript n'a pas la même notion de programmation orientée objet qu'en Python ou la plupart des autres langages orientés objets : JavaScript ne*

définit pas de classe, mais bien des prototypes. Ainsi, l'instanciation d'un objet se fera comme précédemment ou par référence à un autre objet ; l'héritage se fait exclusivement à l'échelle des objets, puisqu'il n'y a pas de classe. Pour plus d'informations, reportez-vous par exemple au Guide JavaScript sur Mozilla Developer Network.

---

[19] À quelques exceptions près : le caractère de retour à la ligne, par exemple, est autorisé par JSON à l'intérieur d'une clé, mais pas en JavaScript (le caractère doit être échappé : `\n`). Dans l'autre sens, JavaScript autorise les clés sans guillemets et entre guillemets simples `'`, mais JSON n'autorise que les guillemets doubles `"`.

15

Créer une fenêtre principale

Niveau : débutant

Objectifs : créer une application de bureautique traditionnelle avec Qt Quick, découvrir les composants de base

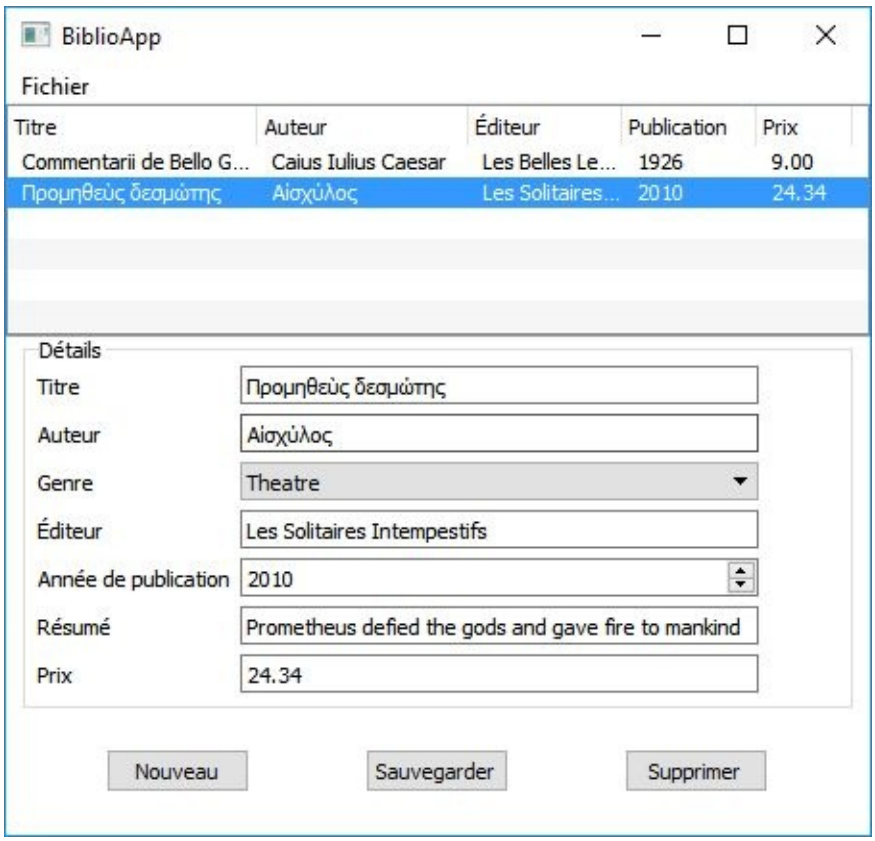
Prérequis : Premiers pas avec Qt Quick, Présentation de JavaScript

Les deux premiers chapitres vous ont donné toutes les bases nécessaires pour créer de petites applications Qt Quick. Cependant, l'environnement propose bien plus de composants pour faciliter la création d'interfaces, qu'elles soient orientées bureautique ou non. Cette section se focalisera sur des applications de type gestion, avec une barre de menus, des barres d'outils, une zone centrale de travail et une barre d'état — un type d'application dont le succès ne se dément pas depuis les années 1980.

Au niveau pratique, ce chapitre mettra en place l'interface globale de l'application d'exemple pour la gestion d'une bibliothèque.

Note *> Le code des exemples de ce chapitres est disponible dans le dossier* `application-bureautique` .

Figure 15.1 : *Vue de l'application d'exemple*



***Note** > Les composants utilisés dans ce chapitre proviennent tous des modules `QtQuick.Controls` et `QtQuick.Layouts`, ils ne sont pas disponibles dans le module `QtQuick` (qui ne contient que des composants de bas niveau).*

***Note** > Ce chapitre se focalise sur l'utilisation des Qt Quick Controls 1, pas la version 2 disponible depuis Qt 5.7. À l'heure actuelle, il est prévu que ces deux versions coexistent pendant encore un certain temps (probablement jusqu'à Qt 6). Les différences à l'utilisation entre les deux versions sont relativement faibles et indiquées dans le texte ; à terme, les deux versions devraient être équivalentes en fonctionnalités, mais ce n'est pas encore le cas.*

Cette deuxième version vient principalement combler des manques en performance lors de l'utilisation dans le domaine de l'embarqué (téléphones, tablettes, mais aussi machines à café, installations industrielles, voitures, etc.), en effectuant des compromis sur l'apparence. Par exemple, une application Qt Quick Controls 2 ne s'intégrera pas au style du système : exécutée sur un autre système, l'application gardera le même aspect.

En d'autres termes, la première version reste toujours d'actualité pour bon nombre d'applications. Dans les cas où il n'y a pas d'intégration souhaitée avec le système

d'exploitation et les autres applications que l'utilisateur pourrait lancer (dans l'embarqué, principalement), si la performance est un point critique, alors les Qt Quick Controls 2 deviennent intéressants.

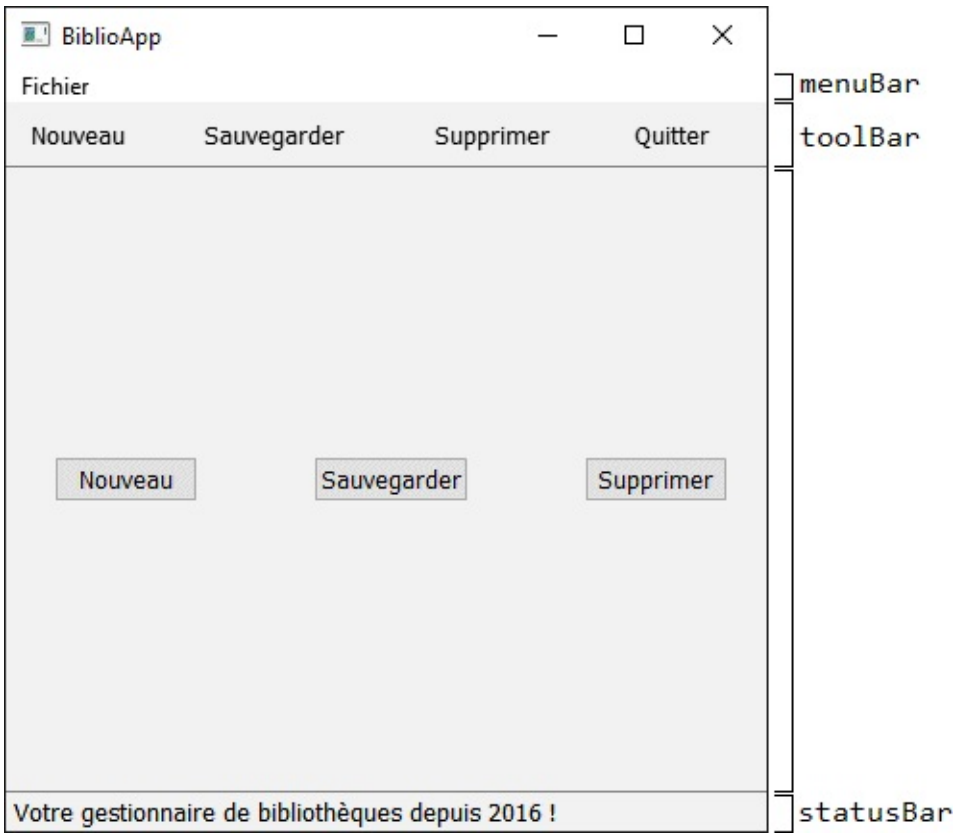
1. Composant ApplicationWindow

Une interface bureautique s'articule généralement autour du composant ApplicationWindow, déjà utilisé dans les exemples précédents. Elle correspond à la classe QMainWindow côté Python.

Son objectif est de fournir un modèle type d'interface pour une application de bureautique, avec des facilités pour définir des barres de menus et des barres d'outils. Son principal avantage par rapport à une solution faite à la main, avec les outils de positionnement déjà montrés, c'est que ce composant se charge tout seul d'ancrer les éléments à leur emplacement traditionnel dans l'interface.

Un point important des interfaces créées de la sorte (par rapport à un positionnement entièrement manuel des éléments) est qu'elle respectera toujours les préceptes de la plateforme d'exécution : la barre de menus sera déportée tout en haut de l'écran pour macOS et GNOME Shell, mais sous le titre de l'application sous Windows et KDE.

Figure 15.2 : *Fenêtre type d'une application de bureautique*



1.1. Interface statique

Cette interface, certes très simple, correspond à une quarantaine de lignes de code, sans la moindre interactivité : elle est statique, le menu se déroule, mais aucun bouton n'a d'action. Son code est très court et très lisible : il est possible de créer ainsi très rapidement des prototypes d'application, mais aussi de les faire évoluer.

Note *> Le code des exemples qui suivent est disponible dans le dossier* `applicationwindow` .

```
ApplicationWindow {
    visible: true; width: 396; height: 398
    title: "BiblioApp"

    menuBar: MenuBar {
        Menu {
            title: "&Fichier"
            MenuItem { text: "Ouvrir…" }
            MenuItem { text: "Enregistrer" }
            MenuItem { text: "Quitter" }
        }
    }

    toolBar: ToolBar {
        RowLayout {
            anchors.fill: parent
            ToolButton { text: "Nouveau" }
            ToolButton { text: "Sauvegarder" }
            ToolButton { text: "Supprimer" }
            ToolButton { text: "Quitter" }
        }
    }

    statusBar: StatusBar {
        Label { text: "Votre gestionnaire de bibliothèques depuis
2016 !" }
    }

    RowLayout {
        x: 25; y: 150; spacing: 60
        Button { text: "Nouveau" }
        Button { text: "Sauvegarder" }
        Button { text: "Supprimer" }
    }
}
```

L'interface est composée de quatre zones, comme on peut le voir à la Figure 15.2 : la

barre de menus (pour Qt Quick, la propriété `menuBar`), la barre d'outils (`toolBar`), la barre d'état (`statusBar`) et la zone de travail. Bien évidemment, toutes ces parties sont optionnelles : toutes les applications n'ont pas forcément besoin d'une barre d'outils ou d'état, comme BiblioApp, l'exemple de cet ouvrage.

La première barre, celle des menus, se définit à l'aide d'une instance de MenuBar. Elle prend en argument une série de Menu, qui sont disposés le long de la barre des menus dans leur ordre de définition. Chaque menu dispose d'un titre (`title`), puis d'une suite d'éléments : des MenuItem, mais aussi des MenuSeparator pour séparer deux blocs d'éléments par une ligne.

Les deux autres barres, d'outils avec ToolBar et d'état avec StatusBar, sont très similaires : à l'usage, la différence principale est le nom du composant utilisé. À la différence de la barre de menus, *elles ne disposent pas leurs éléments elles-mêmes* : si on leur donne une suite d'éléments à afficher, ils se superposeront, les uns par-dessus les autres — ce qui n'est guère fonctionnel. Il faut recourir à un composant positionneur pour les afficher comme souhaité, principalement RowLayout pour les aligner à l'horizontale. Ce mécanisme offre une certaine flexibilité : pour créer plusieurs barres d'outils, il suffit d'utiliser un positionneur vertical comme ColumnLayout.

***Note** > Pour Qt Quick, le contenu des barres d'outils et d'état est totalement libre : ToolButton a une apparence prévue pour les barres d'outils, mais rien n'empêche d'utiliser un autre composant à la place. De même, la barre d'état ne contient pas forcément du texte.*

***Note** > Les Qt Quick Controls 2 disposent, dès leur version 2.1 (PyQt 5.8), du composant ToolSeparator, qui correspond à la barre verticale qui sépare deux boutons dans les barres d'outils. Pour les autres versions, vous pouvez l'implémenter vous-même.*

Au contraire, pour les menus, vous pouvez d'ores et déjà utiliser MenuSeparator.

1.2. Interactivité dans les menus et barres d'outils

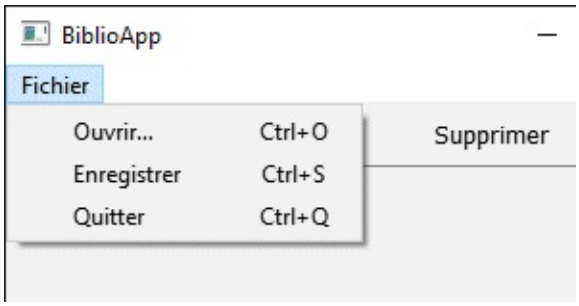
Jusqu'à présent, l'interface était statique. Cependant, il est très facile d'ajouter de l'interactivité aux éléments introduits, tout en respectant les mécanismes standards définis pour chaque plateforme.

Par exemple, vous pouvez définir une action à exécuter lorsqu'un élément du menu est sélectionné (`onTriggered`, qui est un gestionnaire de signal). De plus, la propriété `shortcut` permet de lui associer un raccourci clavier, qui déclenche la même action. Ainsi, l'élément permettant de fermer l'application devient :

```
MenuItem {
    text: "Quitter"
    shortcut: "Ctrl+Q" // StandardKey.Quit
    onTriggered: Qt.quit()
}
```

Note *> Qt Quick propose deux manières de définir un raccourci clavier : soit par une chaîne de caractères qui le décrit (ici, il s'agit d'appuyer sur les touches Ctrl et Q en même temps), soit par un raccourci standard, défini de manière indépendante de la plateforme. La chaîne de caractères traduit automatiquement les raccourcis selon la plateforme : pour macOS, Ctrl sera compris comme la touche Cmd.*

Figure 15.3 : *Ajout d'un raccourci clavier à une entrée de menu*



De manière très similaire, vous pouvez définir des actions pour les boutons de la barre d'outils avec le gestionnaire de signal `onClicked` (exactement comme pour Button) :

```
ToolButton {
    text: "Quitter"
    onClicked: Qt.quit()
}
```

Le texte peut être remplacé par une image grâce à la propriété `iconSource`.

1.3. Partage d'actions entre les éléments

Jusqu'à présent, chaque action possible aussi bien depuis les menus que les barres d'outils a été dupliquée : d'un côté, l'entrée de menu définit un texte et un raccourci clavier ; de l'autre, l'outil définit une icône. Les deux exécutent le même code. Cela ne

pose pas de problème particulier en centralisant le code dans une bibliothèque JavaScript. Par contre, comment faire pour désactiver une telle action de manière globale ? Le code qui l'effectue doit se souvenir des deux emplacements où l'action est possible.

Le composant Action permet de factoriser le code : toutes les données d'une action sont stockées d'un côté (code à exécuter, texte, icône, raccourci clavier, principalement), les menus et outils y font appel. Quand l'action est désactivée, elle l'est partout. Quand elle est cochée (comme l'option gras dans un traitement de texte), elle l'est partout.

De manière générale, une action reprend tous les champs définis précédemment (dans les boutons et entrées de menu). Le code effectué par l'action est donné par le gestionnaire de signal `onTriggered` (comme pour une entrée de menu, contrairement à un bouton).

```
Action {
    id: quitAction
    text: "Quitter"
    onTriggered: Qt.quit()
    shortcut: "Ctrl+Q" // StandardKey.Quit
}
```

Ensuite, cette même action peut être exploitée depuis plusieurs endroits en utilisant la propriété `action` de MenuItem et de ToolButton :

```
menuBar: MenuBar {
    Menu {
        title: "&Fichier"
        MenuItem { action: quitAction }
    }
}

toolBar: ToolBar {
    RowLayout {
        // …
        ToolButton { action: quitAction }
    }
}
```

Fonctionnellement, l'application est identique pour l'utilisateur ; par contre, le code est plus facile à écrire (ici, pas de duplication du texte à afficher dans le menu et dans la barre d'outils).

***Note** > Qt Quick Controls 2 n'a pas encore de correspondance pour le composant Action — le plus proche est Shortcut, qui fonctionne uniquement pour les raccourcis clavier. Il n'existe plus de manière de factoriser le code pour toutes les actions, mais*

cela devrait être corrigé dans une version ultérieure.

En attendant, le mieux est d'écrire le code de l'action sous la forme d'une fonction, qui sera appelée de tous les endroits où cette action est possible ; les autres informations peuvent être mémorisées dans un tableau d'objets global.

1.4. Fenêtre générique

ApplicationWindow est un type très spécifique de fenêtre. Qt Quick propose aussi Window, un composant qui représente une fenêtre, sans imposer de structure aussi forte que les trois parties d'ApplicationWindow. En réalité, une fenêtre créée avec Window est extrêmement personnalisable : ApplicationWindow est implémenté par-dessus Window, mais il est aussi possible de définir des fenêtres transparentes.

```
import QtQuick 2.5
import QtQuick.Window 2.0

Window {
    Rectangle { … }
}
```

Si un composant Window est utilisé à l'intérieur d'une fenêtre existante, alors cela créera une nouvelle fenêtre. En utilisant les options ad hoc (des drapeaux), celle-ci peut s'ouvrir comme une boîte de dialogue :

```
Window {
    flags: Qt.Dialog
    Rectangle { … }
}
```

(Qt Quick propose une série de boîtes de dialogue déjà codées, qui correspondent à la majorité des besoins.)

Window est principalement utile pour personnaliser entièrement la fenêtre affichée, sans ressembler aux fenêtres habituelles pour le système d'exploitation : dans ce cas, la propriété `flags` peut prendre la valeur `Qt.FramelessWindowHint`. Par exemple, avec ce mode, on peut créer une fenêtre semi-transparente, qui s'affiche par-dessus les autres fenêtres (grâce à la propriété `opacity`).

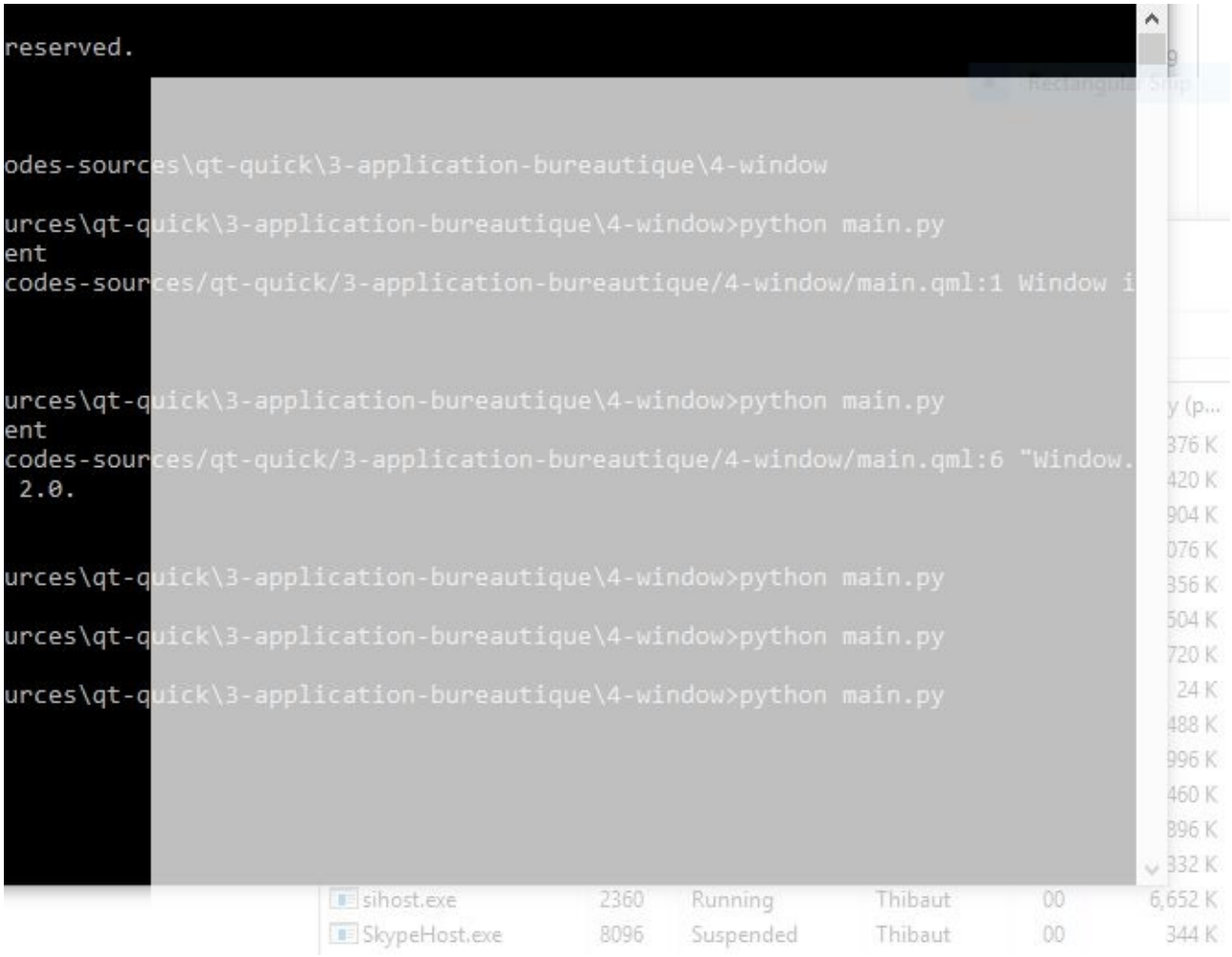
```
Window {
    flags: Qt.FramelessWindowHint
    opacity: .75
    width: 640
```

```
    height: 480
    visible: true
}
```

***Note** > Cette opacité ne fonctionnera que si le drapeau `Qt.FramelessWindowHint` est défini !*

Il est aussi possible de n'afficher que certains des boutons par défaut d'une fenêtre (les plus courants étant minimiser, maximiser et fermer), en jouant sur les valeurs données à `flags`. Toutes sont définies dans la documentation et peuvent être combinées par un OU binaire `|`.

***Figure 15.4 :** Fenêtre Qt Quick entièrement transparente, sans bordure ni boutons par défaut*



2. Positionneurs

Pour afficher plusieurs barres d'outils, par exemple, les *positionneurs* (layouts) de Qt Quick se révèlent très utiles : en fonction de certains paramètres, ces composants particuliers se chargent de disposer leurs éléments dans l'espace alloué. Par exemple, RowLayout sert à répartir les éléments sur une ligne. Le mécanisme des positionneurs est cependant plus général : il est aussi possible de disposer des éléments dans une grille ou dans une colonne.

Cette manière de procéder est plus simple que le recours à des ancres, puisque tous les éléments sont automatiquement affichés dans leur ordre de définition, sans qu'il soit nécessaire d'indiquer les liens entre deux éléments consécutifs. De plus, ces positionneurs gèrent automatiquement la dimension des composants selon le redimensionnement de l'interface.

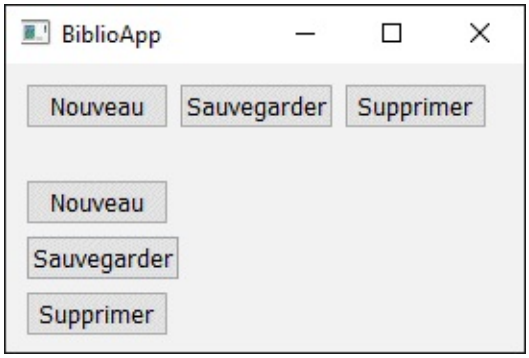
Tous les positionneurs se trouvent dans un module particulier de Qt Quick : `QtQuick.Layouts`. Les deux principaux ont déjà été brièvement présentés pour afficher les boutons d'une barre : RowLayout pour les aligner horizontalement, ColumnLayout verticalement.

2.1. Redimensionnement

La manière la plus simple d'utiliser des positionneurs est probablement de leur attribuer une position absolue dans l'interface, comme n'importe quel autre composant ; dans ce cas, ils disposeront leurs éléments à partir de cette position, qui correspond à leur coin en haut à gauche. Cependant, de la sorte, le conteneur ne s'adaptera jamais à la largeur de la fenêtre en cas de redimensionnement.

```
RowLayout {
    x: 10; y: 10

    Button { text: "Nouveau" }
    Button { text: "Sauvegarder" }
    Button { text: "Supprimer" }
}

ColumnLayout {
    // Idem
}
```

Figure 15.5 : *Positionnement absolu de positionneurs : aucun redimensionnement*



Au contraire, en utilisant des ancres pour, par exemple, occuper tout l'espace disponible du parent, l'affichage du contenu s'adaptera à la largeur en cas de redimensionnement. De plus, en indiquant des contraintes sur la taille de chaque composant, tous les éléments du conteneur pourront s'adapter eux-mêmes à la nouvelle taille. Pour ce faire, chaque composant dispose de propriétés attachées `Layout.`, qui contrôlent ce mécanisme.

```
RowLayout {
    // …
    Rectangle {
        width: 50; height: 150
```

❶ Text { … } } Rectangle { height: 150 ❷ Layout.fillWidth: true ❸ Layout.minimumWidth: 50 Layout.preferredWidth: 50 ❹ Layout.maximumWidth: 150 Text { … } } Rectangle { height: 150 Layout.fillWidth: true Layout.minimumWidth: 75 Layout.preferredWidth: 100 ❺ Layout.maximumWidth: 250 Text { … } } }

❶ Le premier rectangle du positionneur a une taille fixée : jamais il ne sera redimensionné, ses dimensions resteront celles fixées à cet endroit (à moins que la valeur de ses propriétés soit changée à l'exécution).

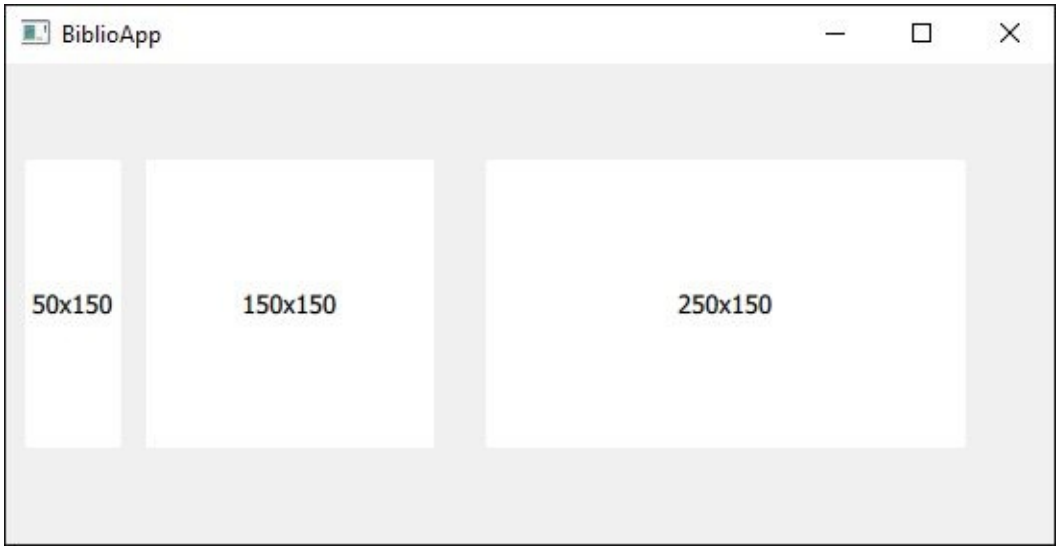
❷ Le deuxième rectangle aura sa hauteur fixée, le reste de l'exemple se focalisera sur la largeur.

❸ Pour que Qt Quick se charge de changer la largeur des éléments à l'exécution, cette propriété doit être activée. Sinon, les éléments ne seront pas impactés, peu importe les autres propriétés.

❹ Trois propriétés contrôlent la largeur possible pour le rectangle : un minimum `minimumWidth` et un maximum `maximumWidth` sont obligatoires, plus une largeur "préférée" `preferredWidth`. Cette dernière est facultative et indique la largeur vers laquelle les algorithmes de redimensionnement devraient tendre.

❺ À redimensionnement donné (voir, par exemple, Figure 15.6), le rectangle avec une largeur préférée plus élevée prendra plus de place que les autres.

Figure 15.6 : *Impact du redimensionnement sur les éléments d'un positionneur*



Note *> Pour que le positionneur gère la hauteur des éléments qui lui sont fournis, les mêmes propriétés s'appliquent, en remplaçant* `Width` *par* `Height`.

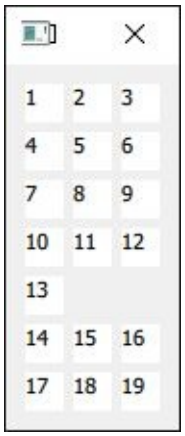
2.2. Grilles

Les positionneurs servent également à afficher des éléments sous la forme d'une grille, par exemple dans un formulaire pour aligner les labels et les contrôles correspondants, mais aussi sur l'écran d'accueil d'un téléphone mobile pour étaler les différentes applications installées.

Ce conteneur se nomme GridLayout. Par défaut, il se configure principalement avec la propriété `columns` et dispose son contenu de droite à gauche dans chacune des colonnes. La propriété attachée `Layout.columnSpan` affiche un élément à cheval sur plusieurs colonnes.

```
GridLayout {
    // …
    columns: 3

    Rectangle { width: 20; height: 20; Text { text: "1" } }
    // …
    Rectangle {
        width: 20; height: 20; Text { text: "13" }
        Layout.columnSpan: 3
    }
    // …
    Rectangle { width: 20; height: 20; Text { text: "19" } }
}
```

Figure 15.7 : *Positionneur en grille imposant le nombre de colonnes*

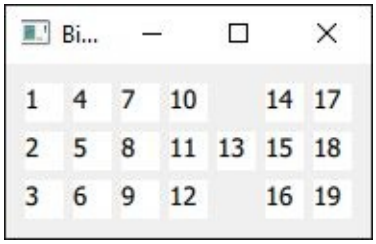


Pour imposer le nombre de lignes (avec la propriété `rows`), il faut changer la manière dont GridLayout dispose les éléments à l'aide de la propriété `flow` : au lieu de les mettre de gauche à droite (`GridLayout.LeftToRight`, la valeur par défaut), ils devront l'être de haut en bas (`GridLayout.TopToBottom`).

```
GridLayout {
    anchors.fill: parent
    anchors.margins: 10

    flow: GridLayout.TopToBottom
    rows: 3

    Rectangle { width: 20; height: 20; Text { text: "1" } }
    // …
    Rectangle {
        width: 20; height: 20; Text { text: "13" }
        Layout.rowSpan: 3
    }
    // …
```

```
    Rectangle { width: 20; height: 20; Text { text: "19" } }
}
```

Figure 15.8 : *Positionneur en grille imposant le nombre de lignes*

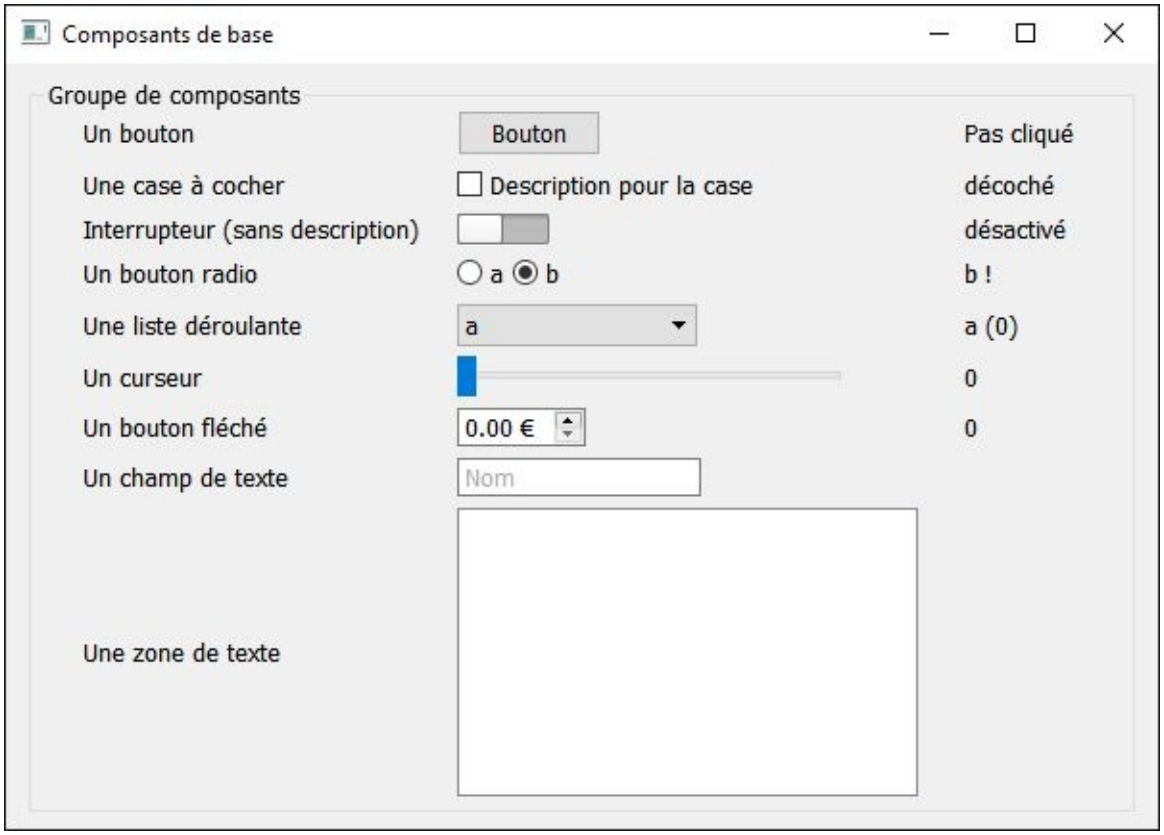


Note > *De par le fonctionnement par défaut de Qt Quick, l'élément 13 se trouve dans la première colonne quand on définit le nombre de colonnes et dans la deuxième ligne pour le nombre de lignes : si rien n'est précisé, les éléments sont positionnés verticalement à gauche et horizontalement au centre.*

3. Composants d'une interface

Les barres d'outils et autres menus sont des parties très importantes d'une interface graphique, mais rares sont les applications à s'en satisfaire : elles ont aussi très souvent besoin de composants comme des boutons, des champs de texte, des cases à cocher, etc. Bien évidemment, Qt Quick propose tous ces composants de base.

Figure 15.9 : *Démonstration de composants de base*



Bouton

Le premier est le bouton Button, déjà présenté : le texte est indiqué dans la propriété `text`, l'action déclenchée se fait par le gestionnaire de signal `onClicked`.

```
Button { text: "Bouton"; onClicked: btnLbl.text = "Cliqué !" }
Label { id: btnLbl; text: "Pas cliqué" }
```

Case à cocher

La case à cocher CheckBox peut s'associer à une description `text`, affichée à côté d'elle. Son résultat est disponible dans la propriété `checked`.

```
CheckBox { id: check; text: "Description pour la case" }
Label { text: check.checked ? "coché" : "décoché" }
```

Interrupteur

L'interrupteur Switch fonctionne sur le même principe, sans la description.

```
Switch { id: sw; }
Label { text: sw.checked ? "activé" : "désactivé" }
```

Bouton radio

Les boutons radio RadioButton proposent un choix exclusif dans un groupe (une seule case est cochée dans un bloc). Cette contrainte est imposée à travers un ExclusiveGroup, référencé dans chaque bouton radio par la propriété `exclusiveGroup`.

```
RowLayout {
    ExclusiveGroup { id: grp }
    RadioButton { id: radioA; text: "a"; exclusiveGroup: grp }
    RadioButton { text: "b"; exclusiveGroup: grp; checked: true }
}
Label { text: radioA.checked ? "a !" : "b !" }
```

Liste déroulante

Une liste déroulante ComboBox propose de choisir un élément dans une liste d'options, précisées dans la propriété `model` sous la forme d'une liste de chaînes de caractères (ou, en toute généralité, d'un modèle de chaînes de caractères). La sélection de l'utilisateur peut être récupérée soit par son indice dans le modèle (`currentIndex`), soit directement par la valeur textuelle de l'élément sélectionné (`currentText`).

```
ComboBox { id: combo; model: ["a", "b", "c"] }
Label { text: combo.currentText + " (" + combo.currentIndex + ")"
}
```

Curseur

Pour choisir un nombre, le curseur `Slider` se déplace sur une ligne graduée entre `minimumValue` et `maximumValue`, avec un intervalle entre deux graduations de `stepSize`.

```
Slider {
    id: slider
    minimumValue: 0
    maximumValue: 25
    stepSize: 1
}
Label { text: slider.value }
```

Une autre manière de sélectionner un nombre, plus classique, passe par un bouton fléché SpinBox, qui indique directement la valeur. Celui-ci peut imposer, en plus, un nombre de décimales affichées `decimals` et un préfixe `prefix` ou un suffixe `suffix`, notamment pour afficher une unité (ici, un symbole monétaire).

```
SpinBox {
    id: spin
    minimumValue: 0
    maximumValue: 25
    stepSize: .50
    decimals: 2
    suffix: " €"
}
Label { text: spin.value }
```

Champ de texte

Pour entrer du texte court, le choix le plus logique est un simple champ de texte TextField, qui se limite à une ligne de texte.[20]

```
TextField { id: field; placeholderText: "Nom" }
Label { text: field.text }
```

Au contraire, pour de plus grandes étendues de texte, il est préférable d'y réserver une zone plus imposante, avec `TextArea`.

```
TextArea { id: area }
Label { text: area.text }
```

Groupe

Un dernier composant bien pratique permet de grouper une série d'autres éléments de l'interface : GroupBox. Cette zone se présente avec un titre et une bordure tout autour de la zone délimitée. La difficulté avec cet élément est que, par défaut, sa taille est déterminée par son contenu — ce qui peut nuire à l'affichage de la bordure.

```
GroupBox {
    title: "Groupe de composants"
```

```
        width: parent.width * .96
        height: parent.height * .96
        anchors.horizontalCenter: parent.horizontalCenter
        anchors.verticalCenter: parent.verticalCenter
        // Contenu du groupe.
    }
```

---

[20] À ne pas confondre avec TextInput : ce dernier est disponible de base dans Qt Quick (pas besoin d'importer le module Qt Quick Controls), mais est de bien plus bas niveau. Là où TextField affiche une bordure autour du texte à éditer, par exemple, TextInput ne fait que montrer le texte (exactement comme Text) avec un curseur pour le modifier.

4. Boîtes de dialogue

Un autre besoin courant dans une application est d'afficher de petites boîtes de dialogue : sélectionner un fichier à ouvrir, préciser un endroit où enregistrer un fichier, choisir une police, poser une question à l'utilisateur, etc. Qt Quick propose toutes ces fonctionnalités dans le module `QtQuick.Dialogs`. Ce module a l'avantage de s'adapter directement à la plateforme : les boutons affichés sont toujours au bon endroit dans la boîte de dialogue, dans le bon ordre, avec le bon texte dans la langue de l'utilisateur. Par exemple, pour quitter un document sans sauvegarder, macOS affichera Ne pas enregistrer, GNOME Fermer sans enregistrer, les autres Abandonner — le tout, automatiquement !

Boîte standard

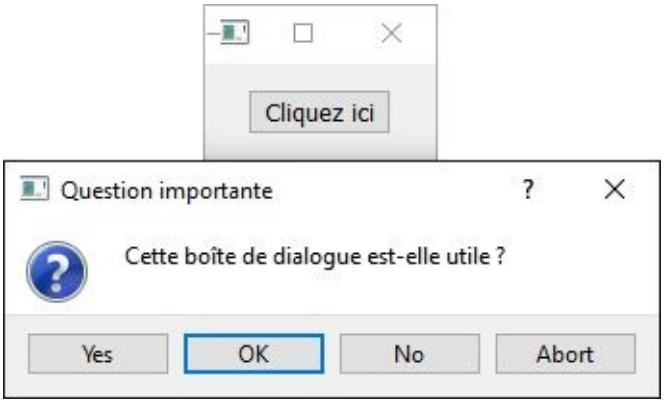
La boîte de base correspond au composant MessageDialog. Par défaut, elle n'affiche qu'un seul bouton : OK. La liste se contrôle par la propriété `standardButtons`, qui prend une combinaison de valeurs donnée dans la documentation. Chacun de ces boutons a un rôle particulier : par exemple, un bouton Oui correspond au rôle `YesRole`, OK à `AcceptRole`. Ces rôles définissent les signaux (et donc les gestionnaires de signal) pour répondre aux clics : `onYes` pour le rôle `YesRole`.

Enfin, la propriété `icon` demande d'afficher une icône : un point d'interrogation pour marquer une question (`StandardIcon.Question`) ou divers degrés d'information, du plus faible (`StandardIcon.Information`) au plus critique (`StandardIcon.Critical`) en passant par l'avertissement (`StandardIcon.Warning`).

```
MessageDialog {
    id: dialog
    standardButtons: StandardButton.Ok | StandardButton.Yes |
                     StandardButton.No | StandardButton.Abort
    title: "Question importante"
    text: "Cette boîte de dialogue est-elle utile ?"
    icon: StandardIcon.Question

    onYes: console.log("Oui")
    onNo: console.log("Non")
    onAccepted: console.log("OK")
    onRejected: console.log("Annuler")
}
```

```
Button {
    anchors.centerIn: parent
    text: "Cliquez ici"
    onClicked: dialog.open()
}
```

Figure 15.10 : *Exemple d'utilisation de MessageDialog avec quatre boutons, dans l'ordre habituel sous Windows et dans la langue de l'utilisateur (ici, en anglais)*



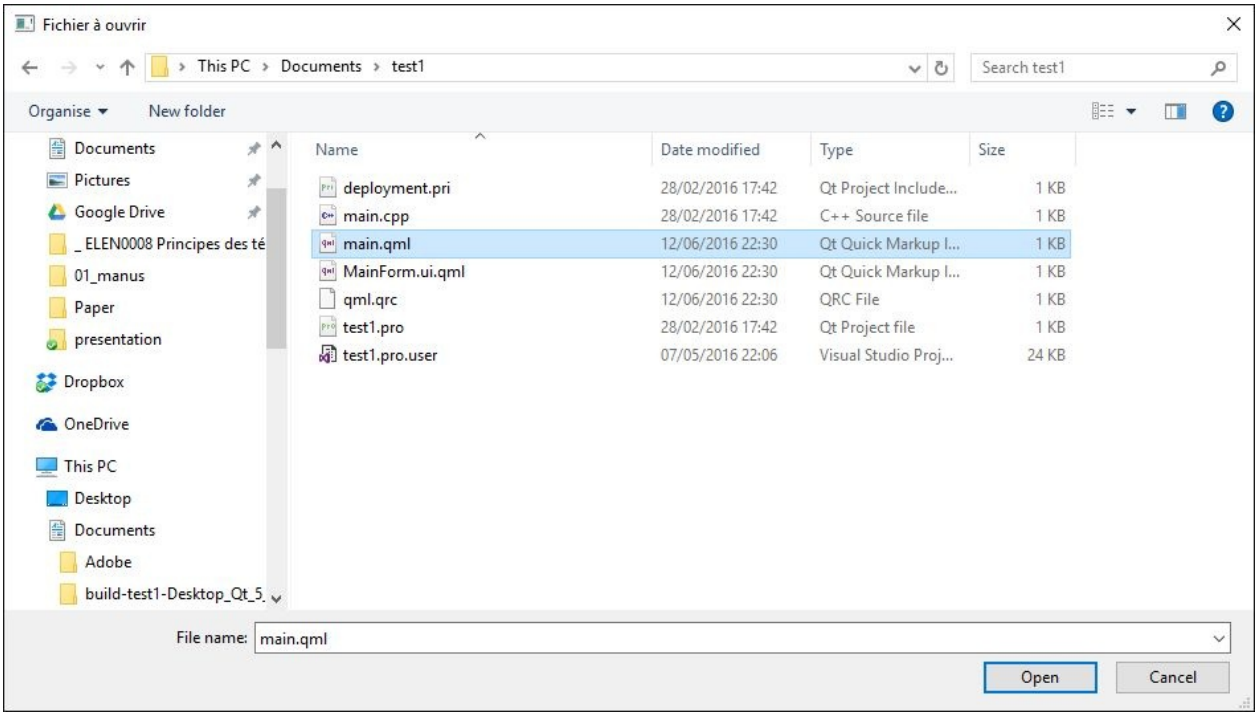
Note > *La traduction de l'interface est gérée de manière distincte mais pas automatique.*

Boîte de navigation

L'autre type de boîte de dialogue très utile offre la possibilité de choisir un fichier, tant pour la lecture que l'enregistrement (la distinction est que, dans le second cas, le fichier à sélectionner n'existe pas forcément). Cette fois, il faudra utiliser le composant FileDialog, la propriété déterminant si le fichier choisi doit exister étant `selectExisting` (vraie par défaut). Le fichier choisi est accessible, après acceptation par l'utilisateur, dans la propriété `fileUrl`.

```
FileDialog {
    id: file
    title: "Fichier à ouvrir"
    onAccepted: accept.open()
}

MessageDialog {
    id: accept
    text: "Fichier choisi : " + file.fileUrl
}
```

Figure 15.11 : *Exemple de boîte de sélection d'un fichier à ouvrir*


Fichier à ouvrir
This PC › Documents › test1
Search test1
Organise
New folder
Documents
Pictures
Google Drive
_ ELEN0008 Principes des té
01_manus
Paper
presentation
Dropbox
OneDrive
This PC
Desktop
Documents
Adobe
build-test1-Desktop_Qt_5.
Name
Date modified
Type
Size
deployment.pri 28/02/2016 17:42 Qt Project Include... 1 KB
main.cpp 28/02/2016 17:42 C++ Source file 1 KB
main.qml 12/06/2016 22:30 Qt Quick Markup l... 1 KB
MainForm.ui.qml 12/06/2016 22:30 Qt Quick Markup l... 1 KB
qml.qrc 12/06/2016 22:30 QRC File 1 KB
test1.pro 28/02/2016 17:42 Qt Project file 1 KB
test1.pro.user 07/05/2016 22:06 Visual Studio Proj... 24 KB
File name: main.qml
Open
Cancel

5. Squelette de BiblioApp

Ces quelques concepts de base sont d'ores et déjà suffisants pour ébaucher l'interface de BiblioApp, l'application d'exemple. Cette interface se divise en deux parties : la première affiche la liste des livres, la seconde permet d'en éditer un en particulier. Ce genre de division peut s'obtenir à l'aide du composant SplitView, qui fonctionne exactement comme un positionneur ; son apport majeur est de laisser l'utilisateur déplacer lui-même la barre qui sépare les deux zones. Notez la propriété `orientation`, qui indique comment effectuer la séparation en deux parties : dans le cas de BiblioApp, comme on veut que les deux blocs s'alignent verticalement, sa valeur sera donc `Qt.Vertical`.

***Note** > Le code source de cet exemple est disponible dans le projet `biblioapp-squelette`*  .

```
ApplicationWindow {
    id: window; width: 280; visible: true

    menuBar: MenuBar {
        Menu {
            title: "&Fichier"
            MenuItem { text: "Quitter" }
        }
    }

    SplitView {
        anchors.fill: parent
        orientation: Qt.Vertical

        Rectangle {
            width: window.width
            Layout.fillHeight: true
            Layout.minimumHeight: 120
            Layout.maximumHeight: 240

            Text {
                anchors.centerIn: parent
                text: "Liste des livres"
            }
        }

        Rectangle {
            width: window.width
            Layout.fillHeight: true
            Layout.minimumHeight: 340
```

```
            Layout.maximumHeight: 680

            Text {
                anchors.centerIn: parent
                text: "Détails d'un livre"
            }
        }
    }
}
```

Figure 15.12 : *Squelette de BiblioApp*

Note > *SplitView n'est pas encore disponible avec Qt Quick Controls 2, mais cela devrait arriver dans le futur.*

5.1. Modèle

Il est relativement facile de remplir la première partie de l'interface. En fait, il s'agit

d'une part de mémoriser des données, d'autre part de les afficher. Avec Qt en général, la manière la plus propre de stocker des données est d'utiliser un *modèle*. Ici, il peut simplement s'agir de données au format JSON sous la forme d'un tableau :

Note *> Le code source de cet exemple est disponible dans le projet* `biblioapp-repeater` .

```
[
    {
        "title": "Commentarii de Bello Gallico",
        "author": "Caius Iulius Caesar",
        "genre": "Nonfiction",
        "publisher": "Les Belles Lettres",
        "year": "1926",
        "summary": "Julius Caesar's firsthand account of the Gallic
Wars",
        "price": "9.00"
    },
    {
        "title": "Προμηθεύς δεσμώτης",
        "author": "Αἰσχύλος",
        "genre": "Theatre",
        "publisher": "Les Solitaires Intempestifs",
        "year": "2010",
        "summary": "Prometheus defied the gods and gave fire to
mankind",
        "price": "24.34"
    }
]
```

Avec ce modèle, le composant Repeater peut être utilisé pour répéter un composant un certain nombre de fois (un *délégué*), chacune en prenant un élément différent du modèle. La position de l'élément en cours d'affichage est disponible dans la propriété attachée `index`, tandis que le contenu de cet élément l'est dans `modelData`.

```
ColumnLayout {
    Repeater {
        model: […]
        Text {
            text: modelData["title"] + ", " + modelData["author"]
        }
    }
}
```

C'est tout ce qui est nécessaire. Certes, l'affichage est rudimentaire, mais il est fonctionnel.

Note *> La propriété attachée* `modelData` *n'est disponible que pour certains types de*

modèles. Le sujet est approfondi au chapitre Utiliser la méthodologie modèle - vue.

Figure 15.13 : *BiblioApp avec une liste élémentaire de livres*

Note > *Le code précédent embarque du texte dans un alphabet tout à fait différent du français ou de l'anglais (des caractères en grec ancien). Cela est possible sans encombre et sans configuration spécifique, Qt Creator fonctionnant par défaut avec un encodage UTF-8. Vous pourriez donc tout aussi bien insérer, par exemple, une phrase en japonais sans problème.*

5.2. Détails d'un livre

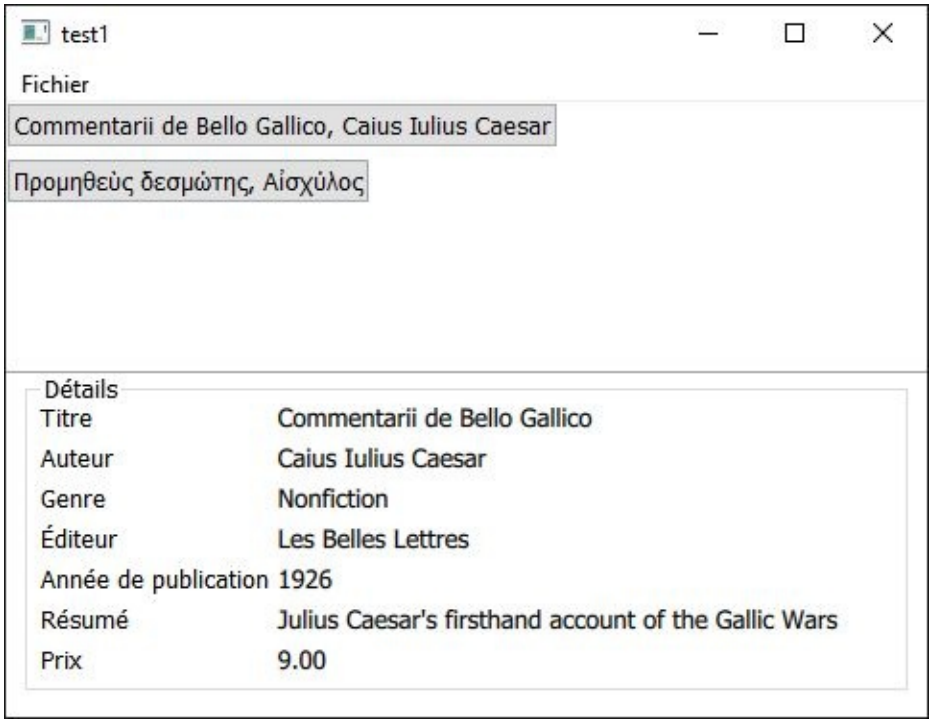
La deuxième partie de l'interface sert à afficher les détails du livre sélectionné. Elle peut être remplie avec une grille à deux colonnes : d'un côté, les labels des champs affichés ; de l'autre, les valeurs pour le livre sélectionné.

Le mode de communication avec la liste est extrêmement simple pour le moment : les différents éléments de la liste sont transformés en boutons, qui imposent aux composants de texte dans la grille les valeurs correspondantes.

Note *> Le code source de cet exemple est disponible dans le projet* `biblioapp-basique`  *.*

```
Repeater {
    id: r
    model: […]
    Button {
        text: modelData["title"] + ", " + modelData["author"]
        onClicked: {
            dTitle.text = modelData['title'];
            // …
        }
    }
}

GroupBox {
    // …
    GridLayout {
        columns: 2

        Text {
            Layout.columnSpan: 2
            Layout.minimumWidth: window.width
            Layout.maximumWidth: window.width

            text: "Détails d'un livre"
            horizontalAlignment: Text.AlignHCenter
        }

        Label { text: "Titre" }
        Text { id: dTitle; }
        // …
    }
}
```

Figure 15.14 : *Première version complète de BiblioApp, en cent lignes de code*


test1
Fichier
Commentarii de Bello Gallico, Caius Iulius Caesar
Προμηθεὺς δεσμώτης, Αἰσχύλος
Détails
Titre
Commentarii de Bello Gallico
Auteur
Caius Iulius Caesar
Genre
Nonfiction
Éditeur
Les Belles Lettres
Année de publication
1926
Résumé
Julius Caesar's firsthand account of the Gallic Wars
Prix
9.00

16

Utiliser la méthodologie modèle-vue

Niveau : débutant à intermédiaire

Objectifs : exploiter le paradigme modèle-vue pour concevoir des applications

Prérequis : Premiers pas avec Qt Quick, Présentation de JavaScript, Créer une fenêtre principale

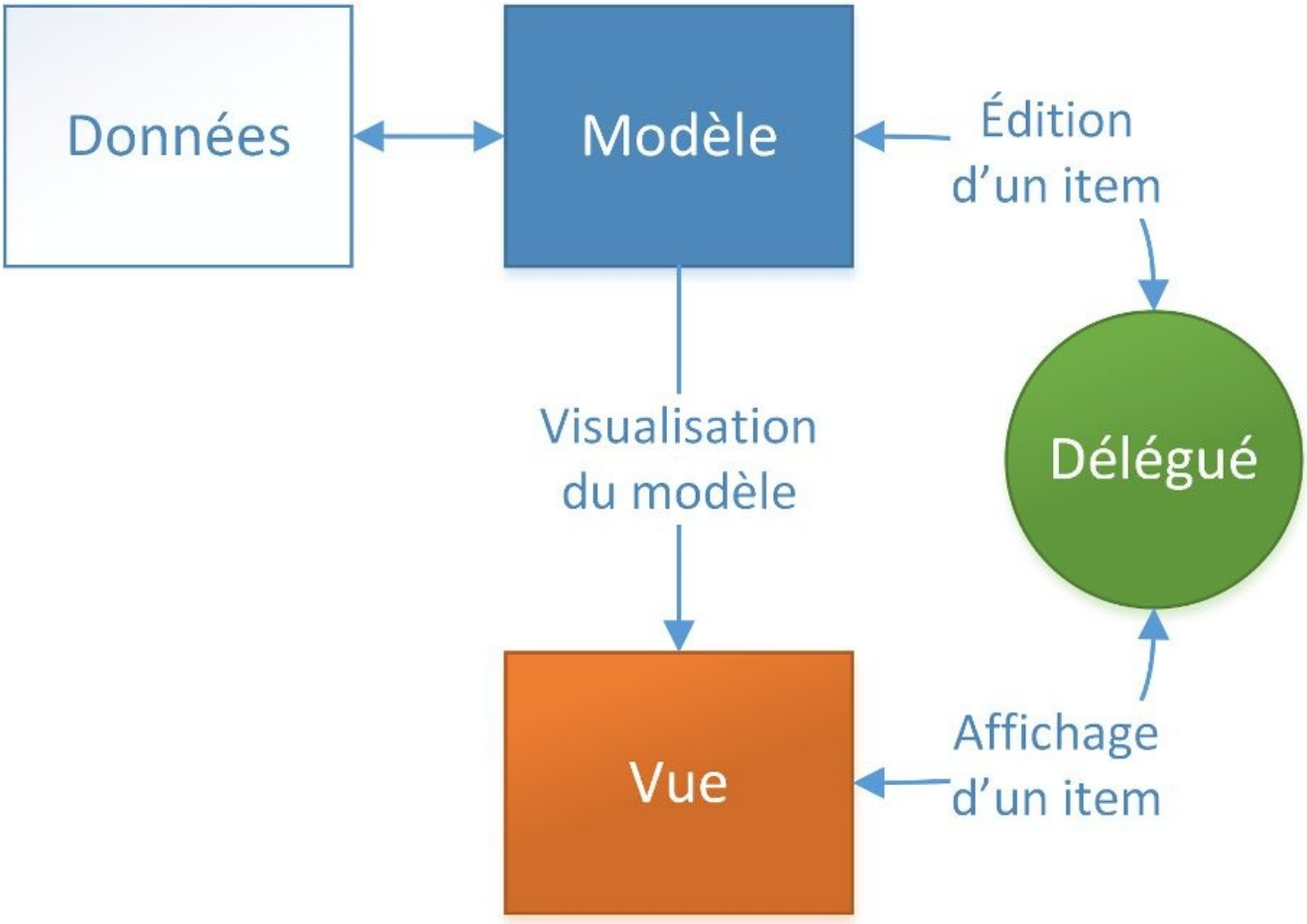
Dans le domaine des interfaces graphiques, une bonne pratique courante est de séparer les données (comme une liste de livres pour BiblioApp, y compris le titre, l'auteur, etc. de chaque livre) de leur présentation (un tableau, une liste, un formulaire…). Qt Quick encourage cette bonne pratique avec le patron de conception *modèle-vue* : le modèle définit les données, la vue les affiche à l'utilisateur. Ainsi, grâce à cette conception, pour proposer plusieurs manières de visualiser les mêmes données, il suffit de changer la vue : un tableau des livres, une grille des couvertures, etc.

À cette définition standard s'ajoute aussi bien avec Qt Quick que PyQt la notion de *délégué*, qui se charge plus particulièrement de l'affichage d'un élément donné, mais aussi de son édition. Ce délégué peut être simple (afficher un texte avec Text) ou plus compliqué (gérer l'édition d'une note avec des étoiles, ce qui n'existe pas de base dans Qt Quick). Ces trois parties — modèle, vue et délégué — forment le patron de conception MVD.

***Note** > Les Qt Quick Controls 2 sont loin d'être aussi avancés que la première version en ce qui concerne les vues. Les développeurs prévoient de combler ces lacunes. Voir la Note au début du chapitre Créer une fenêtre principale pour plus d'informations.*

L'itération actuelle de BiblioApp utilise déjà ce patron de conception, avec une vision très simpliste des choses : le modèle est un tableau JavaScript, la vue exploite Repeater, qui est la vue la plus basique de Qt Quick.

***Figure 16.1 :** Organisation du patron de conception modèle-vue-délégué*



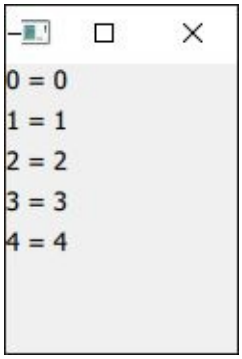
Note *> Le code des exemples de ce chapitre est disponible dans le dossier* `mvd` .

1. Modèle

Un modèle contient des données, sans avoir de forme imposée. Le modèle le plus simple possible est un simple entier : dans ce cas, il ne s'agit que de répéter un élément simple un certain nombre de fois.

```
ColumnLayout {
    Repeater {
        model: 5
        Text { text: modelData + " = " + index }
    }
}
```

Figure 16.2 : *Exemple de modèle entier*



1.1. Structures de données JavaScript

Bien évidemment, un modèle peut aussi être plus complexe, comme un tableau JavaScript, qu'il soit simple (pas de structure sous-jacente) ou compliqué (chaque élément du tableau est un objet, par exemple).

```
var simple = ["Anvers", "Gand", "Charleroi", "Liège", "Bruxelles"]
var complex = [
    {
        "title": "Commentarii de Bello Gallico",
        "author": "Caius Iulius Caesar",
        "genre": "Nonfiction",
        "publisher": "Les Belles Lettres",
        "year": "1926",
        "summary": "Julius Caesar's firsthand account of the Gallic
```

```
Wars",
        "price": "9.00"
    }
]
```

C'est exactement ce type de modèle qui a été utilisé jusqu'à présent dans BiblioApp. Cependant, il souffre de quelques défauts dans le cadre d'une application réaliste, comme la difficulté d'ajouter des transitions ou des problèmes potentiels pour l'ajout d'éléments en parallèle (depuis plusieurs fils d'exécution). De plus, la syntaxe pour accéder à un élément du modèle n'est pas pratique : il faut passer par la propriété attachée `modelData`, les champs ne sont pas accessibles directement.

***Note** > C'est une particularité de ces modèles avec des listes JavaScript de proposer une propriété attachée `modelData` pour accéder à un élément du modèle depuis le code.*

1.2. Modèle de liste

Cependant, Qt Quick fournit des composants spécifiquement prévus comme des modèles, principalement ListModel. Comme son nom l'indique, ce modèle correspond à une liste d'objets, chacun possédant plusieurs champs, ici nommés *rôles*. Cette liste est ordonnée : les éléments ont un indice qui les trie.

ListModel s'utilise de manière très similaire aux modèles JavaScript, sauf pour l'accès aux différents rôles : ils sont directement accessibles, sans syntaxe particulière.

```
ListModel {
    id: bookModel
    ListElement {
        title: "Commentarii de Bello Gallico"
        author: "Caius Iulius Caesar"
        // …
    }
    // …
}

Repeater {
    model: bookModel
    Text {
        text: title + ", " + author
    }
}
```

***Attention** > La propriété attachée `modelData` n'est plus disponible avec ListModel, les rôles sont accessibles directement par leur nom.*

Les opérations sur le modèle se font en JavaScript, tout comme précédemment. Par exemple, vous pourrez écrire ce code JavaScript pour manipuler le modèle de liste :

```
var elt = {
```

❶ "title": "Προμηθεύς δεσμώτης", "author": "Αἰσχύλος", // … } var elt2 { "title": "Edda", "author": "Snorri Sturluson", // … } bookModel.count ❷ // 1 bookModel.append(elt); ❸ // ["Commentarii", "Προμηθεύς"] bookModel.insert(0, elt2) ❹ // ["Edda", "Commentarii", "Προμηθεύς"] bookModel.move(0, bookModel.count - 2, 2) ❺ // ["Προμηθεύς", "Edda", "Commentarii"] bookModel.remove(1) ❻ // ["Προμηθεύς", "Commentarii"] bookModel.set(1, elt) ❼ // ["Προμηθεύς", "Edda"] bookModel.setProperty(0, "title", elt["title"]) ❽ bookModel.clear() ❾ // []

❶ Un élément qui sera ajouté dans le modèle par la suite. Il est défini comme un objet JavaScript, tout comme dans les modèles JavaScript. Les différents champs de l'objet seront traduits en rôles du modèle Qt Quick.

❷ Nombre d'éléments disponibles dans le modèle.

❸ Ajoute un élément à la fin du modèle (son index sera `bookModel.count`).

❹ Ajoute un élément à la position indiquée : les éléments suivants seront donc décalés.

❺ Déplace les deux premiers éléments à la fin du modèle.

```
                      Nombre d'éléments à déplacer
                      ▼
❺ model.move(from, to, number)
           ▲     ▲
           Premier indice à déplacer
                 |
                 Premier indice où déplacer les éléments
```

❻ Enlève l'élément à la position donnée.

❼ Remplace l'élément indiqué par celui donné en deuxième argument.

❽ Remplace, pour l'élément donné, la propriété spécifiée par la valeur du troisième argument.

❾ Vide le modèle.

Note *> Qt Quick fournit également d'autres types de modèle, comme XmlListModel pour utiliser directement un fichier XML externe comme source de données (par exemple, un flux RSS provenant d'un blog). VisualItemModel permet de mélanger le modèle avec le délégué : chaque élément du modèle doit être un composant graphique et est directement affiché. Il est aussi possible d'implémenter ses propres modèles en Python.*

2. Vue

Le composant Repeater est assez rudimentaire pour l'affichage d'un modèle : il ne prédéfinit rien, il n'impose aucune structure sur la séquence d'éléments générés. La vraie puissance du patron MVD s'exprime avec des vues plus intelligentes, comme ListView pour afficher une liste, GridView pour une grille, TableView pour un tableau, voire PathView pour un chemin.

2.1. En liste

Le code actuel de BiblioApp peut se réécrire de manière bien plus élégante pour la liste des livres, dans la partie supérieure de l'écran, à l'aide de ListView. Le modèle et le délégué restent identiques, mais la structure du code de la vue change : au lieu de disposer, au niveau du code de BiblioApp, les éléments en liste, la vue s'en charge elle-même. Graphiquement, l'application ne change pas d'un poil, mais cette restructuration facilitera l'évolution sur le plan visuel.

Note *> Les deux exemples qui suivent sont disponibles dans les dossiers* `vue-listview-biblioapp` *et* `vue-listview-highlight-biblioapp` .

```
Rectangle {
    width: window.width
    // …

    ListView {
        anchors.fill: parent
```

❶ model: bookModel ❷ delegate: Button { ❸ text: title + ", " + author onClicked: // … (comme précédemment) } } }

❶ Tout composant Qt Quick doit être positionné dans l'interface (voir Section 3.3, Système d'ancres). Sans cette instruction, la vue n'aurait pas d'espace déterminé : elle serait alors affichée de manière arbitraire.

❷ Une vue s'associe toujours à un modèle, tout comme Repeater.

❸ De même, le délégué est identique au cas précédent.

Cependant, le code tel quel reste peu intuitif pour le lien avec l'affichage en bas de l'interface : la valeur de chaque champ reste imposée par un clic sur le bouton correspondant au livre. Il est possible d'améliorer fortement l'affichage, en transformant les boutons en une série de champs de texte. Lorsque l'utilisateur en sélectionne un, il est mis en évidence : les détails de ce livre sont indiqués dans la partie édition.

Pour implémenter cette sélection avec surbrillance, l'environnement Qt Quick impose de procéder en trois étapes :

1. Créer un nouveau composant qui servira à mettre en évidence un élément. Il s'agit le plus souvent d'un rectangle, passé à la propriété `highlight` ;

2. Pour que la vue réagisse aux mouvements du clavier (déplacer la sélection avec les flèches directionnelles), il faut lui transférer les événements en provenance du clavier. Pour ce faire, il est nécessaire de lui donner le focus par `focus: true` (plus de détails à la Section 1, Interactions par le clavier) ;

3. Finalement, pour réagir aux clics de la souris, un MouseArea doit être placé au même niveau que Text dans le délégué (comme à la Section 3.5, Gestionnaires de signal). Il leur faut donc un parent commun, puisque le délégué n'est constitué que d'un seul composant. Item est tout désigné, étant donné que ce parent commun ne doit rien faire, si ce n'est afficher ses deux enfants et avoir une taille.

***Note** > Item se situe tout en haut de la hiérarchie des composants Qt Quick visuels, c'est-à-dire ceux qui ont un impact à l'écran (comme un rectangle, mais contrairement à un modèle). Il ne définit qu'un comportement de base (coordonnées, dimensions, principalement).*

```
ListView {
    id: booksList
    anchors.fill: parent
    spacing: 2

    model: bookModel
    delegate: Item {
        width: parent.width
        height: 18

        Text {
            text: title + ", " + author
        }
```

```
        MouseArea {
            onClicked: booksList.currentIndex = index
            anchors.fill: parent
        }
    }

    highlight: Rectangle { color: "lightsteelblue" }
    focus: true
}
```

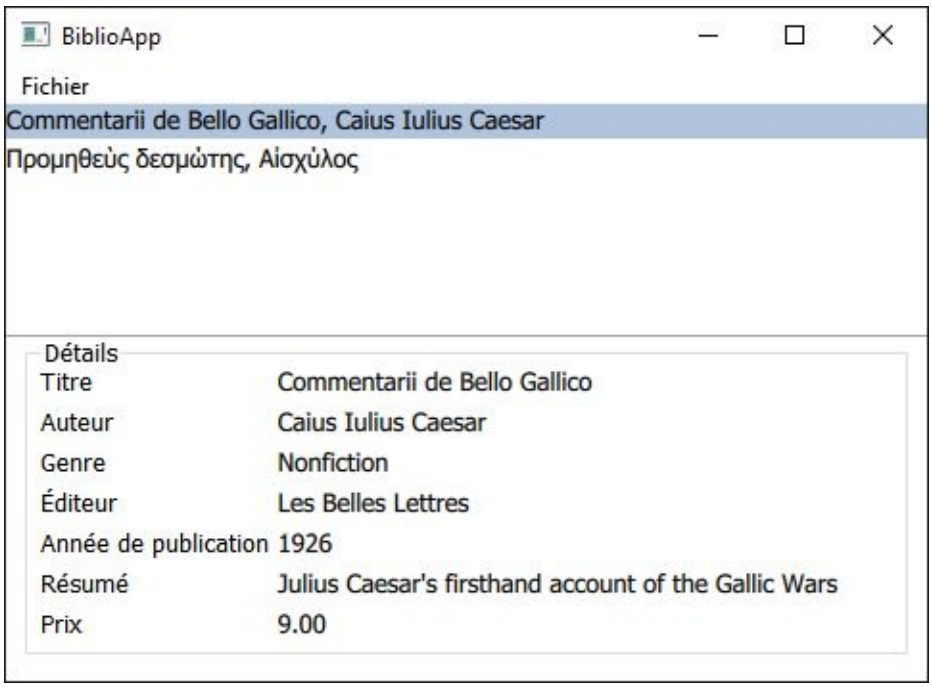
Cette manière de procéder est bien plus propre que précédemment, tant pour l'utilisateur (les boutons étaient détournés de leur usage premier, ce qui nuit à la compréhension rapide de l'interface) que dans le code. Ainsi, le modèle est disponible par `bookModel`, l'indice de l'élément actuellement sélectionné par `booksList.currentIndex`. Il n'en faut pas plus pour effectuer un lien direct entre les deux vues (inutile de définir les valeurs dans la vue depuis les boutons, le lien se fait naturellement).

```
Label { text: "Titre" }
Text { text: bookModel.get(booksList.currentIndex).title }
```

***Note** > Le changement de sélection fonctionne automatiquement avec les flèches directionnelles du clavier : la mise à jour de `booksList.currentIndex` ne doit pas être codée manuellement (à moins que le comportement par défaut ne suffise pas). Par contre, aucun comportement n'est prévu par défaut pour gérer les clics de souris.*

L'élément en surbrillance est toujours indiqué par `booksList.currentIndex` : tout changement dans la valeur de cette propriété déplace automatiquement le composant de mise en évidence. Ce mécanisme est utilisé ici pour gérer les clics de souris de manière naturelle.

***Figure 16.3 :** BiblioApp réécrite avec la sélection dans ListView*



***Note** > Aux niveaux fonctionnel et visuel, cette nouvelle itération de BiblioApp est nettement plus jolie et facile à maintenir, tout en ne représentant qu'une dizaine de lignes supplémentaires par rapport à la version à base de Repeater.*

2.2. En grille et en chemin

La partie supérieure de l'interface pourrait être remplacée par un affichage plus sympathique, en faisant défiler les couvertures des livres : soit sous la forme d'une grille, soit en les faisant glisser sur un chemin prédéfini. Ces vues peuvent s'implémenter assez facilement, avec les développements précédents : le délégué contiendra simplement une image avec sa légende, la vue se chargera de les disposer de manière intelligente — et de gérer la sélection, comme précédemment.

```
Rectangle {
    width: window.width
    Layout.fillHeight: true
    Layout.minimumHeight: booksList.cellHeight + 20
    Layout.maximumHeight: 600

    GridView {
```

❶ id: booksList anchors.fill: parent cellWidth: 140; cellHeight: 260 ❷ model: bookModel delegate: Item { ❸ width: booksList.cellWidth height: booksList.cellHeight Column { ❹ anchors.fill: parent anchors.topMargin: 5 ❺ Image { ❻ source: image

height: booksList.cellHeight * .8 ❼ fillMode: Image.PreserveAspectFit ❽ anchors.horizontalCenter: parent.horizontalCenter } Text { text: title + ", " + author wrapMode: Text.WordWrap ❾ width: parent.width anchors.horizontalCenter: parent.horizontalCenter ❿ horizontalAlignment: Text.AlignHCenter } } MouseArea { ⓫ onClicked: booksList.currentIndex = index anchors.fill: parent } } focus: true ⓬ highlight: Rectangle { color: "lightsteelblue" } } }

❶ Une grille est définie par la vue GridView. Elle dispose ses éléments dans une grille, comme on s'y attend.

❷ Chaque cellule de cette grille a une taille imposée.

❸ Comme précédemment, il faudra rassembler d'un côté l'image et la description, puis de l'autre une zone sensible aux clics. Le composant Item est prévu pour ces usages.

❹ L'image et la description peuvent être affichées dans une colonne.

❺ Pour des raisons esthétiques, on laisse un peu d'espace au-dessus de l'image, pour que le rectangle de sélection soit bien visible.

❻ Une image est affichée avec le composant Image. Le fichier à utiliser est donné dans la propriété `source`.

❼ Pour laisser place à la description, l'image ne prendra que 80 % de la hauteur disponible dans la case.

❽ Afin d'éviter des distorsions de l'image, la largeur n'est pas fixée : la propriété `fillMode` indique qu'elle doit être calculée pour préserver l'aspect de l'image (valeur `Image.PreserveAspectFit`).

❾ Par défaut, le composant Text n'a pas de largeur imposée sur son contenu. Cependant, ici, cela signifierait qu'il pourrait sortir de la case imposée par GridView. Dans ce cas, il faut aussi préciser avec la propriété `wrap` comment les

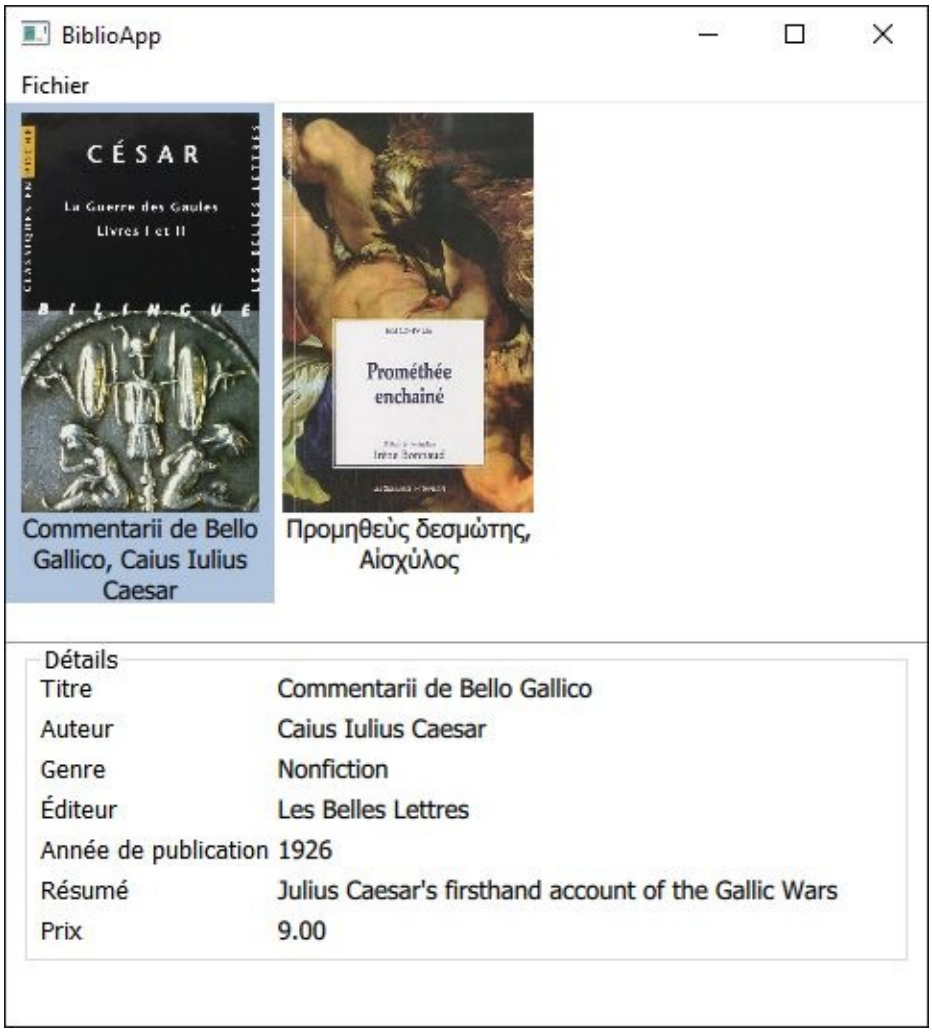
retours à la ligne sont décidés pour les lignes trop longues : ici, la valeur `Text.WordWrap` indique que les coupures doivent avoir lieu entre deux mots.

⑩ Pour des raisons esthétiques, le texte est centré au même niveau que l'image, c'est-à-dire à la moitié de l'image. Il faut, pour cela, définir deux paramètres : une ancre pour le centre horizontal, puis l'alignement du texte.

⑪ Comme précédemment, il faut activer manuellement le défilement avec le clavier et la souris.

⑫ L'élément actuellement sélectionné dans la vue est mis en évidence par ce rectangle, dont les dimensions s'adapteront à la cellule à surligner.

Figure 16.4 : *BiblioApp avec une grille des livres encodés*



Cependant, cette manière d'afficher et de faire défiler des images ne fait pas très moderne. PathView permet bien plus de fantaisies à ce niveau, en alignant les éléments sur une courbe, avec les effets de déplacement qui correspondent.

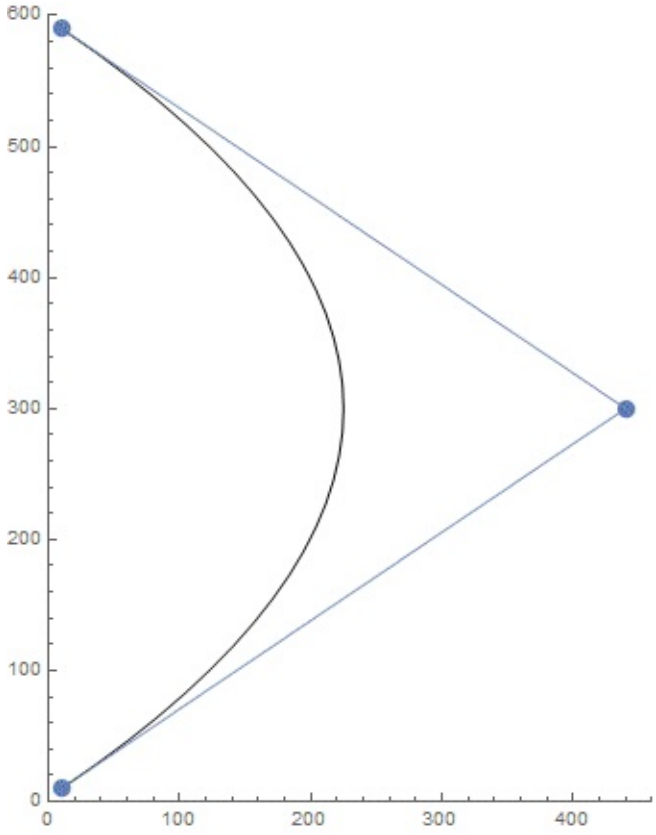
```
Rectangle {
    width: window.width
    Layout.fillHeight: true
    Layout.minimumHeight: 600
    Layout.maximumHeight: 600

    PathView {
        id: booksList
        anchors.fill: parent

        focus: true
        Keys.onLeftPressed: decrementCurrentIndex()
```

❶ Keys.onRightPressed: incrementCurrentIndex() model: bookModel delegate: Item { … } path: Path { ❷ startX: 10; startY: 10 PathQuad { x: 10; y: booksList.height - 10 controlX: booksList.width controlY: booksList.height / 2 } } }

La propriété la plus importante d'un PathView est, sans conteste, le chemin que les éléments devront suivre. Il commence par un point de départ et se poursuit par un ou plusieurs points de passage ou de contrôle (la courbe passe impérativement par les premiers et est déformée par les seconds). Ici, PathQuad indique que le lien avec le point précédent suivra une courbe de Bézier, dont le point final est donné dans les propriétés `x` et `y`, avec un point de contrôle. Ainsi, ce chemin ira de `(startX, startY)` à `(x, y)`, en suivant une courbe de Bézier dont le point de contrôle est `(controlX, controlY)` (voir Figure 16.5).

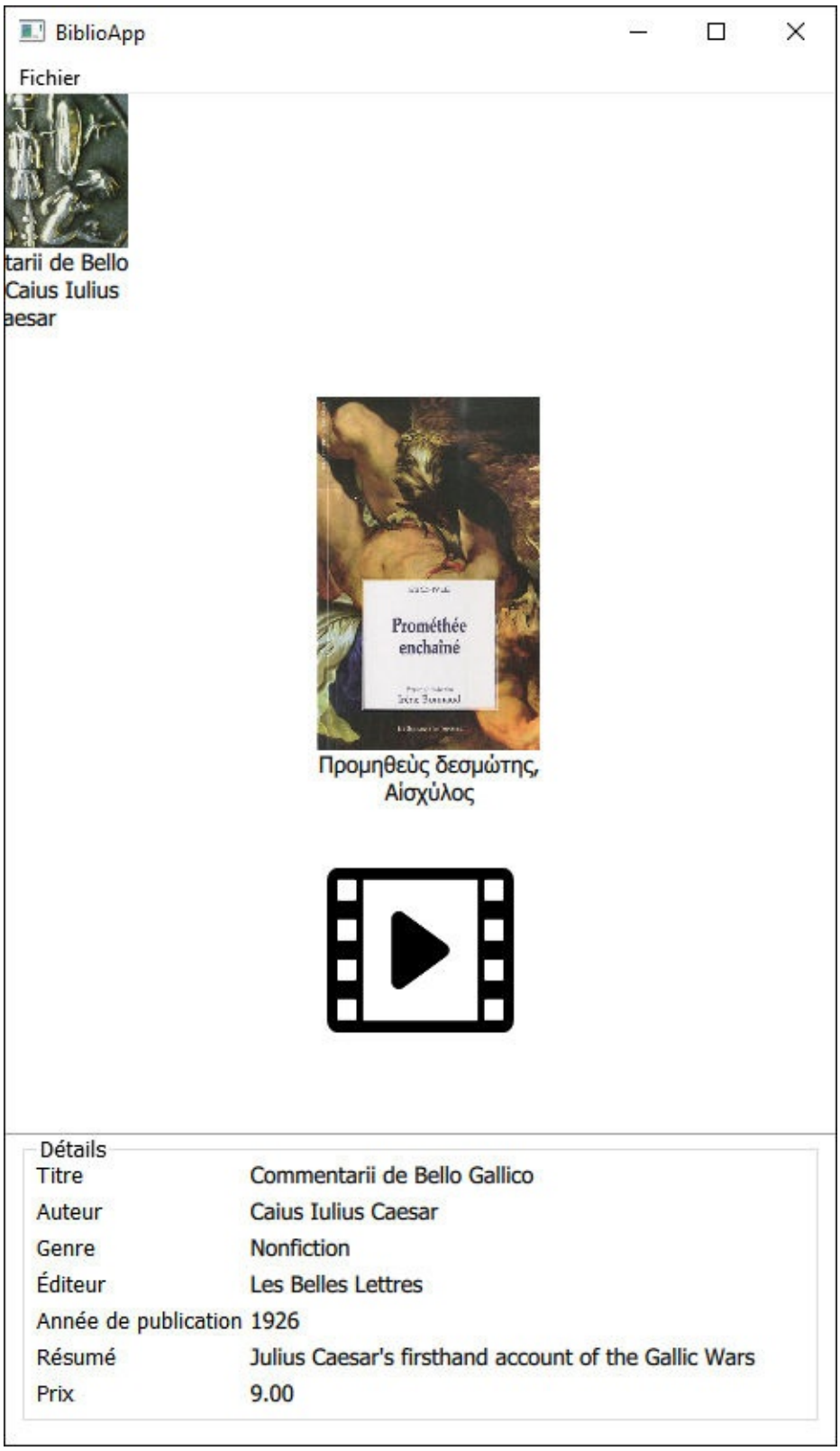
Figure 16.5 : *Courbe suivie par les éléments de PathView par rapport aux trois points donnés : un début, une fin, puis un point de contrôle*

❷



Contrairement aux autres vues, PathView ne gère pas nativement les interactions
❶ avec le clavier, il faut les implémenter soi-même. Ces mécanismes sont détaillés à la Section 1, Interactions par le clavier.

Figure 16.6 : *BiblioApp avec un défilement en courbe des livres encodés (video)*



Note *> Il est évidemment possible d'ajuster encore l'affichage, avec une courbe plus complexe ou des transformations sur les éléments lors du défilement, par exemple en*

changeant leur opacité ou leur taille. Le composant PathAttribute sert justement à ces fins. Les possibilités offertes par PathView sont vastes, mais assez difficiles à mettre en œuvre.

2.3. En tableau

Pour finir, une dernière vue est très utilisée avec Qt Quick : TableView, qui propose un tableau. Celui-ci est fortement lié au modèle : TableView requiert d'indiquer toutes les colonnes une par une, en précisant le rôle à afficher (c'est-à-dire la variable de chaque élément du modèle), dans une TableViewColumn. Ces colonnes remplacent les délégués précédemment utilisés ; la notion de délégué reste présente, mais seulement au niveau d'une colonne, pour personnaliser l'affichage d'un champ donné (remplacer un nombre par des étoiles dans une colonne Note, par exemple).

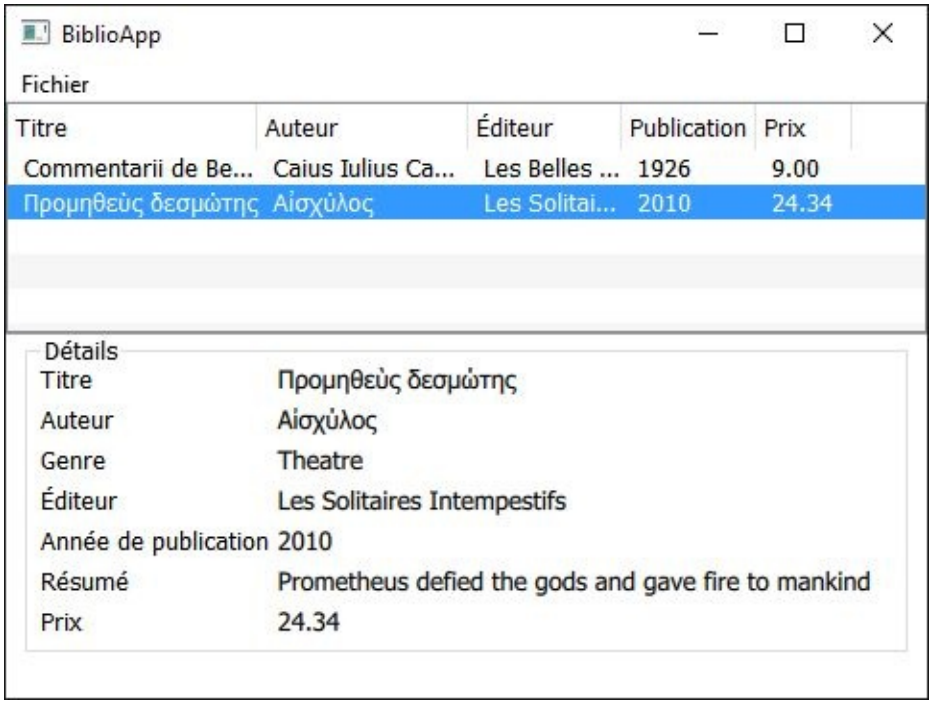
```
TableView {
    id: booksList
    anchors.fill: parent

    model: bookModel

    TableViewColumn {
        role: "title"
        title: "Titre"
        width: 130
    }
    // …
}
```

À la différence des autres vues, pour récupérer l'élément actuellement sélectionné, il faut utiliser la propriété `currentRow` (et non `currentIndex`). Ce champ prend la valeur `-1` lorsqu'aucune ligne n'est sélectionnée, ce qui génère immédiatement une série d'erreurs dans la console pour indiquer que le programme tente de lire un élément qui n'existe pas dans le modèle. Pour éviter cela, le code ne peut pas directement lire à l'adresse indiquée : il doit vérifier avant l'affichage si un élément est sélectionné ou imposer la sélection par défaut d'un élément au démarrage (avec le gestionnaire de signal `Component.onCompleted`).

```
Label { text: "Titre" }
Text {
    text:
        if(booksList.currentRow >= 0)
            bookModel.get(booksList.currentRow).title;
        else
            "";
```

```
}
```

Figure 16.7 : *BiblioApp avec un tableau des livres encodés*



3. Édition du modèle

Pour que cette application soit réellement utile, une fonctionnalité manque à l'appel : la modification du contenu. Pour rendre cela possible, il faut remplacer l'affichage de texte par des composants d'édition : TextField pour le texte libre, ComboBox pour choisir le genre dans une liste prédéfinie, SpinBox pour déterminer l'année.

```
TextField { // Au lieu de Text
    id: fieldTitle
    text: … bookModel.get(booksList.currentRow).title …
    Layout.minimumWidth: window.width * 0.6
}
// …
```

La liste des genres est stockée dans un modèle à part. Pour retrouver l'identifiant d'un genre dans la liste depuis son nom, il est commode de créer une fonction attachée à ce modèle : `findGenreIndexByName`. Celle-ci opère en itérant sur tous les éléments du modèle et en les comparant avec le texte donné.

```
ListModel {
    id: genreModel
    ListElement { text: "(None)" }
    // …

    function findGenreIndexByName(name) {
        for(var i = 0; i < count; i++) {
            if(name === get(i).text) {
                return i;
            }
        }
        return -1;
    }
}
```

***Note** > Il serait assez simple de proposer à l'utilisateur de changer cette liste de genres : par exemple, en ouvrant une nouvelle fenêtre depuis un menu, dont le contenu serait assez similaire à la fenêtre principale.*

Finalement vient la partie la plus intéressante : les modifications effectives de la bibliothèque. L'utilisateur peut effectuer trois actions, donc trois boutons dans l'interface : l'un pour enregistrer les modifications du livre courant ; un autre pour créer un nouveau livre (le livre courant devient vide) ; le dernier pour supprimer le livre actuellement sélectionné. Ainsi, les modifications effectuées dans les champs d'édition

ne sont jamais directement visibles dans le modèle affiché dans la partie haute de l'interface, l'utilisateur doit valider cette opération.

```
RowLayout {
```

❶ spacing: 60 anchors.horizontalCenter: parent.horizontalCenter anchors.top: detailsGroup.bottom anchors.topMargin: 20 Button { text: "Nouveau" onClicked: { booksList.currentRow = -1; ❷ booksList.selection.deselect(); } } Button { text: "Sauvegarder" onClicked: { var contents = { title: fieldTitle.text, author: fieldAuthor.text… }; ❸ if (booksList.currentRow >= 0) { ❹ bookModel.set(booksList.currentRow, contents); } else { ❺ bookModel.append(contents); } } } Button { text: "Supprimer" enabled: booksList.currentRow >= 0 ❻ onClicked: if (booksList.currentRow >= 0) bookModel.remove(booksList.currentRow); ❼ } }

❶ Les trois boutons sont affichés en bas de l'interface, alignés et espacés.

❷ Pour créer un nouveau livre, la seule étape à effectuer est d'invalider la sélection actuelle dans le tableau — sinon, les données seront reportées comme modifications dans un livre existant. De par les liens qui ont été créés entre le tableau et la partie d'édition, tous les champs seront remis à leur valeur par défaut pour ce nouveau livre, sans code supplémentaire.

❸ La sauvegarde doit distinguer deux cas : soit l'utilisateur édite un élément existant, soit il en crée un nouveau. Dans tous les cas, les fonctions à appeler prennent en argument la même structure de données décrivant les valeurs à modifier.

❹ Si une sélection est effectuée dans le tableau, alors la sauvegarde remplace un élément existant avec la fonction `set`.

❺ Sinon, il faut créer un nouvel élément en fin de modèle avec `append`.

❼ La suppression s'effectue uniquement si un élément est sélectionné avec `remove`.

❻ Pour fignoler cet exemple, le bouton de suppression n'est activé que lorsqu'il peut effectuer une action ; sinon, il est grisé.

Un point manque encore dans l'interface : quand un nouveau livre est créé, sa ligne doit être sélectionnée dans le tableau. Autrement, si l'utilisateur effectue de nouvelles modifications et clique sur Sauvegarder… une nouvelle ligne sera créée. Cependant, cette opération ne peut pas s'effectuer au niveau du bouton : le modèle vient à peine d'être modifié, la vue n'a pas encore eu le temps de l'être (elle ne le sera qu'après l'exécution de la fonction modifiant le modèle). Pour contourner ce problème, le gestionnaire de signal `onRowCountChanged()` de TableView vient à la rescousse : il est appelé dès que le nombre de lignes de la vue a changé, c'est-à-dire pile-poil au moment intéressant. À cet instant, la sélection est déplacée vers le dernier élément, celui qui vient d'être ajouté.

```
onRowCountChanged: {
    selection.clear();
    if (rowCount > 0) {
        selection.select(rowCount - 1);
    }
}
```

***Note** > La condition `rowCount > 0` est nécessaire dans le cas où l'on supprime le dernier élément de la liste : il ne reste plus aucun élément, impossible d'en sélectionner un. Sans cette condition, Qt Quick affiche une erreur dans la console (même si l'application fonctionne normalement).*

17

Persistance des données avec LocalStorage

Niveau : intermédiaire

Objectifs : stocker des informations localement dans une base de données relationnelle

Prérequis : Premiers pas avec Qt Quick, Présentation de JavaScript, Utiliser la méthodologie modèle - vue

L'application BiblioApp développée jusqu'ici répond au cahier des charges minimal : il est possible de créer des livres, visualiser l'état actuel de la bibliothèque et en modifier le contenu. Par contre, dès la fermeture du programme, toutes ces informations sont perdues. Pour résoudre ce problème, il est nécessaire de développer une couche de persistance des données.

Le stockage de données informatiques ne peut pas se faire de trente-six manières : en bonne approximation, le monde se divise entre le stockage dans des fichiers (soit en texte, c'est-à-dire lisibles avec un simple éditeur de texte, soit en binaire) et dans des bases de données (le plus souvent relationnelles[21]).

Côté Qt Quick, seules les bases de données relationnelles sont directement accessibles, à l'aide de l'API LocalStorage. Cette dernière est très fortement inspirée de l'API Web SQL Database définie pour HTML5, qui permet de stocker des données sous cette forme dans un navigateur web. Côté implémentation, ce module donne accès à une base de données SQLite, accessible depuis l'extérieur de l'application Qt Quick au besoin.

***Note** > Cette API est assez légère, elle ne permet notamment pas d'exploiter toute la flexibilité du module Qt SQL. Elle se révèle cependant largement suffisante pour bien des cas d'utilisation. Pour aller plus loin dans le stockage des données, il est nécessaire de recourir à des extensions en Python (Chapitre Communiquer avec Python).*

1. Deux mots sur SQL

Le langage SQL a été développé pour communiquer avec la base de données relationnelle ; son principal avantage est que les mêmes requêtes peuvent être utilisées (relativement) indépendamment du système de gestion de base de données sous-jacent, puisque le langage a été normalisé. SQL est un langage de description très complet des requêtes que l'on souhaite effectuer. Il permet notamment d'interroger la base de données, pour obtenir par exemple une liste de livres d'un éditeur donné ; il est suffisamment général pour effectuer toute opération nécessaire sur la base de données, comme son initialisation.

De manière générale, une base de données relationnelle stocke ses informations sous la forme de tableaux à deux entrées. Chacun de ces tableaux correspond à une *table*. Lors de la création d'une de ces tables, les *colonnes* doivent être nommées, avec le type de données à y stocker. Ensuite, les opérations d'*insertion* ajouteront des *enregistrements* dans ces tables. Finalement, les opérations de lecture portent sur des sous-ensembles de ces tables : on peut sélectionner tout ou partie des colonnes (il suffit de donner la liste des colonnes intéressantes), tout ou partie des enregistrements (à l'aide d'un critère de sélection).

1.1. Création d'une table

Dans une base de données, une table se crée avec une requête `CREATE TABLE`. Par exemple, pour créer une table des genres littéraires :

```
CREATE TABLE genres (

❶ genre_id INTEGER PRIMARY KEY, ❷ genre TEXT ❸ );
```

❶ La requête `CREATE TABLE` prend deux arguments : le nom de la table à créer (ici, `genres`), puis une liste de colonnes entre parenthèses `()`.

La première colonne se nomme `genre_id` et sera de type entier (`INTEGER`). Ce numéro identifiera de manière unique chaque genre et permettra d'y faire référence ultérieurement. Cette manière de procéder a plusieurs avantages par rapport à l'approche précédente, qui est de copier *in extenso* le genre au niveau de chaque

livre :

- dans l'application, il est facile de déterminer tous les genres littéraires définis (le contenu de cette table) et donc de remplir la liste déroulante correspondante ;

- si on remarque une faute de frappe dans un genre, il suffit de la corriger une seule fois (dans la table `genres`) plutôt que devoir parcourir chaque livre stocké en base de données.

❷

L'enregistrement correspondant à chaque livre contiendra uniquement ce numéro pour identifier le genre.

Le mot clé `PRIMARY KEY` donne la clé primaire de la table : en d'autres termes, le moteur de base de données comprend que ce champ sera utilisé pour faire référence à un genre et se prépare à optimiser des requêtes qui cherchent un genre à partir de son identifiant. Une seule clé primaire est autorisée par table. (Il est possible de demander au moteur d'optimiser d'autres types de requêtes par des index.)

***Note** > Pour SQLite, le moteur utilisé par Qt Quick, un entier utilisé comme clé primaire sera automatiquement incrémenté pour chaque nouvelle entrée dans la table.*

La deuxième colonne correspond à la représentation textuelle du genre lui-même,
❸ qui sera présentée à l'utilisateur. Il s'agit donc, en termes SQL, d'un `TEXT`, une chaîne de caractères. SQLite gère automatiquement l'encodage du texte inséré.

Une base de données ne peut jamais contenir deux tables avec le même nom : le moteur d'exécution renvoie alors une erreur. Cependant, au moment de lancer l'application, il faut vérifier que toutes les tables nécessaires existent. Justement, la norme SQL permet d'indiquer que la table ne doit être créée que s'il n'en existe pas encore avec ce nom :

```
CREATE TABLE IF NOT EXISTS genres (
    genre_id INTEGER PRIMARY KEY,
    genre TEXT
);
```

***Note** > D'autres types de données sont possibles que des entiers ou du texte. SQLite définit également les nombres réels `REAL` et des données binaires brutes `BLOB`. La norme SQL prévoit bon nombre d'autres types, sans qu'ils soient gérés de la même manière*

sur tous les serveurs de bases de données. Par exemple, les autres moteurs SQL définissent le type `VARCHAR`*, c'est-à-dire une chaîne de caractères de longueur maximale fixée (ce qui permet au moteur d'optimiser son fonctionnement et de répondre aux requêtes plus rapidement).*

***Note** > Pour supprimer une table, il faut utiliser la commande* `DROP TABLE`*, qui s'utilise de la même manière. Si la table n'existe pas, le moteur d'exécution renverra une erreur : pour contourner ce cas, il faut ajouter les mots clés* `IF EXISTS`*.*

```
DROP TABLE genres;
DROP TABLE IF EXISTS genre;
```

1.2. Ajout de données

Une fois une table créée, la première opération à effectuer est probablement de la remplir. Pour ce faire, SQL propose la requête `INSERT INTO`. Elle ne garantit pas que le genre inséré sera unique, cependant !

La syntaxe de base est la suivante :

```
INSERT INTO genres
```

❶ (genre) ❷ VALUES ('Nonfiction'); ❸

❶ Le premier paramètre de la requête d'insertion est le nom de la table qui recevra les nouvelles données.

Ensuite vient, entre parenthèses, le nom des colonnes dans lesquelles on veut insérer une valeur. Ici, l'identifiant `genre_id` n'est pas précisé : la base de données se chargera elle-même d'en attribuer un.

❷ ***Note** > Au sens strict, cette partie n'est pas utile (puisque le moteur de bases de données connaît les colonnes de la table concernée), mais elle rend le code bien plus lisible. Si cette partie est absente, l'ordre des colonnes sera celui de définition de la table — qui peut être assez loin des insertions, ce qui complique la lecture du code.*

❸ Finalement, après le mot clé `VALUES` viennent les valeurs à insérer, entre parenthèses : la valeur de chaque colonne est séparée de la suivante par une virgule.

1.3. Opérations de lecture

Une autre opération courante sur une base de données est la lecture d'opérations, avec la requête `SELECT`. Celle-ci est très puissante, et nous nous contenterons ici de l'effleurer.

Par exemple, pour sélectionner toutes les entrées d'une table donnée, cette requête retournera toutes les colonnes (ce qui est indiqué par `*`) de la table `genres` ;

```
SELECT * FROM genres;
```

Cette requête peut être rendue plus sélective pour ne retourner qu'une certaine liste de colonnes, par exemple :

```
SELECT genre_id, genre FROM genres;
```

Cependant, `SELECT` ne montre son vrai visage que lorsque l'on ajoute des contraintes sur les données à récupérer. Elles sont indiquées après le mot clé `WHERE`. Par exemple, pour sélectionner les cinquante premières entrées (en supposant que les identifiants ont été attribués sans trou à partir de l'unité) :

```
SELECT * FROM genres WHERE genres.genre_id <= 50;
```

La fonctionnalité la plus puissante du langage SQL est la *jointure*, très utile dans les applications réelles : quand un enregistrement d'une table fait référence à un enregistrement d'une autre table par le biais de son identifiant, il est possible de fusionner les deux lignes dans une seule requête (ce qui évite plusieurs requêtes de sélection). Si la table des livres est définie comme ceci :

```
CREATE TABLE IF NOT EXISTS books (
    book_id INTEGER PRIMARY KEY,
    title TEXT,
    author TEXT,
    genre_id INTEGER
);
```

alors une sélection avec jointure sur les genres pour sélectionner le livre d'identifiant 1 pourra s'écrire avec l'opérateur `JOIN` comme :

```
SELECT title, author, genre
```

```
FROM books
JOIN genres
    ON genres.genre_id = books.genre_id
WHERE books.book_id = 1;
```

Ici, il est nécessaire de préciser la table d'appartenance de chaque colonne (c'est-à-dire `books.genre_id` et `genres.genre_id` au lieu de simplement `genre_id` à chaque fois) pour éviter la confusion entre les deux.

***Note** > Dans ce cas, puisque les deux attributs sur lesquels la jointure s'applique portent le même nom (`genre_id`), la clause `ON genres.genre_id = books.genre_id` est optionnelle. L'opérateur n'est alors plus `JOIN`, mais `NATURAL JOIN`.*

1.4. Mise à jour

La mise à jour concerne uniquement les éléments qui existent déjà. Au niveau syntaxique, la requête `UPDATE` est un mélange entre une sélection d'informations et un ajout. Par exemple, pour changer l'auteur d'un livre d'identifiant donné, la requête sera la suivante :

```
UPDATE books
SET author = "J. K. Rowling"
WHERE book_id = 1;
```

1.5. Suppression d'éléments

En SQL, la suppression d'éléments reprend la syntaxe de la sélection, en changeant l'opérateur en `DELETE`. Par exemple, pour supprimer tous les livres de la base de données :

```
DELETE FROM books;
```

Il est évidemment possible d'être plus sélectif, par exemple en ne supprimant qu'un seul livre par son identifiant :

```
DELETE FROM books
WHERE book_id = 1;
```

1.6. Notion de transaction

SQL comprend également la notion de transaction : toutes les requêtes d'une transaction donnée doivent être exécutées, sinon aucune ne le sera. Ce mécanisme permet de garantir la cohérence d'une base de données : une transaction n'est jamais effectuée à moitié. Notamment, en cas d'erreur lors de l'exécution d'une commande SQL, toute la transaction est automatiquement arrêtée.

Il s'agit d'ailleurs du concept de base pour l'API LocalStorage, qui démarre et arrête les transactions automatiquement.

1.7. Aller plus loin

Cette petite introduction à SQL est très parcellaire et ne donne que les bases nécessaires à l'écriture de petites requêtes : elle ne rentre même pas dans les détails relationnels, essentiels à la conception d'une base de données viable. En outre, pour la simplicité du développement et du déploiement, nous avons fait le choix d'utiliser un moteur de base de données très léger : SQLite. Il a l'avantage d'être toujours inclus avec PyQt (et est le seul accessible directement depuis Qt Quick).

Parmi ses limitations, ce dernier ne gère que très peu de types de données. Des moteurs plus complets (plus lourds), comme MariaDB, Oracle Database, Microsoft SQL Server, PostgreSQL, disposent d'une liste de types bien plus exhaustive, ce qui leur permet d'optimiser le stockage des données : par exemple, pour du texte, SQLite ne dispose que de `TEXT`, soit un texte de longueur arbitraire ; à peu près tous les moteurs implémentent cependant `VARCHAR`, qui est une chaîne de caractères avec une longueur maximale.

SQL n'est pas non plus l'unique manière de voir les bases de données : le paradigme *non relationnel* (NoSQL) commence à prendre de l'importance. La grande différence est que ces bases de données ne traitent plus de la notion de relation (qui existe, par exemple, entre la table des livres et celle des genres : un livre possède un genre) : elles peuvent être remplacées par des arêtes comme dans un graphe (Neo4j, OrientDB, OpenLink Virtuoso, etc.)… ou rien du tout (base de données orientée colonne à la Cassandra, clé-valeur à la Redis). Parfois, la notion de base est le document, comme une page web ou un produit dans un catalogue : chaque entrée peut alors se structurer (un prix est la combinaison d'un nombre et d'une monnaie, un produit peut avoir une liste de mots clés), voire avoir un schéma adapté (CouchDB, MongoDB) ; les bases de données orientées objets stockent directement des objets, sans autre forme de procès (ZODB, VelocityDB, etc.). Certaines bases de données se spécialisent dans le stockage d'informations géographiques (PostGIS) ou XML (MarkLogic, Sedna, etc.).

[21] Le registre de Windows correspond, par exemple, à une base de données hiérarchique.

2. LocalStorage

Les requêtes et commandes SQL qui viennent d'être présentées ne sont pas utilisables telles quelles dans une application Qt Quick : elles sont passées en argument à des méthodes JavaScript, qui se chargent de leur exécution sur la base de données à travers le moteur SQLite.

La première étape consiste à créer une connexion vers la base de données avec la méthode `LocalStorage.openDatabaseSync`. Pour ce faire, deux paramètres sont essentiels : le nom de cette base de données (par exemple, `BiblioApp`), puis un numéro de version pour la base de données, qui sert à gérer les changements dans les définitions des tables. Deux autres paramètres sont demandés : une description de la base de données ainsi qu'une taille (en octets) ; aucun des deux n'est actuellement utilisé par Qt Quick (ils sont simplement recopiés dans un fichier de configuration pour référence future).

```
var name = "BiblioApp"; // Nom
var ver = "1.0"; // Version
var desc = "BiblioApp SQL database"; // Description
var size = 1000000; // Taille
var db = LocalStorage.openDatabaseSync(name, ver, desc, size);
```

***Note** > Il n'est pas possible de supprimer ou de lister les bases de données créées par le module LocalStorage ! Les bases de données sont stockées à un endroit qui dépend de la plateforme :*
`C:\Users\Utilisateur\AppData\Local\Application\QML\OfflineStorage\Dat`
sous Windows,
`/home/Utilisateur/.local/share/Application/QML/OfflineStorage/Databas`
sous Linux, `/Users/Utilisateur/Library/Application Support/Application/QML/OfflineStorage/Databases` sous macOS.

Le nom de l'application correspond à la commande exécutée pour lancer le code Qt Quick. Par exemple, depuis Qt Creator, il correspond au nom du projet ; si le lanceur est codé en Python et lancé comme un script Python (avec une commande comme `python main.py`), ce nom d'application sera `python`.

Lors de l'initialisation de cet objet `db`, la méthode `openDatabaseSync` peut prendre un cinquième et dernier argument : une fonction, souvent anonyme , appelée uniquement pour créer la base de données. Cette fonction anonyme prend en argument la base de données en cours de création. Il faut alors lancer une transaction par la méthode

`db.transaction()`, qui prend en argument une nouvelle fonction anonyme : ce deuxième niveau d'imbrication pourra enfin effectuer des requêtes en base de données, à l'aide de l'argument de transaction (conventionnellement nommé `tx`). Chaque requête est effectuée par la méthode `tx.executeSql()`.

Note *> Le code source final de cet exemple est disponible dans le projet* `basique-biblioapp` .

```
var db = LocalStorage.openDatabaseSync(
    name, ver, desc, size,
    function (db) {
        db.transaction(function (tx) {
            tx.executeSql('CREATE TABLE genres (' +
                                'genre_id INTEGER PRIMARY KEY,' +
                                'genre TEXT' +
                          ');');
            tx.executeSql('CREATE TABLE books (' +
                                'book_id INTEGER PRIMARY KEY,' +
                                'title TEXT,' +
                                'author TEXT,' +
                                'genre_id INTEGER' +
                          ');');
            tx.executeSql('INSERT INTO genres (genre) ' +
                          'VALUES ("(None)"), ("Biography") […]');
        });
        db.changeVersion("", ver);
    }
);
```

Attention *> Quand cette fonction est utilisée, le numéro de version est par défaut défini à une valeur vide : en d'autres termes, l'argument de version passé à* `LocalStorage.openDatabaseSync` *est ignoré quand une fonction lui est passée en argument. Il faut donc changer manuellement ce numéro de version avec* `db.changeVersion(ancien, nouveau)`.

Ensuite, pour le reste de l'exécution du programme, il est possible de créer des transactions depuis l'objet `db` par sa méthode `transaction()`, comme précédemment. La valeur de retour de `tx.executeSql` dépend de la requête exécutée (voir Tableau 17.1) et permet de vérifier que l'opération s'est déroulée avec le succès attendu. Par exemple, pour l'insertion d'un genre dans la table correspondante, `executeSql` renvoie l'identifiant du nouvel enregistrement, qui peut alors être utilisé dans la suite du programme.

```
db.transaction(
    function (tx) {
        var id_nonfiction =
```

```
                tx.executeSql("INSERT INTO genres " +
                              "(genre) " +
                              "VALUES ('Nonfiction');");
        tx.executeSql("INSERT INTO books " +
                      "(title, author, genre_id) " +
                      "VALUES ('Προμηθεὺς δεσμώτης', " +
                              "'Αἰσχύλος', " + id_nonfiction + ");");
    }
);
```

Tableau 17.1 : Valeur de retour de `tx.executeSql`

Requête SQL	Valeur importante
`SELECT`	`returnedRows.rows.length` : nombre de valeurs sélectionnées
	`returnedRows.rows.item(i)` : ligne `i` de la sélection
`INSERT`	`insertedId` : identifiant de la nouvelle ligne
`UPDATE`, `DELETE`	`rowsAffected` : nombre de lignes modifiées

La fonction `tx.executeSql` permet également l'utilisation de paramètres positionnels dans les requêtes. Concrètement, il s'agit d'utiliser des points d'interrogation `?` dans la requête SQL, puis de passer un tableau de valeurs en deuxième argument. Les points d'interrogation seront remplacés par les valeurs correspondantes dans le tableau.

```
tx.executeSql("INSERT INTO books " +
              "(title, author, genre_id) " +
              "VALUES (?, ?, ?);",
              ["Προμηθεύς δεσμώτης", "Αἰσχύλος", id_nonfiction]);
```

Cette manière de procéder a deux avantages. Le premier est que la construction de la requête est plus simple : il n'est pas nécessaire de faire appel à de la concaténation de chaînes de caractères. Le second est plus pragmatique : cette manière de faire est bien plus sécurisée. Même en donnant des valeurs très compliquées aux paramètres, il ne sera pas possible de modifier le sens de la requête (par exemple, la faire modifier des données sensibles). Ce genre de faille se nomme injection SQL.

3. Lien avec l'application

Pour que BiblioApp ait accès à la base de données, le composant Qt Quick devra mémoriser la variable de base de données (précédemment nommée `db`) de manière globale à l'application. Une solution est de *définir une nouvelle propriété* au niveau de la fenêtre. Ceci se fait avec le mot clé `property`, suivi du type de la propriété (ici, `var`) et du nom voulu, à la même hauteur que les autres propriétés :

```
ApplicationWindow {
    id: window
    // …
    property var db
```

Ensuite, le reste de l'activité par rapport à la base de données est encapsulé dans une série de fonctions JavaScript, qui peuvent être incluses directement dans la fenêtre principale.

***Note** > La définition de nouvelles propriétés est expliquée à la Section 2, Interface d'un composant et propriétés.*

La première de ces fonctions concerne l'ouverture d'une connexion à la base de données, mais aussi sa création. La table des genres doit être remplie (BiblioApp ne fournit aucun moyen de l'éditer), celle des livres peut l'être pour montrer directement à l'utilisateur ce dont l'application est capable.

```
function openDB() {
    db = LocalStorage.openDatabaseSync(
        "BiblioApp", "1.0", "BiblioApp SQL database", 1000000,
        function (db) {
            db.transaction(function (tx) {
                tx.executeSql('CREATE TABLE genres (' +
                                  'genre_id INTEGER PRIMARY KEY,' +
                                  'genre TEXT' +
                              ');');
                tx.executeSql('CREATE TABLE books (' +
                                  'book_id INTEGER PRIMARY KEY,' +
                                  'title TEXT,' +
                                  'author TEXT,' +
                                  'genre_id INTEGER' +
                                  // …
                              ');');
                tx.executeSql('INSERT INTO genres (genre) VALUES ' +
                              '("(None)"), ("Biography"), ("Crime"),
' +
```

```
                                '("Science-fiction"), ("Fantastic"), '
+
                                '("Historic"), ("Nonfiction"),
("Theatre")');
                    // …
                })
                db.changeVersion("", "1.0");
            }
        );
}
```

Cette fonction doit être appelée avant tout appel sur la base de données, par exemple lors de son lancement, avec le gestionnaire de signal `onCompleted` sur le composant racine (ApplicationWindow dans la plupart des exemples).

3.1. Lecture des données

Ensuite, il faudra lire chaque genre stocké en base de données et l'afficher à l'utilisateur avant que l'interface de BiblioApp soit utilisable (lors de son lancement, en d'autres termes). Une deuxième fonction se charge de cette tâche : une requête SQL est effectuée lors de la transaction, puis chaque élément est transféré dans le modèle `genreModel`. Il sera nécessaire d'y définir un deuxième rôle pour stocker l'identifiant de chaque genre (l'identifiant interne au modèle n'a potentiellement aucun sens en base de données : elle n'impose pas la continuité, il peut y avoir des trous dans la numérotation, notamment à cause de suppressions).

```
function populateGenres() {
    db.transaction(function(tx) {
        var genres = tx.executeSql("SELECT * FROM genres");
        for (var i = 0; i < genres.rows.length; ++i) {
            var item = genres.rows.item(i);
            genreModel.append({ text: item.genre, id: item.genre_id
});
        }
    });
}
```

***Note** > Avec ces changements, les listes déroulantes des genres et des livres n'arriveront plus à accéder aux données. Le problème vient du fait que le composant ComboBox ne sélectionne aucun rôle à afficher. Il faut donc lui indiquer spécifiquement le rôle à lire et à afficher à l'utilisateur.*

```
ComboBox {
    id: fieldGenre
    // …
    textRole: "text"
```

```
}
```

La même opération est effectuée pour les livres.

3.2. Modifications

Dernière étape : lors de l'utilisation de l'application, les modifications apportées devront être répercutées en base de données. Pour ce faire, le plus simple est de se connecter sur les boutons de sauvegarde et de suppression et d'appeler le code transférant les nouvelles données en base.

Les fonctions correspondantes peuvent s'écrire comme suit. Leur argument fait référence à la ligne de l'élément modifié dans le modèle côté Qt Quick.

```
function addBook(id) {
    db.transaction(function(tx) {
        var book = bookModel.get(id);
        var gid =
genreModel.findGenreDatabaseIndexByName(book.genre);
        tx.executeSql('INSERT INTO books ' +
                      '(title, author, genre_id…) ' +
                      'VALUES (?, ?, ?…);',
                      [book.title, book.author, gid…]);
    });
}
function updateBook(id) {
    db.transaction(function(tx) {
        var book = bookModel.get(id);
        var gid =
genreModel.findGenreDatabaseIndexByName(book.genre);
        tx.executeSql('UPDATE books ' +
                      'SET title = ?, author = ?…' +
                      'WHERE book_id = ?;',
                      [book.title, book.author…, book.id]);
    });
}
function removeBook(id) {
    db.transaction(function(tx) {
        var book = bookModel.get(id);
        tx.executeSql('DELETE FROM books WHERE book_id = ?;',
[book.id]);
    });
}
```

Ensuite, ces fonctions doivent être appelées depuis les boutons. Dans le cas de la suppression, l'appel à la base de données doit se faire *avant* la suppression de l'entrée

dans le modèle, sinon l'identifiant donné à la fonction ne sera plus valide. Au contraire, dans le cas d'un ajout ou d'une modification, l'appel à la base de données doit se faire *après* l'ajout dans le modèle, sinon la fonction n'aura rien à lire.

```
Button {
    text: "Sauvegarder"
    onClicked: {
        var contents = …
        if (booksList.currentRow >= 0) {
            bookModel.set(booksList.currentRow, contents);
            updateBook(booksList.currentRow);
        } else { // Create an item.
            bookModel.append(contents);
            addBook(bookModel.count - 1);
        }
    }
}
Button {
    text: "Supprimer"
    onClicked:
        if (booksList.currentRow >= 0) {
            removeBook(booksList.currentRow);
            bookModel.remove(booksList.currentRow);
        }
}
```

Quid de l'efficacité ?

Ajouter une base de données dans l'application pour charger les livres et genres d'origine a fortement augmenté le temps de lancement — même sur une machine puissante, l'application passe maintenant par une fenêtre blanche avant d'afficher son contenu. Lors de la mise à jour de la base, la situation est différente, car les modifications sont assez légères. Cependant, pour une application complexe, ces modifications pourraient se traduire par un plus grand nombre de requêtes, donc un manque de réactivité de l'application lors de la répercussion de ces informations.

En premier lieu, il faut éviter de recopier tout le modèle à la fermeture de l'application, c'est-à-dire de vider entièrement les tables en base de données et d'écrire chaque genre et chaque livre. Cela gaspillerait inutilement des ressources, vu que seul un sous-ensemble des données a été modifié. La fermeture de l'application pourrait ainsi prendre très longtemps, ce qui risque d'exaspérer l'utilisateur (à qui le système d'exploitation pourrait proposer de fermer la fenêtre non réactive, la pensant plantée !).

Une manière d'améliorer la situation est de marquer les informations nouvelles (une technique appelée *dirty bit* en anglais) et de n'écrire que ces parties modifiées en base de données, de manière groupée. Par exemple, au lieu d'envoyer une requête pour chaque nouveau genre ou livre inséré, le rôle `dirty` (à créer) des éléments modifiés prend une valeur booléenne fausse. À un certain moment (par exemple, lorsque l'utilisateur ferme la fenêtre d'édition des genres), toutes ces modifications sont envoyées d'un coup au serveur, dans une seule requête SQL (ou encore après, en même temps qu'une série de livres). Il faut cependant faire attention à bien ajouter les nouveaux genres avant les nouveaux livres, ces derniers pouvant faire référence aux premiers.

Quid des mises à jour du schéma ?

Le module LocalStorage n'est, actuellement, pas prévu pour gérer les mises à jour du schéma de la base de données (en d'autres termes, la structure des tables), contrairement à l'API HTML5 dont il s'inspire librement : à l'ouverture d'une connexion avec la base de données, la fonction `LocalStorage.openDatabaseSync` permet de spécifier un numéro de version et renvoie une erreur si le numéro demandé ne correspond pas à celui de la base de données. Cependant, les codes d'erreur ne sont pas documentés, c'est-à-dire qu'ils pourraient changer d'une version à l'autre du module sans notification.

Pour passer outre cette limitation, la seule solution est de gérer soi-même les numéros de version à l'intérieur de la base de données, en créant une table dont le seul enregistrement contiendra la version actuelle de la base de données complète (ou un numéro par table). Ainsi, lorsque la fonction `openDatabaseSync` renvoie une instance de la base, avant toute autre opération, l'idée est de vérifier les numéros de version : s'ils ne correspondent pas, il faut faire une mise à jour du schéma de la base de données pour qu'il corresponde à celui attendu par la nouvelle version de l'application.

La manière la plus courante d'écrire le code qui implémente ces mises à jour est de le faire incrémentalement : si une table est en version `1.0`, mais si la version attendue par l'application est la `1.3`, alors le code effectuera la mise à jour de la `1.0` vers la `1.1`, puis la `1.2` et enfin la `1.3`. Ainsi, vous ne devrez écrire que des scripts de mise à jour assez simples et en nombre réduit. Les numéros de version du schéma de la base de données ne doivent pas forcément correspondre à ceux de l'application.

18

Créer un composant réutilisable

Niveau : intermédiaire

Objectifs : développer une application de manière modulaire plutôt que monolithique, avec plusieurs fichiers

Prérequis : Premiers pas avec Qt Quick, Présentation de JavaScript

Jusqu'à présent, le code de BiblioApp n'est pas réutilisable : tout le contenu de l'interface est présent dans un seul fichier (`main.qml`), sans possibilité de changer le comportement depuis l'extérieur de ce fichier principal. Cependant, certaines parties pourraient être utiles à l'extérieur de BiblioApp : l'accès à la base de données est indépendant de la manière d'écrire l'interface, tout comme le modèle des genres (qui ne diffère d'une liste que par des fonctions de recherche). La seule dépendance entre le stockage des données et l'application BiblioApp est l'interface de programmation à respecter pour la communication harmonieuse de ces deux parties.

D'un autre point de vue, si une autre application cherche à intégrer BiblioApp, la réutilisation du code sera difficile. Par exemple, pour gérer une bibliothèque de quartier, les fonctionnalités incluront une consultation des livres existants, l'ajout de livres à la collection (ces deux-là sont très proches de la version actuelle de BiblioApp), mais aussi le prêt (qui utilisera la même base de données). Dans ce cas, une bonne pratique de développement serait de réutiliser l'interface de BiblioApp, mais de changer la manière de stocker les informations pour la coupler avec le prêt.

Bien évidemment, Qt Quick propose de créer ses propres composants réutilisables : soit directement en QML (l'objet de ce chapitre), soit en Python (voir chapitre Communiquer avec Python). Ils offrent la même latitude que tous les composants utilisés jusqu'à maintenant (ApplicationWindow pour la fenêtre principale, ListModel pour stocker les données, SplitView pour séparer l'interface en deux) : affichage, paramétrage par des propriétés, gestionnaires de signal.

Une fois cette modularisation effectuée, les composants peuvent être distribués sous la forme d'une extension Qt Quick, ce qui les rend utilisables par la directive `import` dans des applications externes.

1. Composant statique

Pour être appelable de l'extérieur, un composant doit être placé seul dans un fichier, dont le nom correspond au composant ; ainsi, le composant Menu est implémenté dans un fichier nommé `Menu.qml`. Ce nom doit commencer par une capitale, sous peine de ne pas être reconnu comme tel ; d'ailleurs, tous les composants utilisés jusqu'à présent suivent cette convention.

Ensuite, le contenu de ce composant s'écrit de la même manière que l'interface principale : il dérive d'un autre composant (souvent, d'un Rectangle), dont le contenu est ensuite adapté (valeurs des propriétés, composants enfants, etc.).

***Note** > Si le composant a besoin d'une existence à l'affichage qui ne correspond pas à un rectangle, il vaut mieux utiliser Item (dont dérivent tous les composants graphiques), qui définit toute une série de propriétés de base[22] mais pas d'apparence. MouseArea est par exemple dans ce cas : une zone sensible doit exister à l'écran (avoir une existence graphique), mais se superpose à un autre composant.*

Tout au contraire, les modèles ne sont pas des composants graphiques : ils héritent de Component. Les fonctionnalités disponibles sont alors basiques, mais nécessaires dans toute application Qt Quick : posséder un identifiant, gérer la fin du chargement du composant, principalement.

Une fois défini, un composant est directement utilisable par d'autres fichiers QML du même dossier, sans formalité supplémentaire. S'il se situe ailleurs dans la hiérarchie, il devient nécessaire d'importer le dossier qui contient le composant souhaité : par exemple, pour des composants rangés dans un dossier `app/components` quand le fichier principal est dans le dossier `app/main`, il faut utiliser la directive `import ../components`.

```
app
  components
    main
      main.qml
    components
      ReuseableComponent.qml
```

Dans le cadre de BiblioApp, on peut facilement extraire l'interface principale, qui hérite de ApplicationWindow, pour la déplacer dans un composant `BiblioApp.qml` (attention à la capitale !). L'interface principale du fichier `main.qml` se réduit alors à

peau de chagrin, avec la configuration de base d'ApplicationWindow :

Note *> Le code source de cet exemple et les fichiers d'éléments associés sont disponibles dans le dossier* `biblioapp-decompose`  .

```
import QtQuick 2.5

BiblioApp {
    width: 450
    height: 400
    visible: true
    title: "BiblioApp"
}
```

La modularisation du code peut continuer avec le modèle des genres, qui n'est rien d'autre qu'une liste avec deux fonctions JavaScript. En effet, cette partie du code pourrait être utile pour bon nombre d'applications, mémoriser une liste d'éléments et effectuer des recherches par nom étant des choses relativement courantes. Cette opération se fait en créant un fichier `GenreModel.qml`, qui contient le code de ce composant.

```
import QtQuick 2.5

ListModel {
    function findGenreIndexByName(name) { … }
    function findGenreDatabaseIndexByName(name) { … }
}
```

Le fichier principal `main.qml` n'étant plus qu'une coquille vide, il faut que le code de BiblioApp se retrouve quelque part : il s'agit du composant BiblioApp (utilisé plus haut), c'est-à-dire du fichier `BiblioApp.qml`. Il correspond en très grande partie au fichier principal des chapitres précédents, avec certains paramètres déplacés dans `main.qml` et le modèle des genres GenreModel.

```
ApplicationWindow {
    id: window
    // …
    GenreModel { id: genreModel }
    ListModel { id: bookModel }
    // …
}
```

---

[22] Une position *x* et *y*, des dimensions *width* et *height*, des ancres *anchors*.

2. Interface d'un composant et propriétés

En faisant hériter un composant d'un autre, toute l'interface publique de ce dernier devient disponible. Ainsi, dans l'exemple BiblioApp, on a pu récupérer facilement la propriété `title` de ApplicationWindow. Cependant, si l'on veut redistribuer le composant, cette manière de procéder n'est pas très propre : l'interface accessible n'est pas celle que vous avez définie, elle n'a donc pas forcément du sens pour le composant exposé (même si elle en a pour le composant dont il hérite). La solution est alors de dériver d'Item, puisque ce composant ne définit pas d'interface (outre le strict minimum pour l'affichage) : on peut alors définir soi-même les propriétés faisant partie de l'interface que l'on souhaite exposer.

Une nouvelle propriété est définie à l'aide du mot clé `property`, suivi du type de la propriété[23] : les principaux types sont `int` pour un entier, `string` pour une chaîne de caractères, `bool` pour une valeur binaire vraie ou fausse, ainsi que `var` pour un type indéfini (pour la liste complète, voir la documentation). Ensuite vient le nom attribué à la propriété. En quatrième lieu, mais de manière optionnelle, on peut également fournir une valeur par défaut après le symbole `:`.

```
property type nom [: valeur par défaut]
```

Pour le composant BiblioApp, afin de faciliter la réutilisation du composant dans une interface plus grande, les modèles (qui stockent les livres et les genres) devraient être fournis par l'utilisateur du composant BiblioApp. Pour ce faire, on peut définir deux propriétés dans ce composant.

***Note** > Le code source de cet exemple et les fichiers d'éléments associés sont disponibles dans le projet `2-biblioapp-proprietes` .*

```
ApplicationWindow {
    id: window
    property var genreModel
    property var bookModel
    // …
}
```

Les noms de ces propriétés correspondent aux identifiants précédents : il ne sera pas nécessaire d'adapter le code, il fera automatiquement référence à ces propriétés (les identifiants n'existant plus). Du côté utilisateur, dans le fichier `main.qml`, il faudra spécifier les deux modèles :

```
BiblioApp {
    // …
    bookModel: ListModel {}
    genreModel: GenreModel {}
}
```

On a ainsi un composant BiblioApp générique et réutilisable (les types des modèles ne sont plus imposés par le composant BiblioApp), sur lequel l'utilisateur vient greffer les modèles qu'il a développés (le cas échéant).

---

[23] Cette obligation provient de Qt : la notion de type est très importante en C++, qui ne dispose pas, de base, de mécanismes dynamiques comme Python ou JavaScript.

3. Signaux

Le dernier mécanisme spécifique aux composants réutilisables est la création de signal (Qt Quick s'occupe de la partie gestionnaire de signal automatiquement). La syntaxe employée est à la frontière entre celle des propriétés et celle des fonctions JavaScript : le mot clé nécessaire est `signal`, suivi du nom du signal. Ensuite, de manière optionnelle, le signal peut transmettre des arguments, indiqués entre parenthèses : il faut préciser le type (comme pour une propriété). Ainsi, tous ces exemples sont des définitions autorisées de signaux :

```
signal mySignal   // Aucun paramètre (parenthèses facultatives)
signal mySignal() // Aucun paramètre
signal mySignal(int x) // Un seul paramètre : x
```

Le gestionnaire de signal correspondant sera `onMySignal` : il commence toujours par `on` et reprend le nom du signal (en transformant la première lettre en capitale au besoin). Le code JavaScript associé au gestionnaire aura accès aux variables définies comme arguments du signal.

L'émission d'un signal se fait en l'appelant comme une fonction, par exemple :

```
mySignal(4);
```

Note *> À la différence de PyQt, l'émission d'un signal ne se fait pas avec une méthode* `emit()`.

19

Communiquer avec Python

Niveau : intermédiaire à avancé

Objectifs : communiquer entre les parties Qt Quick et Python d'une application

Prérequis : Premiers pas avec Qt Quick, Présentation de JavaScript, Utiliser la méthodologie modèle - vue, Créer un composant réutilisable

Qt Quick, en tant que tel, est un environnement prévu presque exclusivement pour le développement d'interfaces graphiques. Dès qu'il s'agit de sortir de ce cadre, on bute rapidement sur ses limites :

- l'accès aux bases de données se fait par LocalStorage, un module relativement simple… mais restreint et sans possibilité de réutiliser du code existant (par exemple, en cas d'ajout d'une interface graphique par-dessus une application existante) ;

- il est impossible d'écrire des fichiers (pour de la lecture, il est possible d'exploiter XMLHttpRequest et la transparence réseau) ;

- certaines opérations sont extrêmement lentes à effectuer en JavaScript (à moins de passer beaucoup de temps à peaufiner une implémentation), alors qu'il est souvent possible de réutiliser une bibliothèque Python déjà optimisée. D'ailleurs, une très grande partie des composants livrés avec Qt Quick n'est pas implémentée en QML, mais bien directement en C++ (il aurait été possible de le faire en Python), pour des raisons de performance.

Pour toutes ces choses et bien d'autres, il s'avère nécessaire de sortir du cadre strict de Qt Quick et du code JavaScript uniquement. Ça tombe bien, tout est prévu : écrire des extensions en Python est une chose relativement aisée.

PyQt offre deux manières de procéder pour assurer la communication entre les deux environnements :

- d'un côté, l'exposition de propriétés contextuelles, c'est-à-dire de nouvelles variables (et, potentiellement, un objet avec ses méthodes) ;

- de l'autre, la création d'un nouveau composant Qt Quick (comme un modèle), qui

pourra par la suite s'utiliser exactement comme s'il avait été défini directement en QML.

Pour ces deux méthodes, il est nécessaire de charger l'interface Qt Quick depuis un script Python par l'intermédiaire d'une instance du moteur d'exécution Qt Quick. Ce dernier offre de multiples points de contact entre Python et Qt Quick — notamment pour intégrer une interface Qt Quick complète dans une interface PyQt existante.

JavaScript ou Python ?

Jusqu'à présent, tous les exemples utilisaient du JavaScript. Cependant, avec ce chapitre, un troisième langage vient se mêler à la partie : Python. La question du choix de langage pour implémenter telle ou telle partie se pose donc : si l'interface doit être décrite en QML, les interactions pourront être écrites en JavaScript ou en Python.

Les meilleures pratiques avec Qt Quick sont, pour des applications en production, de limiter la partie JavaScript à l'interface graphique pure (y compris les interactions), les calculs plus lourds étant tous effectués à l'extérieur, en Python. Cependant, la frontière entre interface graphique et calcul reste toujours floue : bon nombre d'éléments ne se classent pas sans ambiguïté d'un côté ou de l'autre. Le choix de langage dépendra fortement des développeurs et de leurs affinités avec l'un ou l'autre environnement.

Dans le cadre d'un prototype, néanmoins, la situation est plus claire : l'objectif est de produire une interface sur laquelle il est facile d'itérer, d'essayer de nouvelles choses. Mélanger du code JavaScript et Python ajoute une dose de rigidité dans l'application, qu'il vaut mieux éviter dans ce contexte de développement rapide et de changements profonds réguliers.

***Note** > Le code des exemples de ce chapitres est disponible dans le dossier `python`*  .

1. Lancement d'une fenêtre Qt Quick par Python

Pour lancer une application Qt Quick depuis du code Python, on procède de la même manière que pour une application PyQt traditionnelle : création d'un objet QApplication (nommé, par exemple, `app`) ❶, puis de l'interface elle-même ❷ et, pour finir, exécution du code avec `app.exec_()` ❹. La différence est que, pour instancier les composants de l'interface, il faut passer par un moteur d'exécution Qt Quick, une instance de QQmlApplicationEngine. Son constructeur prend en argument le nom du fichier Qt Quick à charger, c'est-à-dire le fichier principal (qui n'est généralement pas un composant Qt Quick réutilisable).

```
import sys

from PyQt5.QtWidgets import QApplication
from PyQt5.QtQml import QQmlApplicationEngine


if __name__ == "__main__":
    app = QApplication(sys.argv)
```

❶ engine = QQmlApplicationEngine("main.qml") ❷ engine.quit.connect(app.quit)) ❸ sys.exit(app.exec_()) ❹

***Note** > Pour que les appels à `Qt.quit()` fonctionnent dans le code JavaScript, il est nécessaire de connecter un signal dans le code Python lançant l'interface. C'est à cela que sert la ligne `engine.quit.connect(app.quit)` ❸.*

***Note** > QQmlApplicationEngine impose que le composant racine chargé soit de type fenêtre, comme Window ou MainWindow. Charger une interface dont la racine est un Rectangle n'affichera rien.*

***Note** > QQmlApplicationEngine est une couche d'abstraction au-dessus du moteur d'exécution Qt Quick, QQmlEngine. Par rapport à ce dernier, QQmlApplicationEngine facilite la création d'une interface visible à l'utilisateur sans devoir effectuer trop d'opérations. Outre cette distinction, leur utilisation est identique.*

Dans la suite de ce chapitre, le moteur d'exécution Qt Quick sera utilisé pour communiquer des variables accessibles globalement dans le code QML ou pour créer

des composants : dans ce cas, le code QML ne peut pas s'exécuter correctement tant que les opérations correspondantes n'ont pas été effectuées (son exécution afficherait une panoplie de messages d'erreur pas toujours compréhensibles). En d'autres termes, donner le nom du fichier QML à charger au constructeur de QQmlApplicationEngine n'a pas de sens. La méthode `load()` est alors utile : elle permet, après avoir créé l'instance du moteur d'exécution, de charger le code QML à lancer.

```
app = QApplication(sys.argv)
engine = QQmlApplicationEngine()
engine.quit.connect(app.quit)
engine.load("main.qml")
sys.exit(app.exec_())
```

Intégrer une partie Qt Quick dans une application Qt Widgets

Pour migrer des applications imposantes de Qt Widgets vers Qt Quick, il est courant de procéder de manière progressive, en mélangeant les deux : on commence par passer en Qt Quick des fonctionnalités assez spécifiques (un composant dans l'interface), puis petit à petit le reste de l'interface, jusqu'à la fenêtre principale. Ainsi, à tout instant, l'interface complète reste utilisable par l'utilisateur : la migration peut s'étaler sur plusieurs années, au fil de l'eau ; toute l'application ne doit pas être réécrite d'une traite.

Selon le type d'application, une réécriture avec Qt Quick n'est pas envisageable ; par contre, les besoins de l'application sont tels que les utilisateurs sont rapidement amenés à écrire leurs petites modifications dans le code, qui peuvent alors se faire à l'aide de Qt Quick : elles seront intégrées à l'interface globale de la même manière.

Cette intégration est facilitée par la classe QQuickWidget. Son but est d'intégrer une vue Qt Quick dans une interface à base de widgets (avec, toutefois, un surcoût en termes de performance).

```
w = QQuickWidget()
w.setSource(QUrl.fromLocalFile("Component.qml"))
w.show()
```

Note *> Cette classe QQuickWidget ne peut pas être utilisée pour créer des fenêtres : autrement dit, le fichier QML chargé ne peut pas utiliser un composant Qt Quick comme Window ou MainWindow. À part ce point, n'importe quelle scène Qt Quick peut être intégrée dans une application à base de widgets. À ce titre, QQuickWidget est l'exact inverse de QQmlApplicationEngine.*

Pour accéder au moteur d'exécution QQmlEngine (notamment afin de le contrôler comme expliqué ci-après), une manière de procéder est de le passer au constructeur de QQuickWidget :

```
engine = QQmlEngine("Component.qml")
w = QQuickWidget(engine, parentWidget)
```

Sinon, le contrôleur de QQuickWidget en instanciera un, qui sera disponible par la méthode `w.engine()`. L'inconvénient sera que toutes les opérations seront effectuées *après* l'instanciation du composant Qt Quick (ce qui peut poser problème si certaines données importantes sont transmises par ce biais).

***Note** > Dans ce cas, l'affichage est géré par la classe QQuickWidget : le moteur d'exécution ne doit pas gérer cet aspect. Par conséquent, on utilisera QQmlEngine au lieu de QQmlApplicationEngine.*

Le sens inverse, c'est-à-dire intégrer une partie Qt Widgets ou vue graphique dans une application Qt Quick, n'est pas possible actuellement.

2. Communication par des propriétés contextuelles

Une fois le moteur d'exécution Qt Quick accessible, on peut s'intégrer en profondeur avec l'application lancée. Le moyen le plus simple de le faire est de créer des *propriétés contextuelles*, c'est-à-dire des propriétés dont la valeur est imposée à l'extérieur de Qt Quick, par le code Python. Quand ces propriétés sont ajoutées au contexte global, elles correspondent à des variables globales.

Une propriété contextuelle est définie sur un contexte, généralement le contexte à la racine, accessible à travers le moteur d'exécution de Qt Quick : `engine.rootContext()`. Une fois un contexte obtenu, la méthode `setContextProperty` lie le nom d'une telle propriété (donnée en premier argument) avec sa valeur (deuxième argument).

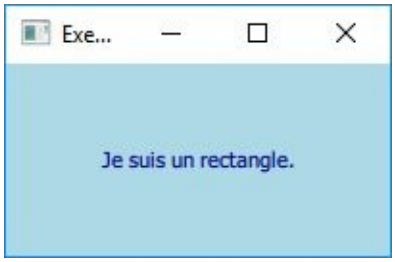
Côté Python :

```
app = QApplication(sys.argv)
engine = QQmlApplicationEngine()
engine.quit.connect(app.quit)
engine.rootContext().setContextProperty("pythonText", "Je suis un rectangle.")
engine.load("main.qml")
sys.exit(app.exec_())
```

Côté QML :

```
// …
Text {
    // …
    text: pythonText
    color: "darkblue"
}
```

Figure 19.1 : *Texte affiché depuis une propriété contextuelle définie dans le code Python*



Les propriétés ainsi partagées peuvent avoir n'importe quel type Python : des nombres, des listes… voire des objets, ce qui permet d'accéder à des méthodes Python depuis Qt Quick. Ces méthodes doivent cependant être définies comme des slots (la classe doit donc hériter de QObject), sans quoi elles ne pourront pas être reconnues.

Côté Python :

```
from PyQt5.QtCore import QObject, pyqtSlot


class TextProvider(QObject):
    @pyqtSlot(result=str)
    def provide(self):
        return "Je viens d'une méthode Python."


# …
engine.rootContext().setContextProperty("textProvider",
TextProvider())
```

Côté QML :

```
Text { // …
    text: textProvider.provide()
}
```

Grâce à ce mécanisme, les données à charger par défaut dans BiblioApp peuvent être stockées n'importe où, puis injectées dans l'application Qt Quick. Dans l'exemple, les données seront stockées au niveau du script lançant l'application, mais cette dernière peut très bien lire les données par défaut d'un fichier de configuration. Les premiers genres et livres ne sont donc plus codés en dur dans les fichiers QML.

Ainsi, deux propriétés contextuelles seront définies : `initialGenres` pour les genres littéraires et `initialBooks` pour les livres. Deux types de base de Python sont ainsi utilisés : d'une part une liste de chaînes de caractères, d'autre part une structure plus complexe, une liste de dictionnaires. PyQt détecte automatiquement ces types et s'occupe du nécessaire pour les rendre accessibles côté Qt Quick (pas besoin de définir de classe héritant de QObject ou de slot, notamment).

Note *> Le code source de cet exemple et les fichiers d'éléments associés sont disponibles dans le projet* `biblioapp-contexte`  .

```
initialGenres = ["(None)", "Biography"…]
```

```
initialBooks = [{'title': "Commentarii de Bello Gallico",
                 'author': "Caius Iulius Caesar"…}, {…}]

if __name__ == "__main__":
    engine = QQmlApplicationEngine()
    engine.rootContext().setContextProperty("initialGenres",
initialGenres)
    engine.rootContext().setContextProperty("initialBooks",
initialBooks)
```

Ensuite, côté Qt Quick, la seule chose à faire est de récupérer ces deux variables lors de l'initialisation de la base de données et de réécrire les requêtes SQL en les utilisant (plutôt que d'utiliser des requêtes fixes). Par exemple, pour les genres :

```
function openDB(initialGenres, initialBooks) {
    // …
    tx.executeSql('INSERT INTO genres (genre) VALUES ' +
                  initialGenres.map(function(val) {
                      return "(\"" + val + "\")";
                  }).join(", ") + ";");
    // …
```

***Note** > La liste des genres pourrait très bien être directement utilisée dans l'application en tant que modèle. Cependant, l'interaction entre Qt Quick et Python ne se déroule pas de la manière la plus intuitive qui soit : aucune mise à jour n'est reportée de l'autre côté. Pour ce faire, il vaut mieux passer par une classe qui s'inscrit dans la hiérarchie des modèles de PyQt ou par un composant qui expose des propriétés (au sens de PyQt).*

3. Communication par composant

Les composants Qt Quick peuvent directement être définis par du code QML, comme au chapitre Créer un composant réutilisable, mais également en Python. Dans le cadre de BiblioApp, les seuls composants qui auraient à gagner à être implémentés en Python ont trait à la gestion des données : le module LocalStorage permet un stockage dans une base de données SQLite, mais nécessite d'écrire des requêtes SQL directement dans le code QML. En passant cette partie en Python, le code QML ne dépend plus de la manière de stocker ces données (dans des fichiers, par exemple) : elles pourront être mémorisées dans des fichiers JSON (en utilisant le module JSON de Python) très facilement… ou bien dans une base de données complexe, sans que l'application Qt Quick doive être modifiée pour passer de l'une à l'autre manière de stocker les informations. L'avantage de JSON, dans ce contexte, est que la traduction des structures de données Python se fait en une ligne de code et que les fichiers produits sont lisibles dans à peu près n'importe quel langage de programmation sur n'importe quel ordinateur.

En pratique, l'objectif est donc de créer un modèle PyQt pour lequel chaque opération est répercutée sur un fichier (dont le nom est précisé par une propriété du modèle). Ce modèle est équivalent à une liste, puisque les éléments sont simplement listés les uns après les autres, sans structure particulière si ce n'est leur ordre[24]. L'implémentation devrait donc être une nouvelle classe (le modèle à créer) qui hérite de QAbstractListModel[25]. Cependant, cela n'est pas obligatoire : les vues Qt Quick utilisent une API totalement différente de celles des modèles de PyQt.

En pratique, il faut donc que la classe Python implémente les différentes méthodes des modèles Qt Quick :

- `insert(self, row, value)` : insère la valeur `value` à la position `row` du modèle ;
- `append(self, value)` : insère la valeur `value` à la fin du modèle (à la position `row = count()`) ;
- `get(self, row)` : récupère la valeur à la position `row` ;
- `set(self, row, value)` : change la valeur à la position `row` pour `value` ;
- `remove(self, row)` : supprime la valeur à la position `row` ;

- en sus, la propriété `count` compte le nombre de lignes du modèle (tout changement doit être notifié par le signal `countChanged`).

Toutes ces méthodes doivent être accessibles depuis Qt Quick. Le plus simple, pour ce faire, est de les définir comme slots ; `count` aura un rôle particulier, puisqu'il s'agit d'une propriété, non d'une fonction.

***Note** > Les composants écrits de la sorte, en Python, peuvent également avoir un rendu graphique, contrairement au modèle que l'on souhaite développer. Pour ce faire, la classe correspondante doit hériter de QQuickItem, ce qui leur permet de dessiner leur apparence en faisant appel au graphe de scène Qt Quick (à l'intérieur de la fonction `paint`) ou directement à du code OpenGL (il faut alors hériter de QQuickFrameBufferObject et implémenter la méthode `createRenderer`).*

Pour BiblioApp, deux modèles seront requis, au vu des différences dans la structure des données à conserver : l'un pour gérer les genres (une simple liste de chaînes de caractères), l'autre pour les livres proprement dits (un tableau à deux entrées : un livre correspond à une ligne, chaque colonne à une information mémorisée sur ce livre). Chacun de ces modèles exposera, en plus de ses données, un champ `file` pour mémoriser l'emplacement du fichier JSON où les données seront recopiées. À chaque modification d'un des deux modèles, le fichier correspondant sera mis à jour ; au lancement de l'application, le fichier correspondant sera lu pour remplir le modèle.

Le code utilisant ces modèles aura besoin d'un champ `justCreated`, qui indique si le fichier de stockage des données vient d'être créé : quand l'application est lancée pour la première fois, le modèle est vide (les fichiers de stockage n'existent pas), c'est le moment pour charger des données d'exemple. Cette opération est effectuée seulement dans le cas où `justCreated` prend la valeur booléenne vraie (sinon, la base de données a déjà été créée).

La partie intéressante du modèle des genres, implémenté dans la classe `FileStringListModel`, est relativement courte. Chaque méthode de l'interface de modèle Qt Quick est implémentée (à l'exception de `set` : puisque les genres ne changent pas, la seule opération requise est l'ajout à la fin du modèle en chargeant les données initiales). Le code de l'opération `insert()` utilise l'API des modèles de PyQt, c'est-à-dire les fonctions `insertRows()` pour donner un nombre de lignes à ajouter dans le modèle et `setData()` pour modifier des valeurs. Les modifications sont directement reportées dans le fichier de stockage. La propriété `justCreated` indique si ce fichier existait à l'instanciation du modèle.

***Note** > Le code source de cet exemple et les fichiers d'éléments associés sont*

disponibles dans le projet `biblioapp-composant`  .

```
from PyQt5.QtCore import pyqtSlot, pyqtProperty, QAbstractListModel,
QModelIndex, Qt


class FileStringListModel(QAbstractListModel):
    def __init__(self, data, parent=None, *args):
        QAbstractListModel.__init__(self, parent, *args)
        self._data = data
        self._file = None
        self._just_created = False

    def _file_write(self):
        if self._file is None:
            return False

        with open(self._file, 'w') as f:
            json.dump(self._data, f)
        return True

    # …

    # API d'un modèle PyQt.
    def setData(self, index, value, role):
        if self._file is None or not index.isValid():
            return False

        self._data[index.row()] = value
        self.dataChanged.emit(index, index, [role])
        self._file_write()
        return True

    def insertRows(self, row, count, parent=None):
        if self._file is None:
            return False

        super(QAbstractListModel,
self).beginInsertRows(QModelIndex(), row, row + count - 1)
        for i in range(count):
            self._data.insert(row + i, None)
        super(QAbstractListModel, self).endInsertRows()
        self.countChanged.emit()
        return True

    # …

    # API d'un modèle Qt Quick.
    countChanged = pyqtSignal()
    @pyqtProperty(int, notify=countChanged)
    def count(self):
```

```
        return self.rowCount()

    @pyqtSlot(int, str, result=bool)
    def insert(self, row, value):
        return self.insertRows(row, 1) and \
               self.setData(self.createIndex(row, 0), value,
Qt.EditRole)

    @pyqtSlot(str, result=bool)
    def append(self, value):
        return self.insert(self.count, value)

    @pyqtSlot(int, result=str)
    def get(self, index):
        if self._file is None:
            return None
        return self._data[index]

    # Gestion d'un fichier.
    @pyqtProperty(str)
    def file(self):
        return self._file

    @file.setter
    def file(self, value):
        self._file = value
        if os.path.isfile(self._file) and os.path.getsize(self._file)
> 0:
            with open(self._file, 'r') as f:
                self._data = json.load(f)
        else:
            self._just_created = True
            self._data = []

    @pyqtProperty(bool)
    def justCreated(self):
        return self._just_created

    # Autre méthode utile sur le modèle :
    # retrouver l'indice d'un genre donné.
    @pyqtSlot(str, result=int)
    def findIndexByName(self, value):
        return [i for i in range(len(self._data)) if self._data[i] ==
value][0]
```

Avant d'utiliser ce modèle dans l'application, Qt Quick oblige à enregistrer ce nouveau type, c'est-à-dire indiquer qu'il existe un module `FileModel` (tout comme `QtQuick` ou `QtQuick.Controls`) qui fournit un type (nommé `FileStringListModel`) implémenté par une classe (ici, aussi `FileStringListModel`). Cette opération est effectuée par la fonction `qmlRegisterType(`*`ClassePython`*`, "`*`Module`*`", `*`VersionMajeure`*`,`

```
VersionMineure, "ComposantQtQuick"):

qmlRegisterType(FileStringListModel,
                "FileModel", 1, 0,
                "FileStringListModel")
```

Côté Qt Quick, on peut alors importer ce nouveau module et l'utiliser :

```
import FileModel 1.0

// …
genreModel: FileStringListModel {
    file: "genres.db"
    Component.onCompleted: {
        if (justCreated) {
            for (var i = 0; i < initialGenres.length; ++i) {
                genreModel.append(initialGenres[i]);
            }
        }
    }
}
```

***Note** > En définissant ainsi la version du composant Qt Quick directement au moment où il est enregistré, une bibliothèque externe peut proposer plusieurs versions de ses composants en appelant plusieurs fois `qmlRegisterType` pour le même nom de composant Qt Quick : les classes Python associées seront différentes, une classe par version exposée. Ainsi, une version très récente de la bibliothèque peut fournir une compatibilité parfaite avec ses versions antérieures : quand le développeur Qt Quick précise la version à importer du module, l'implémentation correspondante est choisie — sans risque de changement. Ainsi, si une version précédente de la bibliothèque présente un défaut, celui-ci sera toujours présent en important cette version antérieure, même s'il a été corrigé dans une version plus récente.*

Le cas du modèle des livres est plus compliqué. PyQt ne permet pas, à l'heure actuelle, d'échanger des dictionnaires avec le code Qt Quick (contrairement à Qt, en C++). Une solution partielle consiste à utiliser des rôles, comme dans un modèle Qt Quick : chaque rôle correspond à une information à mémoriser sur un livre (titre, auteur, genre, etc.). La fonction `roleNames` renvoie une correspondance entre les numéros et les noms des rôles disponibles dans le modèle (les identifiants sont utilisés comme paramètres par PyQt, tandis que les noms sont utilisés comme clés dans le dictionnaire représentant chaque livre) ; ensuite, `get` peut prendre un troisième argument, qui correspond au rôle (indiqué par son nom). Cette solution est partielle, car la fonction d'écriture `set` ne fonctionne pas avec des rôles (elle tente d'écrire, chaque fois, une entrée complète du modèle, c'est-à-dire un dictionnaire) : il est donc impossible de

mettre à jour un livre avec cette implémentation.

Note *> Les noms des rôles renvoyés par* `roleNames` *ne sont pas des chaînes de caractères, mais bien des chaînes d'octets. Le passage d'une représentation à l'autre se fait grâce à un encodage, comme UTF-8 :*

```
chaine = "str"
octets = b"str"

octets == chaine.encode('utf-8')
chaine == octest.decode('utf-8')
```

Également, les numéros des rôles ne peuvent commencer qu'à partir de `PyQt5.QtCore.Qt.UserRole` *: les numéros inférieurs sont réservés par PyQt.*

---

[24] Contrairement à un arbre, qui correspond à l'organisation des dossiers et des fichiers, par exemple : certains éléments sont les *enfants* d'autres. Pour ces cas, les modèles devraient suivre directement l'interface de QAbstractItemModel ; ensuite, ils peuvent être affichés avec TreeView.

[25] La documentation donne des indications sur les actions à réaliser pour cette implémentation ; un exemple est aussi montré dans la Section 2, Modèle.

4. Appel direct de code Python

Une approche alternative pour effectuer la communication entre Python et Qt Quick est d'appeler directement des fonctions Python depuis le code QML, sans vraiment chercher à comprendre le fonctionnement de PyQt. Par défaut, PyQt ne propose rien en ce sens ; ce manque est comblé par l'extension PyOtherSide. Quelques paquets précompilés existent déjà pour Linux, mais il faut le compiler pour les autres plateformes.

La communication se fait à l'aide d'un composant Python :

```
import io.thp.pyotherside 1.5

Python {
    id: python
    Component.onCompleted: {
        // …
    }
}
```

Les opérations sont asynchrones : l'opération à effectuer est transférée à Python et, du côté Qt Quick, la fonction JavaScript correspondante renvoie directement sa valeur. C'est pour ça que toutes les opérations PyOtherSide prennent, en dernier argument, une fonction anonyme : dès que l'opération demandée est effectuée, cette fonction est appelée.

```
import io.thp.pyotherside 1.5

Python {
    id: python
    property bool ready: false
```

❶ Component.onCompleted: { importModule("json", function() { ❷ ready = true ❸ }) } }

❶ Cette propriété prend la valeur booléenne vraie dès que le composant peut être utilisé.

❷ La première opération possible est d'importer un module Python, ici `json`.

❸ Une fois le module chargé, cette fonction est appelée : elle change la valeur de la propriété `ready`.

Ensuite, les appels au code Python se font à l'aide de la fonction `call()`, qui prend trois arguments : le nom de la fonction à appeler (y compris le module, comme `json.load`), les arguments (un tableau, possiblement vide), puis finalement une fonction de retour (dont l'argument est la valeur retournée par l'appel de fonction Python).

```
call("json.load", ["books.db"], function (books) {
    // …
})
```

20

Traduire l'application

Niveau : intermédiaire

Objectifs : traduire une application en fonction de la langue de l'utilisateur

Prérequis : Premiers pas avec Qt Quick, Communiquer avec Python

Qt Quick propose un système d'internationalisation très similaire à celui de PyQt, où les rôles du programmeur et du traducteur sont entièrement découplés. Le programmeur ajoute des marques spécifiques dans son code Qt Quick (avec la fonction `qsTr`), qui sont repérées et collationnées par un outil spécifique (`lupdate`). Le résultat de cette deuxième étape sert de lien entre le programmeur et le traducteur : les fichiers produits par `lupdate` sont directement utilisés par le traducteur pour son travail, puis utilisés dans l'application (après un passage par `lrelease`).

À l'exécution, l'application regarde la langue de l'utilisateur, charge le fichier de traduction correspondant ; de son côté, le moteur d'exécution Qt Quick, par le biais de la fonction `qsTr`, effectuera le remplacement avec la chaîne de caractères qui correspond à la langue choisie à l'exécution pour chaque chaîne marquée.

Du côté du programmeur, la principale difficulté pour traduire l'application se limite à ajouter les appels à `qsTr` aux bons endroits — ni plus ni moins. Cette opération est néanmoins nécessaire : seul le programmeur sait réellement quelles chaînes doivent être traduites et lesquelles peuvent être ignorées (par exemple, les clés dans un dictionnaire, les URL ou toute occurrence de `""`).

***Note** > PyQt fournit tant `pylupdate5` que `lupdate`. Le premier ne peut traiter que des fichiers Python, tandis que le second fonctionne pour du code source C++ et QML. L'inconvénient est que les deux ne peuvent pas fusionner leurs fichiers : deux fichiers de traduction sont nécessaires pour traduire tant le code Python que Qt Quick.*

1. Traduction simple

Quand PyQt utilise la fonction `tr` pour marquer les chaînes de caractères, Qt Quick utilise `qsTr`. Ainsi, le code ci-après permet de traduire le texte du bouton :

```
Button {
    text: qsTr("Valider")
}
```

Le développeur peut ensuite insérer, directement dans son code, des commentaires précisant le contexte de chaque chaîne de caractères. Ces notes spéciales indiquent au traducteur dans quelle partie de l'application l'élément textuel se trouve et aident à lever d'éventuelles ambiguïtés. PyQt reconnaît deux types de tels commentaires :

- ceux qui commencent par `//`: sont détectés comme le contexte pour la traduction suivante (il ne peut y en avoir qu'un seul par chaîne à traduire) ;

- ceux qui commencent par `//~` servent à donner des notes supplémentaires. Le premier mot après `//~` est automatiquement considéré comme un identifiant, tandis que le reste de la ligne constitue le commentaire proprement dit.

Par exemple, "Valider" peut être compris de différentes manières : ce bouton pourrait soumettre un formulaire ("Envoyer") ou bien approuver une décision affichée à l'écran ("Confirmer").

```
Button {
    //: La modification sur l'utilisateur courant a été revérifiée
par l'utilisateur.
    //~ Contexte Aucun rapport avec une soumission de formulaire.
    text: qsTr("Valider")
}
```

***Note** > Pour les chaînes de caractères affichées gérées par Qt Quick (comme les boutons OK, Annuler, etc. des boîtes de dialogue de confirmation), la traduction se fait automatiquement : PyQt est livré avec une série de fichiers de traduction pour toutes ces chaînes de base, avec une adaptation à la plateforme courante (les noms de ces boutons correspondront à ceux habituellement utilisés pour la plateforme de l'utilisateur et sa langue).*

***Note** > En écrivant le code source en français, on risque de se retrouver très vite face à des problèmes d'encodage (certains caractères peuvent provoquer des*

erreurs). La seule manière entièrement fiable de procéder est de n'utiliser que les caractères ASCII ; de manière simple, cela signifie d'écrire toutes les chaînes de caractères en anglais dans le code source. Pour le français, omettez les accents, les cédilles, le symbole euro, etc. : écrivez `Details` *au lieu de* `Détails`*, en remplaçant le* `é` *par sa version non accentuée* `e`*; par la suite, il faudra effectuer une traduction du "français ASCII" vers du bon français, en remettant les accents et autres caractères spéciaux à leur place.*

De plus, la traduction est très souvent effectuée par des sociétés ou des contributeurs externes (dans le cas d'un projet libre) : recruter des traducteurs de l'anglais vers une autre langue est nettement plus facile que depuis le français. Dans ce cas, le meilleur choix est d'écrire toutes les chaînes à traduire ainsi que les commentaires de contexte en anglais.

1.1. Extraction des chaînes

Une fois le code ainsi modifié, l'étape suivante consiste à extraire les chaînes dans des fichiers à distribuer au traducteur. La procédure imposée est d'abord d'écrire un fichier de projet (dont la fonction principale est de lister tous les fichiers à analyser), puis de lancer `lupdate` sur ce fichier de projet.

Deux variables doivent être définies dans le fichier de projet : `SOURCES` liste les fichiers sources (seuls ceux qui doivent être traduits s'y retrouvent obligatoirement, mais ajouter d'autres fichiers sources ne cause pas de tort), tandis que `TRANSLATIONS` donne tous les fichiers de traduction à générer (`.ts`). Le code langue respecte un certain format : la première partie donne la langue proprement dite (`fr`, `en`), la seconde la région (`FR` pour France, `US` pour États-Unis).

```
SOURCES += *.qml

TRANSLATIONS += biblioapp.fr_FR.ts \
                biblioapp.en_US.ts
```

L'extraction proprement dite se fait avec l'utilitaire `lupdate` en ligne de commande ou depuis l'environnement de développement, comme dans le cas PyQt.

```
lupdate biblioapp.pro
```

***Note** > Les fichiers ainsi générés ne sont différents que par leur nom :* `lupdate` *ne cherche pas à comprendre les codes de langue et de pays. Qt Linguist demandera la langue de la traduction lors de la première ouverture de ces fichiers.*

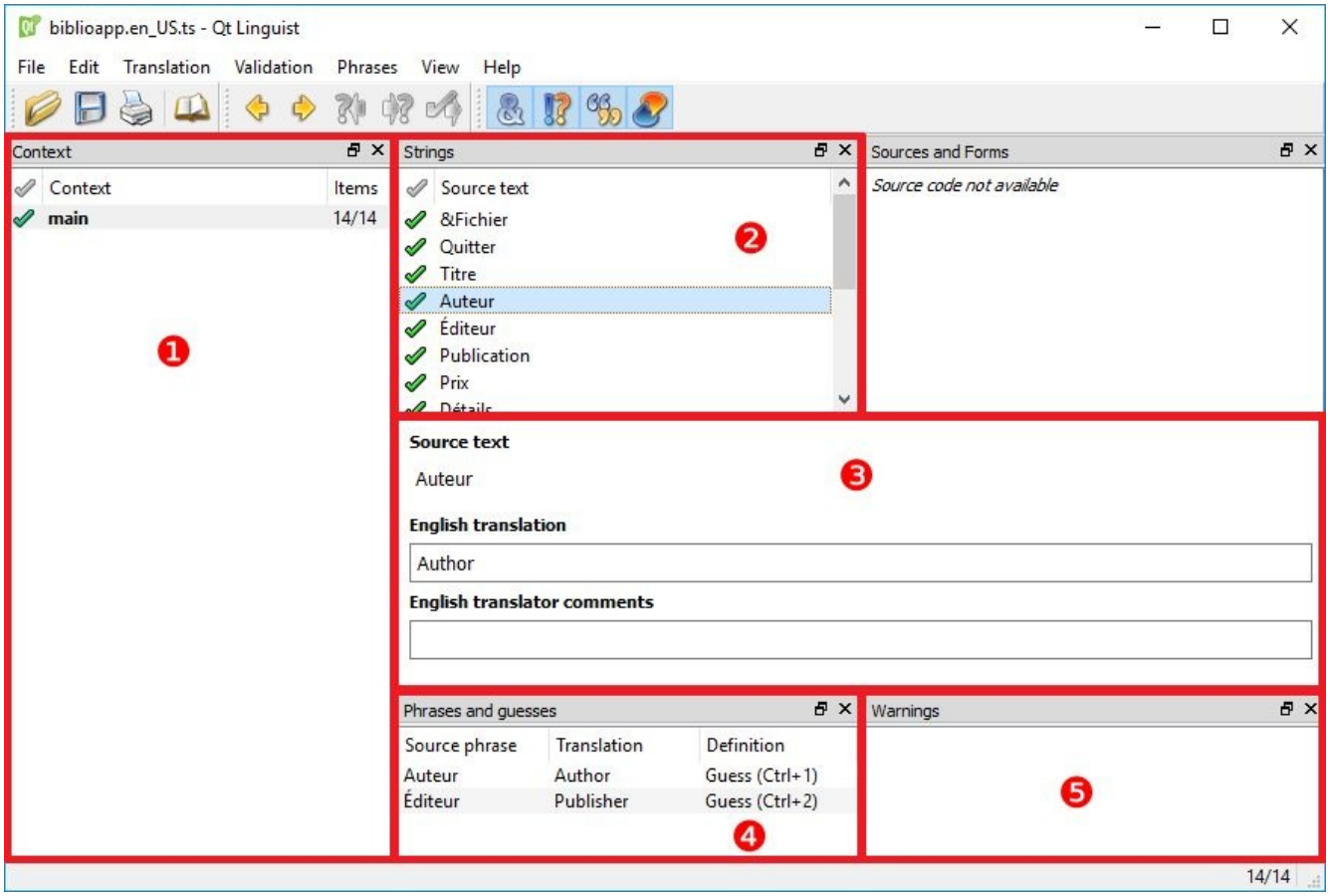
Par la suite, lors de l'évolution de l'application, la seule modification à apporter est la mise à jour du fichier de projet pour lister les nouveaux fichiers sources (et, éventuellement, les nouvelles langues). Ensuite, `lupdate` adaptera les fichiers de traduction pour ajouter les nouvelles chaînes de caractères à traduire ; toutes les traductions déjà effectuées sont gardées.

***Note** > Puisque `lupdate` ne fonctionne pas avec du code Python, dans une application mixte, deux fichiers de projet sont requis : l'un pour le code Python, l'autre pour le code Qt Quick.*

1.2. Qt Linguist

L'étape suivante consiste à la traduction proprement dite. Elle pourra être réalisée à l'aide de Qt Linguist de la même manière qu'avec une application conçue à base de widgets.

***Figure 20.1 :** Qt Linguist*



Voici un survol rapide des différentes vues de Qt Linguist avec un projet Qt Quick :

❶ Contexte : il s'agit des différents composants Qt Quick d'où sont extraits les éléments textuels à traduire. De par la structure de l'application traduite (tous les éléments dans un fichier `main.qml`), seul le module `main` est proposé ; pour une application plus complexe, chaque fichier QML dont des chaînes ont été extraites sera présent dans cette liste. Ces modules servent à donner un contexte d'utilisation au traducteur.

Note *> Chaque contexte indique un petit compteur de l'avancement de la traduction. 14/14 signifie qu'on a traduit et validé 14 des 14 éléments à traduire dans le contexte.*

❷ Chaînes : liste dans la langue source tous les éléments textuels du contexte sélectionné. Une icône indique l'état de la traduction : un point d'interrogation signale que l'élément textuel n'est pas traduit, une coche verte indique qu'il est traduit. Ces icônes de statut sont indicatives, elles peuvent être changées en cliquant dessus.

❸ Zone de saisie de la traduction de l'élément textuel sélectionné dans la vue Chaînes. Il ne s'agit pas d'une vue au sens strict : c'est la zone centrale de l'application qu'on ne peut pas faire disparaître ; les vues viennent se placer autour.

❹ Phrases et propositions : liste des traductions usuelles sauvegardées par l'utilisateur. Ces traductions sont sauvegardées dans un fichier "livre de phrases" avec l'extension `.qph`. Ces fichiers sont gérés au niveau de l'élément Phrases du menu principal. Cette fonction est surtout utile pour les grosses applications.

❺ Avertissements : liste des anomalies potentielles détectées dans les traductions ; il s'agit de différences par rapport au livre de phrases ou d'incohérences entre les éléments textuels source et cible : esperluette manquante (accélérateur), ponctuation différente en fin d'élément ou différence dans les arguments (`%1`, `%2`, etc. : voir la Section 2, Arguments). Ces validations peuvent être désactivées ou réactivées à tout moment via le menu Validation ou les boutons Basculer… sur la barre d'outils.

1.3. Adaptation de l'application

Note *> Le code source final de cet exemple et les fichiers d'éléments associés sont disponibles dans le projet* `traduction`  .

Une fois les fichiers `.ts` traduits, `lrelease` les transforme en une représentation binaire facile à charger à l'exécution.

```
lrelease biblioapp.en_US.ts
lrelease biblioapp.fr_FR.ts
```

Ensuite, dans le code Python qui charge l'interface Qt Quick, trois lignes suffisent pour charger la traduction : créer un objet QTranslator (qui se chargera d'effectuer la traduction), lui donner le fichier de traduction voulu, puis indiquer à QApplication de l'utiliser pour la traduction.

```
app = QApplication(sys.argv)

translator = QTranslator()
translator.load(QLocale(), 'biblioapp', '.')
app.installTranslator(translator)
```

La syntaxe utilisée choisit automatiquement le fichier de traduction selon la langue configurée par l'utilisateur. `translator.load` peut aussi prendre en argument le nom du fichier à charger (par exemple, lors de tests, pour forcer la langue de l'interface facilement).

Note *> Pour charger un autre fichier de traduction, la même procédure s'applique : un objet QTranslator est requis pour chaque fichier chargé. En cas de conflit pour la traduction d'une chaîne donnée (même source, même contexte), le dernier QTranslator installé aura la priorité.*

Attention *> Dans une application bien codée, avec des positionneurs, les différents composants s'adapteront automatiquement à la taille des traductions. Cependant, si les chaînes traduites sont beaucoup plus longues, d'autres adaptations seront peut-être requises. À ce niveau, il est souvent utile de faire des tests en français et en anglais : dans cette dernière langue, les chaînes seront très souvent assez courtes, mais bien plus longues en français ; bon nombre de langues se situent, en moyenne, entre ces deux extrêmes — même si des chaînes spécifiques peuvent s'en écarter franchement.*

2. Arguments

Dans une application plus compliquée que BiblioApp, toutes les chaînes de caractères affichées ne sont pas toujours constantes. Par exemple, dans un éditeur de texte, la fonction de recherche permet souvent de compter les occurrences, avec une réponse de la forme *42 occurrences trouvées*. Si l'application ne gère pas le pluriel, il n'est pas rare d'afficher un *42 occurrence(s) trouvée(s)*, ce qui n'est guère esthétique ou sérieux.

Une solution simple serait d'ajouter une condition dans le code : si le nombre d'occurrences est d'une unité, alors pas de `s` ; sinon, mettre un `s`. Cependant, ces tests simples échouent déjà pour de l'anglais : le pluriel est de mise pour zéro unité, contrairement au français. La situation est encore pire en polonais, qui possède trois nombres (le singulier, le paucal et le pluriel).

Mieux vaut donc utiliser la solution proposée dans l'environnement Qt Quick, qui gère tous ces cas automatiquement et permet au traducteur d'accorder les mots à ce nombre (mais aussi de les réordonner, si nécessaire).

Lors de l'appel à `qsTr`, tous les paramètres à remplacer sont représentés par un symbole pourcent `%` suivi d'un chiffre ; ensuite, les valeurs sont indiquées par des appels à `arg` (un par argument). Par exemple, pour le nombre d'occurrences :

```
Text {
    text: qsTr("%1 occurrence(s) trouvée(s)").arg(nOccurrences)
}
```

Quand il y a plusieurs arguments à remplacer, les appels à `arg` peuvent s'effectuer en cascade :

```
Text {
    text: qsTr("Traitement en cours : fichier %1 sur
%2").arg(i).arg(n)
}
```

Cette structure de base peut s'adapter à d'autres cas. Notamment, l'affichage des nombres varie d'une langue à l'autre : le français utilise l'espace comme séparateur des milliers et la virgule pour les décimales `12 345,67`, l'anglais des virgules et des points `12,345.67`, l'allemand l'inverse `12.345,67`. Utiliser `%L1` à la place de `%1` demande à PyQt de gérer ces diversités, en étant cohérent avec la langue de l'utilisateur.

```
Text {
```

```
    text: qsTr("Valeur : %L1").arg(v)
}
```

Pour les dates, par contre, l'opération doit être faite à la main, dans l'argument passé à `arg`.

```
Text {
    text: qsTr("%1").arg(Date().toLocaleString(Qt.locale()))
}
```

Note *> Ce point s'oppose complètement à ce qui est fait avec PyQt : ici, utiliser les mécanismes fournis par Qt Quick est obligatoire. En effet, les capacités de JavaScript sont, pour l'instant, relativement limitées en ce qui concerne l'interpolation de chaînes de caractères.*

3. Adaptations profondes

Dans certains cas, les modifications à apporter à l'interface sont plus profondes que les textes à afficher à l'utilisateur. Par exemple, il peut s'agir d'agrandir une zone de l'interface pour laisser place à la traduction… ou bien d'afficher une autre icône, de jouer un autre son selon la langue. Dans ces cas, il devient utile de faire des tests selon la langue de l'utilisateur.

Par exemple, ce code change une icône selon le code de langue [les deux premiers caractères de `Qt.locale().name`] :

```
Component.onCompleted: {
    switch (Qt.locale().name.substring(0,2)) {
        case "en":
            languageIcon = "images/icon_en.png";
            break;
        case "fr":
            languageIcon = "images/icon_fi.png";
            break;
        default:
            languageIcon = "images/icon_default.png";
    }
}
```

Comment vérifier que toutes les chaînes ont été traduites ?

Pour vérifier qu'au moins toutes les chaînes nécessaires sont encadrées par un appel à `qsTr`, une technique est de traduire (avec un script) toutes les chaînes du fichier `.ts`, en reprenant les chaînes dans la langue d'origine du code et en les altérant de telle sorte qu'on reconnaisse facilement l'opération (par exemple, en remplaçant toutes les voyelles par d'autres symboles).

Ainsi, en lançant l'application avec cette traduction fantaisiste, on verra rapidement quelles chaînes n'ont pas été traduites : elles ne présentent pas ces altérations. En effet, si un appel à `qsTr` a été oublié, la chaîne correspondante ne sera pas touchée : l'application affichera la chaîne d'origine.

Qt Quick avancé

21

Interactivité avancée

Niveau : intermédiaire

Objectifs : exploiter des mécanismes plus avancés d'interactivité avec l'utilisateur (glisser-déposer, clavier, barres de défilement)

Prérequis : Premiers pas avec Qt Quick, Présentation de JavaScript

Jusqu'à présent, les mécanismes présentés pour l'interactivité avec l'utilisateur concernaient principalement la souris. Cependant, Qt Quick permet bien plus, notamment pour l'interaction avec le clavier (récupérer des appuis de touche et des combinaisons), le glisser-déposer à l'intérieur d'une application et les barres de défilement (pour du contenu trop grand par rapport à l'espace disponible pour l'afficher).

Note *> Le code des exemples de ce chapitre est disponible dans le projet* `interactivite` .

1. Interactions par le clavier

Le type attaché `Keys` permet de réagir aux événements du clavier. Il définit toute une série de gestionnaires de signal selon la touche appuyée. Par exemple, `Keys.onDigit0Pressed` détecte si l'utilisateur appuie sur la touche 0, `Keys.onEnterPressed` si c'est un retour à la ligne, `Keys.onRightPressed` si c'est la flèche directionnelle vers la droite, etc.

```
Item {
    focus: true
    Keys.onDigit0Pressed: console.log("zéro")
    Keys.onRightPressed: console.log("droite")
}
```

Toutes les touches ne sont pas accessibles par ce mécanisme (la liste est donnée dans la documentation). Pour celles qui ne le sont pas, le gestionnaire de signal à utiliser est `Keys.onPressed` : de manière très générique, il est appelé dès que l'utilisateur appuie sur une touche. La variable `event` contient une instance de KeyEvent, qui donne notamment la touche appuyée par `event.key`. La liste des codes disponibles est décrite dans la documentation.

```
Item {
    focus: true
    Keys.onPressed: {
        if (event.key == Qt.Key_0) {
            console.log("zéro")
            event.accepted = true
        } else if (event.key == Qt.Key_Left) {
            console.log("droite")
            event.accepted = true
        }
    }
}
```

***Note** > Le champ `event.accepted` indique que l'événement correspondant est traité et ne doit pas être transmis au reste de l'application. Si cette ligne est omise, l'événement continue d'être traité par tous les éléments et gestionnaires d'événements concernés : par exemple, si un composant utilise le gestionnaire de signal générique `Keys.onPressed` ainsi qu'une version spécifique (comme `Keys.onEnterPressed`) pour un même événement du clavier, les deux seront exécutés (dans un ordre a priori non défini).*

Les modificateurs utilisés (Ctrl, Alt, Maj) sont aussi fournis à travers `event` dans la

propriété `modifiers`. Puisque plusieurs touches de modification peuvent être utilisées simultanément, une comparaison simple comme pour `event.key` ne fonctionne pas : ce champ est une composition de tous les modificateurs possibles. Les tests utilisent donc des opérateurs logiques entre la valeur de `event.modifiers` et les codes des touches à vérifier :

```
Item {
    focus: true
    Keys.onPressed: {
        if ((event.key == Qt.Key_0)
              && (event.modifiers & Qt.AltModifier)
              && (event.modifiers & Qt.ControlModifier)) {
            console.log("Alt Ctrl zéro")
            event.accepted = true;
        }
    }
}
```

***Note** > Qt Quick émet aussi des signaux quand les touches sont lâchées, ils sont accessibles par `Keys.onReleased`. Il n'existe aucun raccourci dans ce cas (par exemple, aucun `Keys.onRightReleased`) : tous les cas doivent être gérés dans `Keys.onReleased`.*

1.1. Vie d'un événement

Ces trois exemples ont tous la ligne `focus: true`. Son utilité est de gérer la cible de saisie : elle indique que les événements en provenance du clavier pourraient être traités par le composant concerné (ils seront dans tous les cas transmis au composant, qui pourra décider de les traiter ou non). Le traitement de ces événements est quelque peu complexe :

1. L'utilisateur appuie sur une touche du **clavier**, qu'il soit physique ou virtuel (par exemple, sur un téléphone portable ou un périphérique embarqué : le clavier est affiché sur un écran tactile). Le système d'exploitation transmet deux types d'information à l'application : l'appui et le relâchement d'une touche.

2. L'**application PyQt** reçoit cet événement et le traite au sein de sa boucle d'événements interne (dans QApplication) : le widget qui doit recevoir l'événement (celui qui est actuellement actif) est identifié et l'information lui est passée.

3. Pour une application Qt Quick, le **gestionnaire de scène Qt Quick** reçoit cet événement et le propage dans la hiérarchie des composants, où l'un d'entre eux

pourrait traiter l'événement (c'est-à-dire exécuter l'instruction `event.accepted = true`).

Dans la scène Qt Quick, l'événement continue de poursuivre son chemin :

- la cible de saisie (marquée par `focus: true`) reçoit en priorité l'événement : si elle l'accepte (`event.accepted = true`), l'événement est considéré comme traité ;
- sinon, l'événement est transmis à l'élément parent. À nouveau, ce dernier peut l'accepter (`event.accepted = true`) ou non ;
- l'élément continue son trajet jusqu'au composant racine, d'enfant en parent, s'il n'est pas traité dans la hiérarchie. À ce niveau, l'événement est ignoré.

1.2. Gestion de la cible de saisie

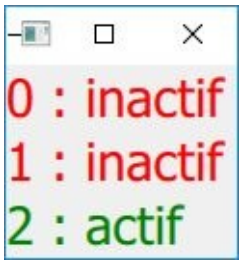
En résumé, la ligne `focus: true` a simplement pour objectif de demander à récupérer les événements du clavier. Elle permet de le faire dès le lancement de l'application. Évidemment, un seul composant peut être la cible de saisie : si plusieurs composants précisent la ligne `focus: true`, seul le dernier sera actif.

La partie intéressante concerne la gestion de cette cible de saisie : elle peut bien évidemment changer au cours de l'application. Par exemple, quand l'utilisateur a terminé de remplir un champ du formulaire, il aimerait passer au suivant, ce qui implique un changement de la cible. Une manière très simple de la déplacer est de changer la valeur de la propriété `focus` du composant qui doit la recevoir, comme dans un gestionnaire de signal. L'exemple suivant montre une série de zones sensibles au clic avec du texte (instanciées par un Repeater) : dès que l'une reçoit un clic, le composant Text correspondant devient la cible de saisie et reçoit les événements du clavier, qu'il transmet dans la console.

```
Repeater {
    model: 3

    Text {
        text: modelData + " : " + (activeFocus ? "actif" : "inactif")
        color: activeFocus ? "green" : "red"
        font.pointSize: 20

        Keys.onPressed: {
            console.log("[" + modelData + "] Touche : " + event.key)
```

```
            event.accepted = true
        }

        MouseArea {
            anchors.fill: parent
            onClicked: parent.focus = true
        }
    }
}
```

Figure 21.1 : *Changer la cible de saisie dynamiquement, ici dans un gestionnaire de signal*



Cependant, il faut aussi prendre en compte les habitudes des utilisateurs : par exemple, dans un formulaire, la touche Tabulation est généralement utilisée pour passer la saisie au champ suivant (Maj+Tabulation servant à revenir en arrière). La propriété `activeFocusOnTab` sert justement à cela : chaque composant qui la définit à une valeur booléenne vraie est ajouté dans une chaîne d'éléments ; dès que l'utilisateur appuie sur la touche Tabulation, l'élément suivant de la chaîne est sélectionné.

Attention *> Bien évidemment, les appuis sur la touche Tabulation ne doivent pas être traités par le composant lui-même pour que le mécanisme fonctionne : sinon, ils y sont absorbés et le moteur d'exécution Qt Quick ne peut plus avancer dans la chaîne.*

```
Text {
    activeFocusOnTab: true
    // …

    Keys.onPressed: {
        if (event.key !== Qt.Key_Tab) {
            console.log("[" + modelData + "] Touche : " + event.key)
            event.accepted = true
        }
    }
}
```

Note *> Pour réaliser des chaînes d'activation de la cible de saisie plus complexes, par exemple en utilisant d'autres touches (comme les flèches directionnelles), le seul choix est d'utiliser le type attaché KeyNavigation.*

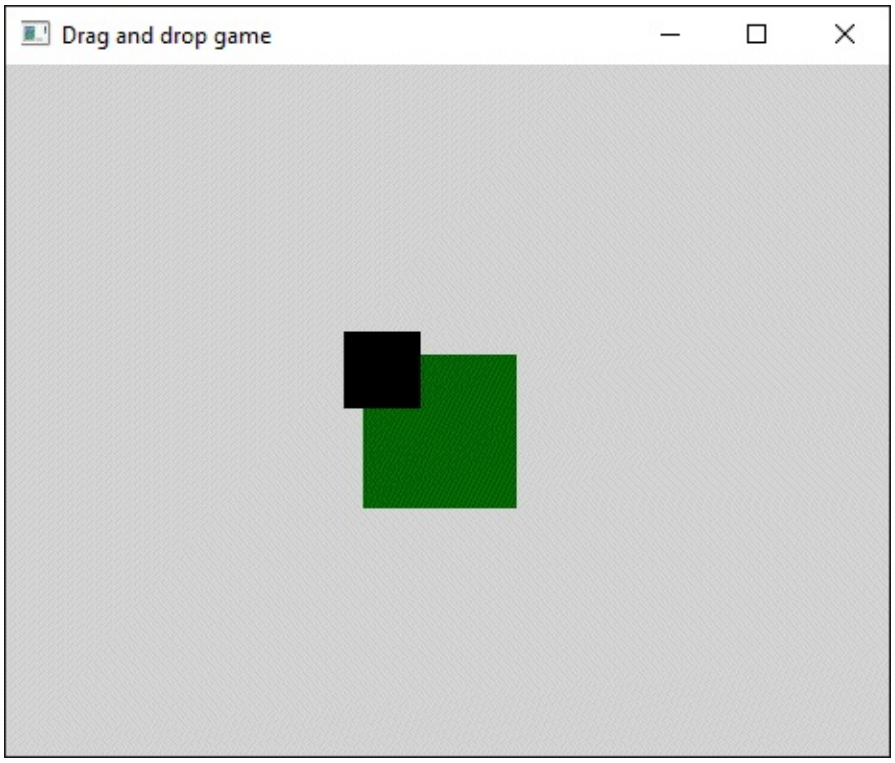
2. Glisser-déposer

Un des avantages de Qt Quick est qu'il est simple d'implémenter des comportements relativement complexes, comme le glisser-déposer. Il s'agit d'exploiter deux élements :

- du côté des éléments qui peuvent être déplacés, les propriétés attachées `drag` (toutes les propriétés correspondantes sont préfixées par `drag.`) ;
- pour recevoir ces éléments glissés, DropArea (dont l'utilisation est très similaire aux MouseArea déjà rencontrés).

Pour illustrer le fonctionnement de cette partie de Qt Quick, nous allons développer un petit jeu : trouver une zone de l'interface en déplaçant un rectangle ; cette zone devient verte une fois trouvée. De là, si vous êtes intéressé, vous pourriez créer un jeu de cartes de type solitaire, qui propose à l'utilisateur de jouer ou de déplacer une carte par du glisser-déposer.

Figure 21.2 : *Petit jeu : trouver une zone*




```
ApplicationWindow {
    // …
```

```
    Rectangle {
        id: rectangle
        x: 40 + Math.random() * (parent.width - 2 * 40)
        y: 40 + Math.random() * (parent.height - 2 * 40)
        width: 40; height: 40
        color: (drop.containsDrag) ? "darkgreen" : "lightgrey"
```

❶ DropArea { ❷ id: drop anchors.fill: parent } } Rectangle { id: drag x: 20; y: 20 width: 20; height: 20 color: "black" Drag.active: true ❸ Drag.hotSpot: { x: 10; y: 10 } ❹ MouseArea { id: dragArea anchors.fill: parent drag.target: parent ❺ } } }

La première chose à faire pour activer le glisser-déposer d'un élément est de le marquer comme glissable. Pour ce faire, il faut que le composant déplaçable réagisse aux événements de souris (d'où le MouseArea), mais aussi que le déplacement concerne un élément graphique, indiqué par la propriété `drag.target`.
❺ Toute la représentation graphique de cet élément sera déplacée avec la souris.

***Note** > De manière générale, ce composant MouseArea gère tous les aspects du déplacement : les propriétés `drag.` indiquent s'il faut limiter la zone accessible ou s'il faut gérer le comportement par rapport à d'autres composants pour le déplacement, par exemple.*

Ensuite, pour que le glissement soit effectif et que d'autres parties du code puissent
❸ gérer le dépôt, il faut activer l'émission d'événements avec la propriété attachée `Drag.active` directement au niveau de l'élément déplaçable.

Même si les deux étapes précédentes sont suffisantes, pour améliorer l'expérience utilisateur, il est intéressant de modifier le point où Qt Quick décide qu'il y a une collision : dès que le point indiqué par la propriété attachée `Drag.hotSpot` touche
❹ une zone DropArea, il envoie des événements de glisser-déposer. Par défaut, il s'agit du coin en haut à gauche, ce qui est un comportement contre-intuitif pour l'utilisateur (il faudrait que le coin en haut à gauche de `drag` touche la zone de `rectangle` pour qu'il s'allume).

Côté récepteur, l'action principale pour recevoir des éléments est de définir un élément DropArea, de la même manière qu'un élément MouseArea est utilisé pour la
❷ souris. Ce composant définit bon nombre de propriétés et de gestionnaires de signal utiles pour la configuration et l'utilisation à l'exécution. Parmi elles, `containsDrag` indique s'il y a (ou non) un événement de dépôt.

Finalement, le rectangle doit changer de couleur quand l'utilisateur arrive au-dessus
❶ de lui : pour ce faire, la propriété `color` prend une valeur conditionnelle par rapport à `drop.containsDrag`.

***Note** > Il pourrait être plus naturel d'écrire l'exemple précédent en cachant le rectangle vert et en ne l'activant que lorsqu'il est survolé, par exemple avec une ligne comme :*

```
visible: drop.containsDrag
```

Cependant, dans ce cas, le rectangle commence invisible, y compris le composant DragArea : ce dernier ne pourra donc recevoir aucun événement, l'exemple ne fonctionnera donc pas.

Ce petit exemple ne montre qu'une petite partie des fonctionnalités du glisser-déposer. Il est notamment possible d'activer le glisser-déposer avec certaines touches du clavier, d'indiquer des actions préférées lors du dépôt (copie ou déplacement, principalement), d'accéder à l'objet déposé, etc., en utilisant d'autres propriétés et fonctions de DropArea.

3. Barres de défilement

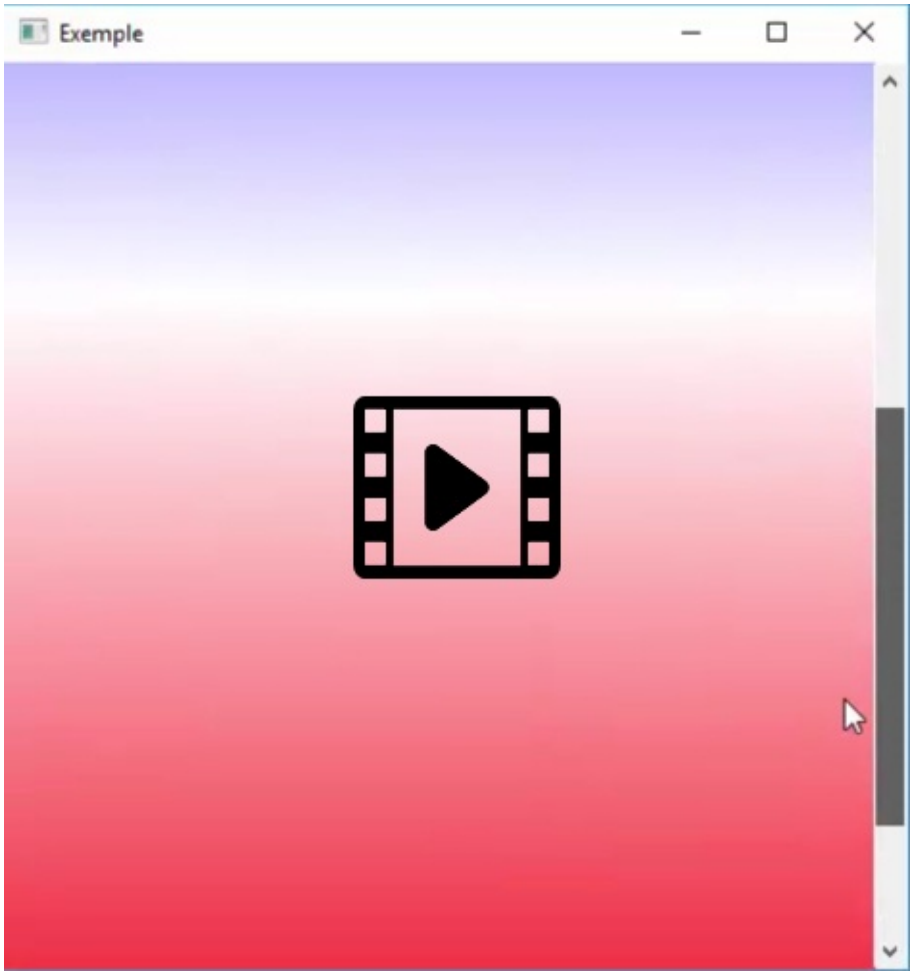
Lorsque du contenu est trop grand pour une fenêtre, toutes les plateformes implémentent un mécanisme ou l'autre pour laisser l'utilisateur choisir la partie à afficher : avec un clavier et une souris, des barres de défilement (ou ascenseurs) ; quand seul un écran tactile est disponible, des gestes.

Avec Qt Quick, le mécanisme des barres de défilement est déjà implémenté par le composant ScrollView. Son utilisation est redoutablement simple : il suffit de donner la taille souhaitée à la ScrollView, puis d'y insérer le contenu.

À titre d'exemple, on peut générer un dégradé de couleurs (du bleu au rouge en passant par le blanc) deux fois plus grand que la fenêtre : initialement, seule la transition du bleu au blanc est visible ; en faisant défiler, on commence à voir du rouge.

```
ScrollView {
    anchors.fill: parent

    Rectangle {
        width: window.width - 17
        height: 2 * window.height

        gradient: Gradient {
            GradientStop { position: 0.0; color: "blue" }
            GradientStop { position: 0.5; color: "white" }
            GradientStop { position: 1.0; color: "red" }
        }
    }
}
```

Figure 21.3 : *Utilisation des barres de défilement à travers un dégradé (vidéo)*



Pour des besoins plus particuliers (notamment sur des écrans tactiles), Flickable est plus souple d'emploi et n'impose pas l'utilisation des barres de défilement. Cependant, il est de bien plus bas niveau, il faut par exemple implémenter soi-même ces barres de défilement.

Et avec Qt Quick Controls 2 ?

Le mécanisme du défilement a été complètement repensé avec les Qt Quick Controls 2 : exit ScrollView, toutes les fonctionnalités se rapportent à Flickable. Les barres de défilement peuvent alors être ajoutées à la main, avec les propriétés attachées et le composant ScrollBar.

```
Flickable {
    Rectangle { … }
    ScrollBar.vertical: ScrollBar {}
}
```

22

États et transitions

Niveau : débutant à intermédiaire

Objectifs : structurer son interface à l'aide de machines d'états et rendre les animations plus fluides avec les transitions

Prérequis : Premiers pas avec Qt Quick, Présentation de JavaScript (Utiliser la méthodologie modèle - vue pour la section Section 4, Transitions et vues)

Bon nombre d'éléments d'une interface peuvent être décrits à l'aide d'*états* : par exemple, un bouton est enfoncé ou ne l'est pas. Qt Quick intègre cette notion, ce qui peut simplifier les réactions aux interactions de l'utilisateur. De plus, ces états sont un marche-pied vers des transitions, qui donnent une apparence plus finie à l'application : le passage d'un état à l'autre se fait progressivement avec des animations.

Dans le cas d'un bouton, l'état *enfoncé* correspondrait à une couleur plus sombre, créant une impression de profondeur ; au contraire, l'état *relâché* correspondrait à une couleur plus claire et l'impression que le bouton dépasse. Ces changements sont purement esthétiques et peuvent être obtenus en modifiant les propriétés de certains composants utilisés pour afficher le bouton. Les états de Qt Quick servent exactement à cela : définir un ensemble de valeurs pour des propriétés. Lors d'un passage à un nouvel état, toutes ces propriétés sont modifiées.

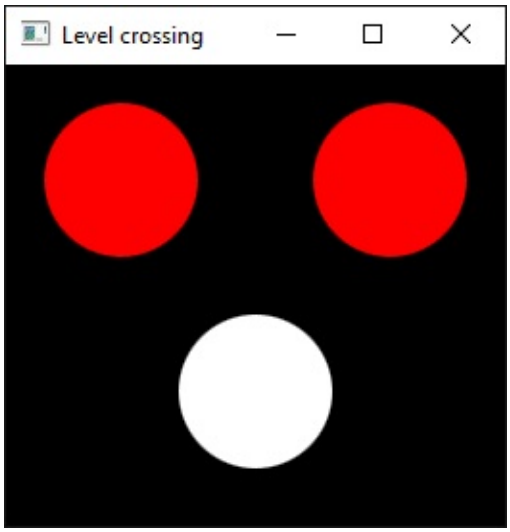
Cette notion d'état est, dans une interface graphique, rapidement liée à celle de transition : si le bouton passe d'un état *enfoncé* à un état *relâché* de manière abrupte, l'effet sera grossier et peu satisfaisant de nos jours (même si c'était encore la norme en 2010, avec la puissance de calcul limitée des ordinateurs de l'époque). Au contraire, s'il se fait avec une petite animation qui montre le bouton s'affaisser, l'interface paraîtra plus aboutie. De même, lors de modifications dans un modèle (ajout d'un livre pour BiblioApp, par exemple), l'élément peut apparaître de manière douce à l'utilisateur : l'image du livre qui passe de l'état transparent à opaque, notamment.

Figure 22.1 : *Exemple d'animation lors de l'activation d'un bouton dans la barre des tâches de Windows 10 (vidéo)*

1. Définition d'un état

Les feux de signalisation représentent un exemple typique d'application des états. Nous nous inspirerons ici des feux réglementant les passages à niveau en Belgique : un feu blanc clignote tant que les voitures peuvent passer, deux feux rouges clignotent alternativement quand un train est à l'approche. L'objectif sera d'afficher les trois parties du feu selon que les voitures peuvent passer ou non ; le passage d'un mode à l'autre se fera par un clic sur l'interface.

Figure 22.2 : *Signalisation d'un passage à niveau*



L'implémentation est relativement simple, en positionnant trois cercles. Cependant, Qt Quick ne dispose pas d'un composant de cercle : l'astuce d'implémentation est d'utiliser des rectangles avec des bords arrondis grâce à la propriété `radius` — suffisamment arrondis, ils finissent en cercles. Autre point nouveau : pour définir l'arrière-plan, nous utiliserons le composant ApplicationWindowStyle (comme son nom l'indique, il définit le style d'un composant ApplicationWindow).

Note *> Le code source de cet exemple et des suivants est disponible dans le projet `etats`. Ce premier exemple est dans le dossier `feu-base`* .

```
import QtQuick 2.5
import QtQuick.Controls 1.4
import QtQuick.Controls.Styles 1.4

ApplicationWindow {
```

```
    visible: true
    width: 260; height: 240
    title: qsTr("Level crossing")
    style: ApplicationWindowStyle {
            background: Rectangle {
                color: "black"
            }
        }

    Rectangle {
        id: red1
        width: 80; height: 80; radius: 40
        color: "red"
        x: 20; y: 20
    }

    Rectangle {
        id: red2
        width: 80; height: 80; radius: 40
        color: "red"

        anchors.top: red1.top
        anchors.left: red1.right
        anchors.leftMargin: 60
    }

    Rectangle {
        id: white
        width: 80; height: 80; radius: 40
        color: "white"

        anchors.top: red1.bottom
        anchors.topMargin: 30
        anchors.left: red1.right
        anchors.leftMargin: -10
    }
}
```

Les états se définissent à l'aide de la propriété `states`, qui prend pour valeur une liste d'états State, c'est-à-dire une série d'éléments séparés par des virgules, entre crochets (par exemple, `[State{}, State{}]`). Chaque State possède un nom (`name`) et peut indiquer, avec des composants PropertyChanges, l'ensemble des propriétés dont la valeur change lors du passage à cet état. Ces états sont stockés au niveau d'un composant MouseArea, dont le seul rôle est de récupérer des clics sur toute sa surface. L'état courant est disponible dans la propriété `state`.

Une première manière de réaliser le feu de signalisation consiste à définir tous les états au même endroit : notamment, le code nécessaire pour le clignotement est commun aux trois feux. Ce dernier point nécessite un composant particulier : Timer. Il permet

d'évaluer du code à intervalle régulier. La propriété `running` indique si le minuteur est démarré, `repeat` s'il se réenclenche à chaque fin d'intervalle. `interval` donne, en millisecondes, le délai imparti entre la définition de `running` à `true` et l'exécution du gestionnaire de signal `onTriggered`.

```
ApplicationWindow {
    Rectangle { … }
    // …

    Timer {
```

❶ id: timer running: false repeat: true } MouseArea { ❷ id: click anchors.fill: parent onClicked: { ❸ switch (state) { case "allowed": state = "disallowed"; break case "disallowed": state = "allowed"; break } } **states:** [❹ State { name: "allowed" ❺ PropertyChanges { target: red1; color: "black" } PropertyChanges { target: red2; color: "black" } PropertyChanges { target: white; color: "white" } PropertyChanges { target: timer ❻ interval: 800 running: true onTriggered: { white.color = Qt.colorEqual(white.color, "white") ? "black" : "white" } } }, State { name: "disallowed" PropertyChanges { target: red1; color: "red" } PropertyChanges { target: red2; color: "black" } PropertyChanges { target: white; color: "black" } PropertyChanges { target: timer interval: 500 running: true onTriggered: { ❼ red2.color = Qt.colorEqual(red1.color, "red") ? "red" : "black" red1.color = Qt.colorEqual(red1.color, "red") ? "black" : "red" } } }] state: "allowed" ❽ } }

❷ Les différents états sont mis en place et gérés au sein d'un composant MouseArea, qui délimitera la zone où l'utilisateur pourra cliquer pour changer d'état. Tous les états y sont définis.

❺ Le premier état défini correspond à celui où les voitures peuvent passer : la lampe blanche clignote. Lors du passage à cet état, il faut donc éteindre les deux lampes rouges (définir leur couleur à `"black"`) et allumer la blanche (définir sa couleur à `"white"`). Chaque changement demandé par un PropertyChanges ne s'applique qu'à un seul élément Qt Quick : sa cible `target` ; toutes les autres propriétés indiquent des modifications à opérer.

❶ Un minuteur est requis pour le clignotement.

❹ Les différents états sont stockés dans un tableau au niveau de la propriété `states`.

Ce minuteur est configuré de manière spécifique pour chaque état, avec notamment une fréquence qui varie (les lampes rouges clignotent plus rapidement que la blanche). Il est démarré lors du changement d'état par la propriété `running`. Un changement de valeur d'une propriété peut inclure les gestionnaires de signal : le
❻ code à exécuter diffère selon l'état (une ou deux ampoules à faire varier). Le code proprement dit inverse simplement la couleur donnée ; le point important est que la comparaison de couleurs doit se faire avec `Qt.colorEqual` (en interne, toutes les couleurs sont stockées comme des triplets de nombres au format RGB, alors que la comparaison se fait avec une chaîne de caractères décrivant la couleur).

L'état où le passage à niveau est interdit aux voitures est très similaire au premier, à l'exception du gestionnaire de signal : l'ordre des deux lignes a de l'importance. La
❼ comparaison se fait selon la couleur d'une des deux lampes et son résultat doit rester identique pour les deux changements : si la comparaison se fait sur la première lampe, alors la couleur de la deuxième doit être modifiée d'abord.

❸ Les changements d'état sont déclenchés par un clic dans l'interface, n'importe où. Si l'état était `allowed`, il devient `disallowed` et vice-versa.

Finalement, on définit l'état initial par la propriété `state`. Dans ce cas, Qt Quick effectue la transition vers cet état, ce qui garantit l'initialisation de l'interface.

❽ ***Note*** *> On aurait aussi pu utiliser le gestionnaire de signal* `Component.onCompleted` *(disponible dans tout composant et exécuté dès la fin de son chargement, le plus souvent au démarrage de l'application) pour exécuter une ligne de code comme* `state = "allowed"`.

2. Exploitation des états pour un code plus déclaratif

Cette première manière de faire demeure très impérative malgré l'utilisation d'états : le code fait souvent appel au JavaScript. Il y aurait moyen d'exploiter davantage les états en réécrivant l'application : un état par couleur de feu (les deux lampes rouges sont coordonnées par l'une des deux), plus un global. Le principal intérêt d'utiliser autant d'états est de découpler les actions : l'état principal sera de très haut niveau, il coordonnera les deux couleurs ; ensuite, chaque couleur de lampe gérera ses allumages. Un autre avantage est la simplicité pour ajouter des transitions sur ce nouveau code.

Note *> Cet exemple est disponible dans le dossier* `feu-etats`  *.*

```
ApplicationWindow {
    // …

    Rectangle {
        id: red1
        // …
        states: [
            State {
                name: "blinking"
                PropertyChanges { target: red1; color: "red" }
                PropertyChanges {
                    target: timer
                    interval: 500; running: true
                    onTriggered: red1.color =
                        Qt.colorEqual(red1.color, "red") ? "black" :
"red"
                }
            },
            State {
                name: "off"
                PropertyChanges { target: red1; color: "black" }
            }
        ]
    }

    Rectangle {
        id: red2
        color: (red1.state == "off") ? "black" :
            Qt.colorEqual(red1.color, "red") ? "black" : "red"
        // …
    }
```

```
    Rectangle {
        id: white
        // … (modifications très similaires à red1)
    }

    Timer { … }

    MouseArea {
        id: click
        // …
        states: [
            State {
                name: "allowed"
                PropertyChanges { target: red1; state: "off" }
                PropertyChanges { target: white; state: "blinking" }
            },
            State {
                name: "disallowed"
                PropertyChanges { target: white; state: "off" }
                PropertyChanges { target: red1; state: "blinking" }
            }
        ]
    }
}
```

Note *> Il est possible de pousser la découpe en états encore un cran plus loin, en suivant les mêmes principes : un état éteint, un état clignotant allumé et un état clignotant éteint. Le code correspondant est celui de l'exemple* `feu-etats-propre`  .

3. Définition de transitions

Une fois tous les états bien distincts pour chaque élément, l'ajout de transitions se fait en un rien de temps. La différence est que l'attention se porte cette fois sur la propriété `transitions` (au même niveau que `states`) : elle contiendra une liste de composants Transition, d'un état (`from`) vers un autre (`to`). Chacune de ces transitions indiquera les propriétés dont la valeur doit évoluer de manière plus douce lors du changement d'état, avec une syntaxe identique à celle des changements d'état ; la différence est que la transition ne donne pas de valeur à donner à une propriété, puisqu'elle ne fait qu'adoucir le passage entre deux états, c'est-à-dire deux valeurs définies au niveau des états.

Ici, nous utiliserons le type d'animation ColorAnimation, étant donné que la transition à adoucir correspond à une couleur. D'autres types de transitions existent, comme les PathAnimation pour déplacer des éléments le long d'une courbe définie. SequentialAnimation impose que deux animations se déroulent l'une après l'autre, tandis que ParallelAnimation les fait se dérouler en simultané. [26]

Le code adapté sera très similaire au cas précédent, où les PropertyChanges sont recopiés comme ColorAnimation en précisant la propriété à modifier sous `properties` et le temps de l'animation sous `duration` :

```
ApplicationWindow {
    // …
    Rectangle {
        // …
        states: [
            State { name: "blinkingOn"; … },
            State { name: "blinkingOff"; …},
            State {
                name: "off"
                PropertyChanges { target: red1; color: "black" }
            }
        ]
        transitions: [
            Transition {
                from: "off"; to: "blinkingOn"
                ColorAnimation {
                    target: red1
                    properties: "color"
                    duration: 300
                }
            },
```

```
            Transition {
                from: "blinkingOn"; to: "blinkingOff"
                ColorAnimation {
                    target: red1
                    properties: "color"
                    duration: 200
                }
            },
            // …
        ]
    }
}
```

Note *> Cet exemple est disponible dans le dossier* `feu-transitions` .

La dernière étape pour améliorer ce code serait une factorisation de chaque lampe du feu dans un composant séparé, en suivant les techniques du chapitre Créer un composant réutilisable.

---

[26] La liste des animations est disponible dans la documentation.

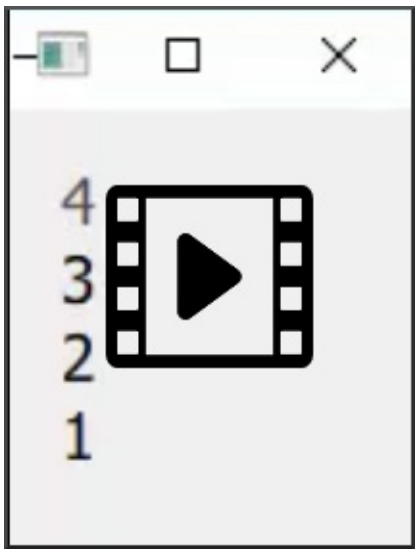
4. Transitions et vues

Les transitions ne servent pas seulement à passer d'un état à un autre (comme à la section précédente), elles peuvent être utilisées dans d'autres contextes : par exemple à l'intérieur des modèles et des vues Qt Quick. Dans ce cas, elles s'appliquent à différents moments (indiqués à la place d'une propriété plus générique `transitions`) : lors de l'ajout (`add`), de la suppression (`remove`), du déplacement (`move`) d'éléments dans la vue. Chaque transition n'a alors plus besoin d'un état initial ou d'un état final : de par la propriété où elle est enregistrée, Qt Quick sait automatiquement quand l'utiliser.

Par exemple, dans une liste, pour changer très progressivement l'opacité de chaque élément ajouté dans le modèle, NumberAnimation permet de changer la valeur de la propriété numérique `opacity` du composant Text utilisé, en passant d'une valeur nulle (`from`, en début d'animation) à une valeur unitaire (`to`, en fin d'animation).

```
ListView {
    // …
    model: model
    delegate: Text {
        text: value
    }

    add: Transition {
        NumberAnimation {
            property: "opacity"
            from: 0
            to: 1.0
            duration: 5000
        }
    }

    focus: true
    Keys.onReturnPressed: model.insert(0, {'value': 4})
}
```

Figure 22.3 : *Transition lors de l'ajout d'un élément dans une vue (vidéo)*


4
3
2
1

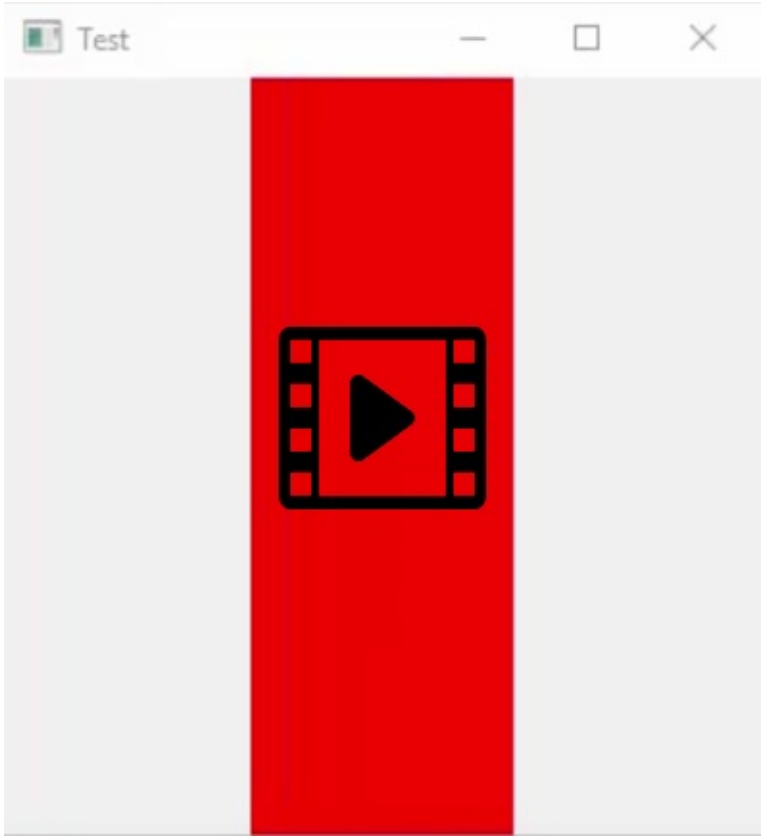
5. Transition lors d'un changement de valeur

Qt Quick propose une syntaxe très simple à utiliser pour les transitions lorsqu'une valeur change (c'est-à-dire lorsqu'un signal est émis), à l'aide du composant Behavior. L'écriture est assez différente :

```
Rectangle {
    // …

    Behavior on width {
        NumberAnimation {
            duration: 2500
        }
    }
}
```

Dans cet exemple, tout changement de la largeur du rectangle s'étalera sur deux secondes et demie. L'exemple `20-behavior-on` propose d'agrandir un rectangle après un certain temps : la hauteur est changée instantanément, tandis que la largeur évolue de manière plus douce grâce à une telle transition. Ce comportement différent n'est pas visible depuis l'extérieur du composant : on peut en changer la valeur normalement, Qt Quick se charge de l'animation.

Figure 22.4 : *Transition sur un changement de valeur (vidéo)*


Test

23

Affichage 2D avec Canvas

Niveau : intermédiaire

Objectifs : dessiner des formes arbitraires avec Canvas

Prérequis : Premiers pas avec Qt Quick, Présentation de JavaScript, Utiliser la méthodologie modèle - vue

Qt Quick a la particularité de n'avoir qu'une seule forme géométrique à sa disposition : le rectangle. Ce choix se justifie, puisque la majorité des interfaces peut s'écrire à l'aide de rectangles, de texte et d'images. Le composant Rectangle dispose quand même d'une option pour des coins arrondis, ce qui permet de dessiner des cercles et des ellipses. Par contre, les composants de base ne proposent rien d'autre : impossible de dessiner un triangle ou une courbe, par exemple.

Une technique pour contourner ces manques est d'implémenter soi-même les formes requises en Python avec une intégration au niveau du graphe de scène (voir Section 3, Communication par composant). Il est aussi possible de tout effectuer en JavaScript, bien confortablement au sein de Qt Quick, à l'aide du composant Canvas. Celui-ci implémente l'API Canvas de HTML5, ce qui permet de capitaliser sur sa documentation.

Note *> À cause de quelques différences entre Qt Quick et les navigateurs web au niveau de l'interpréteur JavaScript, le code HTML5 existant qui utilise l'API Canvas ne fonctionnera pas tel quel dans Qt Quick, mais l'adapter sera très simple. La plus grande différence est qu'il faut déclarer les variables avec le mot clé* `var` *dans le code JavaScript. Pour le reste, la documentation présente les changements à apporter.*

Note *> Le code des exemples de ce chapitre est disponible dans le dossier* `canvas` .

1. Dessin par pinceau/brosse

Tout l'affichage se fait à l'intérieur d'un composant Canvas, une zone rectangulaire où l'on peut dessiner. Cette zone peut être configurée comme tout composant Qt Quick, avec une position et une dimension. Le gestionnaire de signal `onPaint` est exécuté chaque fois que le contenu de la zone est rafraîchi. Voici donc le squelette minimal pour commencer à dessiner :

```
Canvas {
    width: 400
    height: 400

    onPaint: {
        ctx = getContext("2d");
    }
}
```

La fonction `getContext("2d")` sert à récupérer une instance de Context2D, un objet de contexte sur lequel toutes les opérations de dessin seront effectuées (l'équivalent de CanvasRenderingContext2D côté HTML5). Là où Canvas est le composant Qt Quick qui correspond à la zone de dessin, Context2D implémente les opérations sur cette zone.

Une première manière d'utiliser Canvas s'assimile au langage Logo, utilisé pour présenter la programmation aux enfants : l'objectif est de déplacer une tortue-crayon pour réaliser un dessin. Avec Canvas, on commence par définir le pinceau qu'on souhaite utiliser, puis on se déplace dans une direction donnée autant de fois que nécessaire pour effectuer la forme voulue. Ensuite, en fermant le circuit, on peut remplir la forme correspondante avec une couleur ou un dégradé.

```
var ctx = getContext("2d")
```

❶ ctx.lineWidth = 10 ❷ ctx.strokeStyle = ***"darkgreen"*** ctx.fillStyle = ***"darkseagreen"*** ❸ ctx.beginPath() ❹ ctx.moveTo(50, 50) ❺ ctx.lineTo(350, 50) ❻ ctx.lineTo(350, 350) ctx.lineTo(50, 350) ctx.closePath() ❼ ctx.fill() ❽ ctx.stroke() ❾

❶ Comme dans tout code Canvas, il faut commencer par récupérer un contexte pour l'affichage. N'oubliez pas le mot clé `var`, nécessaire pour définir une variable locale.

❷ Ces premiers paramètres définissent le style de la ligne à dessiner, désignée par `stroke` avec Canvas (les coordonnées sont données juste après) : tout d'abord, son épaisseur (`lineWidth`), puis sa couleur (`strokeStyle`, qui accepte aussi des dégradés et des images).

Note *> Dans la terminologie Graphics View, ces paramètres correspondent à ceux d'un pinceau.*

❸ Ensuite, on peut définir les paramètres du remplissage (`fill`) de la forme qui sera définie juste après : `fillStyle` en donne la couleur.

Note *> Dans la terminologie Graphics View, ces paramètres correspondent à ceux d'une brosse.*

❹ Voici le début du dessin proprement dit, avec `beginPath` pour la définition d'un nouveau chemin. Le dessin et le remplissage s'effectueront avec ce chemin.

❺ La première étape consiste à déplacer le pinceau quelque part dans la zone de Canvas avec la méthode `moveTo`. Toutes les coordonnées sont limitées à cette zone. (0, 0) correspond au coin en haut à gauche.

❻ Le point courant (50, 50) est relié au suivant (350, 50) par une ligne droite avec la méthode `lineTo`. On aurait pu utiliser un arc de cercle avec `arcTo`, une courbe quadratique avec `quadraticCurveTo` et une courbe de Bézier avec `bezierCurveTo`.

❼ Une fois le chemin fini, il faut le fermer avec `closePath` : il devient alors utilisable.

❽ Par exemple, on peut remplir ce chemin à l'aide de la fonction `fill`. Le résultat obtenu est montré à la Figure 23.2.

❾ `stroke` sert à dessiner une ligne le long du chemin. La Figure 23.3 montre le résultat obtenu (sans l'appel à `fill` de la ❽).

Note > *Il n'est pas possible de sauvegarder un chemin ainsi créé pour le réutiliser par la suite ! Par contre, vous pouvez écrire une fonction qui recrée ce chemin.*

Figure 23.1 : *Résultat de l'exemple*

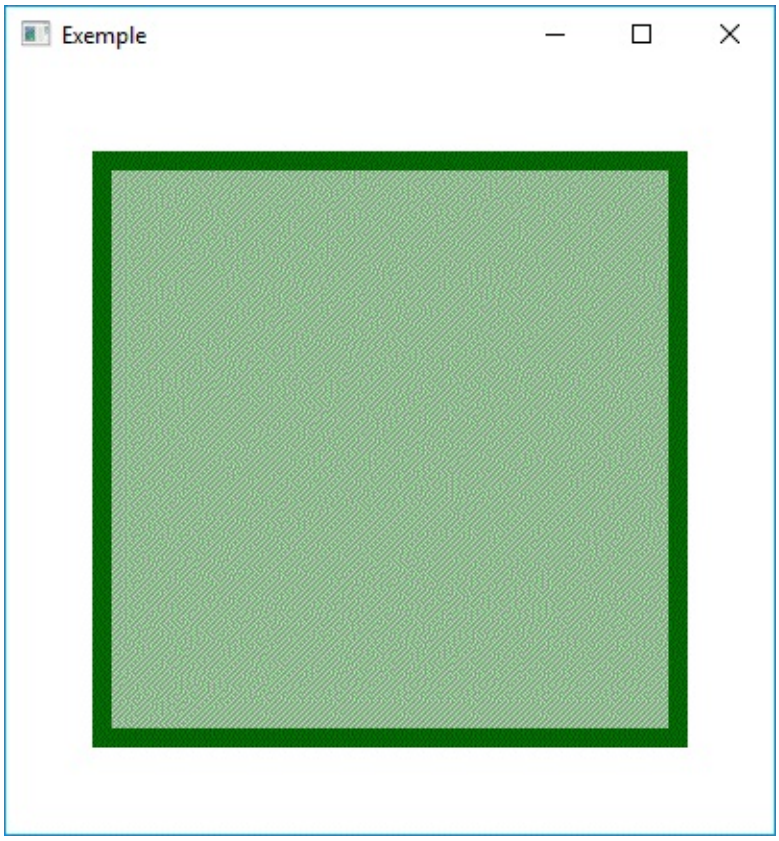


Figure 23.2 : *Résultat de l'exemple en appelant uniquement* `fill` *(sans* `stroke`*)*

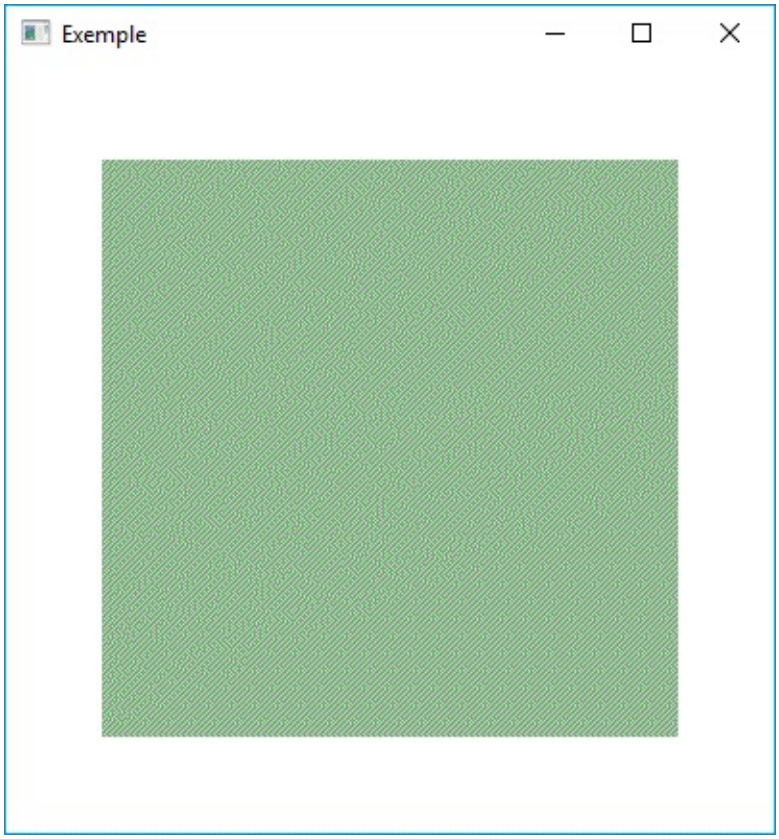
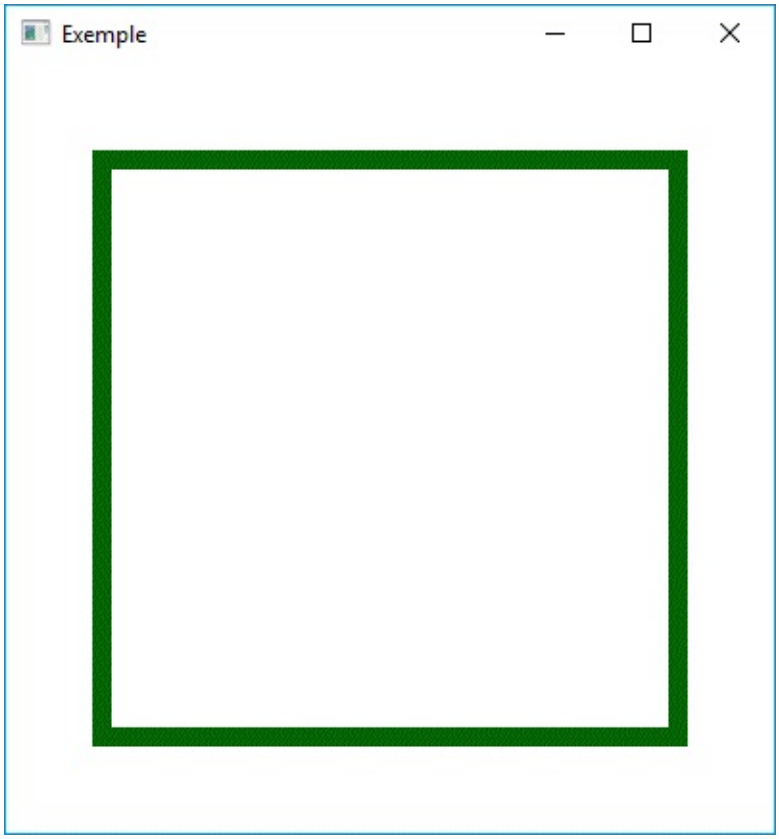


Figure 23.3 : *Résultat de l'affichage en appelant uniquement* `stroke` *(sans* `fill`*)*


Exemple

2. Formes prédéfinies et interactions de base

Pour des formes courantes (rectangles arrondis ou non, ellipses), Canvas propose une série de méthodes qui simplifie la création de chemin, voire le tracé ou le remplissage. Par exemple, `fillRect` remplit un rectangle à l'écran, sans s'alourdir des fonctions pour commencer ou arrêter un chemin : ces opérations sont automatiquement effectuées. `rect` crée un chemin rectangulaire, sans le tracer ou le remplir (il faut donc faire débuter le chemin avec `beginPath` et l'arrêter avec `closePath`).

Contrairement aux autres composants Qt Quick, Canvas ne prévoit pas grand-chose pour traiter les événements reçus. Pour y arriver, il faut coupler cette zone d'affichage avec MouseArea et décider de l'action à effectuer *en fonction de la position du curseur dans le composant Canvas*. À ce niveau, autant que possible, pour chaque zone qui doit réagir à la souris (comme un bouton), mieux vaut utiliser un duo de composants Canvas-MouseArea, afin de faciliter le travail de traitement de l'événement (et laisser le moteur d'exécution de Qt Quick calculer le composant qui doit recevoir l'événement, d'une manière très efficace). Cependant, si la forme n'est pas rectangulaire, il n'y a pas d'autre choix que de gérer les différents événements dans le même composant (voir la Section 4, Détection de collision).

Pour implémenter MosaiQ, la solution la plus simple est probablement de stocker les coordonnées de tous les rectangles à afficher dans un modèle de liste, puis d'afficher ces rectangles à l'aide d'un répétiteur Repeater. Quand l'utilisateur clique sur un rectangle, les quatre sous-rectangles sont ajoutés dans le modèle et celui qui a reçu le clic est supprimé.

Note *> Le code qui suit correspond à l'exemple `mosaiq-canvas` du dépôt Git* . *`mosaiq-rectangle` en est une autre implémentation, équivalente, mais qui n'utilise pas Canvas : elle est plus facile à comprendre, car elle ne fait pas appel à de nouveaux concepts. L' organisation du code est également très similaire.*

```
import QtQuick 2.5
import QtQuick.Window 2.0

Window {
    id: window

    function colour() {
```

❶ return Qt.rgba(Math.random(), Math.random(), Math.random(), 1).toString(); }

```
Repeater { ❷ model: ListModel { ❸ id: mod } delegate: Item { ❹ x: rx; y: ry width: rsize; height: rsize Canvas { ❺ width: parent.width; height: parent.height onPaint: { var ctx = getContext("2d"); ctx.fillStyle = rcolor; ctx.fillRect(0, 0, width, height); ❻ } } MouseArea { ❼ anchors.fill: parent onClicked: { fragment(rx, ry, rsize); mod.remove(index); } } } } Component.onCompleted: { ❽ fragment(0, 0, window.width); } function fragment(_x, _y, _size) { ❾ var half = _size / 2; mod.append({ "rx": _x, "ry": _y, "rsize": half, "rcolor": colour() }); mod.append({ "rx": _x + half, "ry": _y, "rsize": half, "rcolor": colour() }); mod.append({ "rx": _x, "ry": _y + half, "rsize": half, "rcolor": colour() }); mod.append({ "rx": _x + half, "ry": _y + half, "rsize": half, "rcolor": colour() }); } }
```

❶ Les nouveaux rectangles auront une couleur aléatoire, décidée par cette fonction.

❷ Les rectangles sont générés dynamiquement par le composant Repeater…

❸ … selon les entrées du modèle `mod`.

❹ Chaque rectangle de MosaiQ est encadré dans un composant Item, car il faut contenir au même endroit une zone de dessin Canvas et une zone sensible aux clics MouseArea.

Sa taille est donnée par le contenu du modèle (les rôles `rx` et `ry` pour la position, `rsize` pour la taille du côté).

❺ Canvas gère l'affichage d'un seul rectangle, dans une zone très limitée. La taille est reprise de celle de l'Item parent.

❻ La fonction `fillRect` remplit avec la couleur sélectionnée (`ctx.fillStyle`) un rectangle passé en argument : le coin en haut à gauche (0, 0), puis la dimension en largeur et en hauteur. La coordonnée (0, 0) correspond au coin en haut à gauche du composant Canvas courant : le rectangle dessiné sera toujours au bon endroit, étant donné que son parent (Item) est positionné de manière absolue au bon endroit dans la scène.

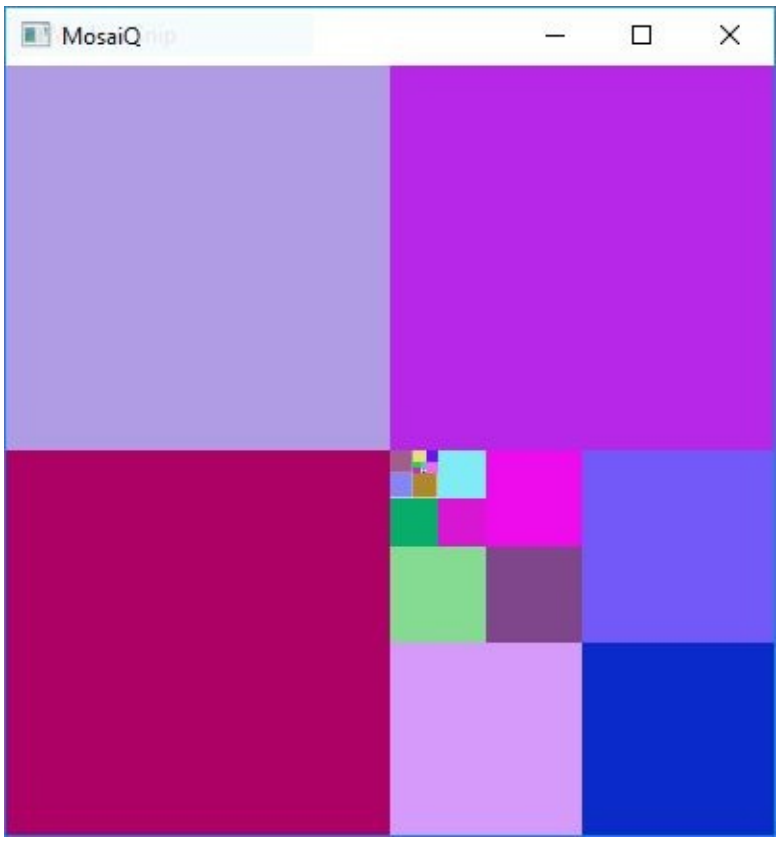
❼ Lors d'un clic, il faut retirer un rectangle du modèle et en ajouter quatre nouveaux,

plus petits. L'ajout des rectangles est effectué par la fonction `fragment` ❾.

❽ Les quatre premiers rectangles sont ajoutés une fois que l'interface est complètement chargée, à l'aide de la fonction `fragment` ❾.

❾ Voici, enfin, l'implémentation de la fonction `fragment`, qui gère les ajouts au modèle (mais pas leur suppression, vu qu'elle n'est nécessaire que pour MouseArea, pas au chargement de l'application).

Figure 23.4 : *MosaiQ avec une certaine fragmentation*



3. Dessiner à la souris

Canvas peut aussi servir à des cas où les appels sont nettement plus fréquents, par exemple dans une application de dessin : l'utilisateur clique et crée ainsi un point à afficher. Quand il déplace la souris, autant de points sont ainsi créés.

C'est exactement le prochain exemple : en fonction des boutons de la souris enfoncés, les points dessinés changent de couleur. Ils seront rouges à l'appui du bouton gauche, verts avec le droit et bleu quand les deux sont enfoncés.

```
Canvas {
    anchors.fill: parent

    onPaint: {
```

❶ if (mouseArea.pressed) { ❷ var ctx = getContext('2d'); if ((mouseArea.pressedButtons & Qt.LeftButton) ❸ && ! (mouseArea.pressedButtons & Qt.RightButton)) { ctx.fillStyle = "red"; } else if (! (mouseArea.pressedButtons & Qt.LeftButton) && (mouseArea.pressedButtons & Qt.RightButton)) { ctx.fillStyle = "green"; } else { ctx.fillStyle = "blue"; } ctx.fillRect(mouseArea.mouseX - 2, mouseArea.mouseY - 2, 5, 5); ❹ } } MouseArea { id: mouseArea anchors.fill: parent acceptedButtons: Qt.LeftButton | Qt.RightButton ❺ onPressed: parent.requestPaint() ❻ onPositionChanged: parent.requestPaint() } }

❶ Cette fonction est appelée chaque fois que l'utilisateur clique ou que Qt envoie un événement de déplacement de la souris (événements `pressed` et `positionChanged`) grâce à ❻ ; le code appelé implémente le comportement souhaité, c'est-à-dire, ici, ajouter le nouveau point à l'écran.

❷ Un détail d'implémentation : au lancement de l'application, le gestionnaire de signal `onPaint` est appelé. À ce moment, personne n'appuie sur la zone sensible, le point correspondant aux coordonnées par défaut est donc dessiné. Cette condition sert à éviter cet effet de bord et à ne laisser l'affichage se produire que lors d'un appui sur la souris.

❹ Un carré de cinq pixels de côté est dessiné au niveau du point actuellement visé par la souris ; les coordonnées de son centre sont mémorisées dans les propriétés `mouseX` et `mouseY` de la zone sensible `mouseArea`.

La couleur du rectangle est sélectionnée en fonction des boutons enfoncés sur la souris. Ils sont indiqués par `mouseArea.pressedButtons`, sous la forme d'une valeur qui condense tous les boutons possibles. Les boutons spécifiques sont alors récupérés un à un par une opération ET binaire : par exemple, `mouseArea.pressedButtons & Qt.LeftButton` prend la valeur vraie quand le bouton gauche est appuyé.

❸ Une comparaison directe comme `mouseArea.pressedButtons == Qt.LeftButton` n'est vraie que quand seul le bouton gauche est appuyé. Ainsi, si le bouton droit, le bouton central ou tout autre bouton présent sur la souris (retour arrière, marche avant ou un autre bouton) est enfoncé, l'égalité stricte ne sera pas vraie. Ainsi, il est assez rare d'utiliser cette comparaison directement, même quand elle pourrait sembler justifiée de prime abord.

❺ Par défaut, MouseArea ne gère que les clics gauche : il faut activer les autres un par un dans la propriété `acceptedButtons`, avec une combinaison similaire à ce qui est utilisé pour la propriété `pressedButtons` ❸, mais utilisant le OU binaire.

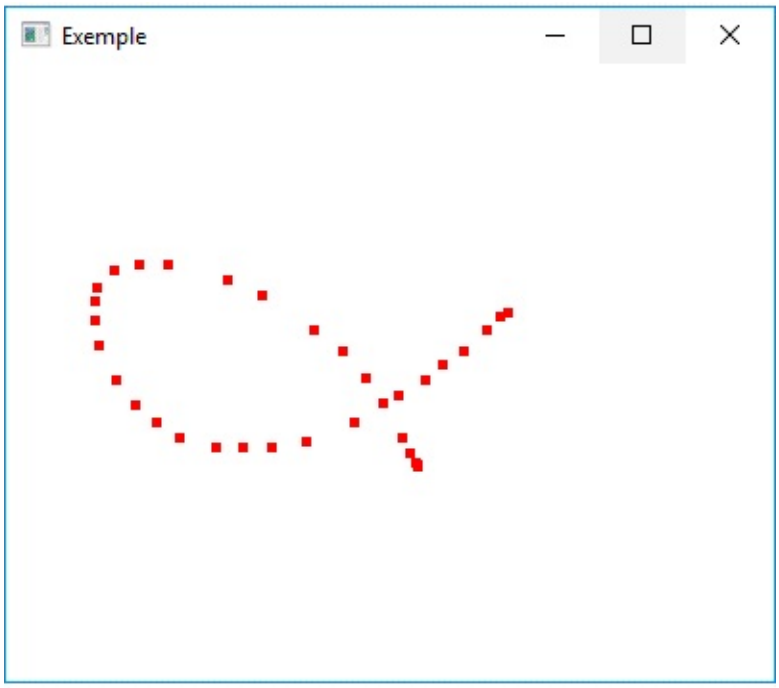
Chaque fois qu'un signal touchant à la souris est détecté (appui et déplacement), les opérations de dessin sont demandées.

❻ La fonction `requestPaint` ne lance cependant pas le rendu immédiatement : elle ne fait que signaler au moteur de rendu de Qt Quick que le composant Canvas doit mettre à jour son affichage (ajouter un point). Ce dernier n'est pas obligé d'accéder à cette requête immédiatement, il peut la retarder (souvent, de quelques millisecondes au plus, ce qui n'a généralement pas d'impact sur le rendu).

Le rendu à l'écran de cette implémentation simple ne sera pas idéal. Notamment, si vous déplacez la souris trop vite, vous passerez pour un adepte de Seurat : les événements de souris ne sont pas envoyés de manière continue, mais bien échantillonnés. Si l'écart entre deux événements est trop grand, les points dessinés seront distants l'un de l'autre. Le problème est qu'il n'est pas possible de contrôler cette fréquence d'échantillonnage ; la solution est donc de mémoriser l'emplacement du point précédent et de dessiner une courbe entre l'ancien point et celui qui vient d'être ajouté — très souvent, il s'agit d'une cerce (*spline*), mais Canvas ne propose que des lignes

droites et des courbes de Bézier (ces dernières nécessitant de préciser des points de contrôle).

Figure 23.5 : *Les déplacements rapides de la souris donnent un rendu en pointillés*



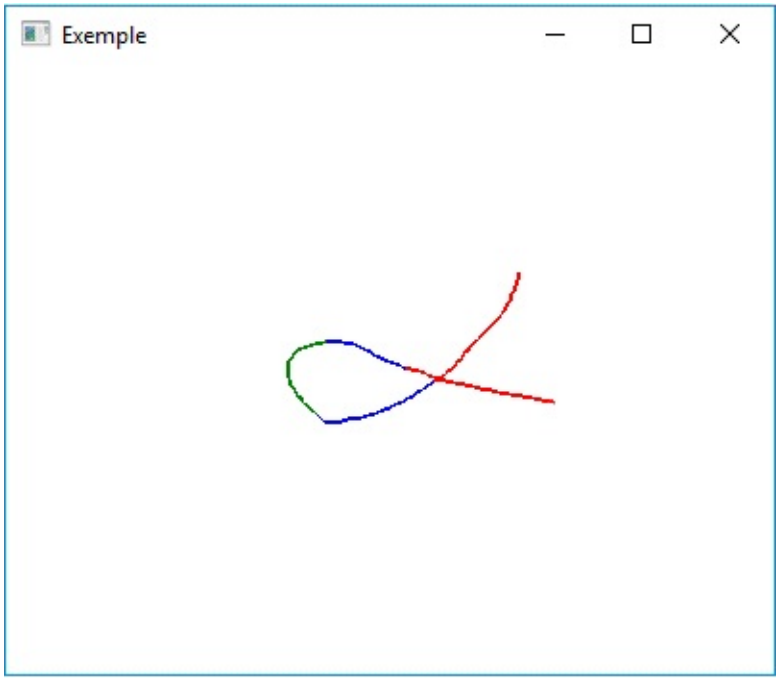
Un autre problème de l'implémentation actuelle est la fragilité par rapport au redimensionnement de la zone de dessin : tout d'abord, elle sera distordue ; ensuite, à la prochaine opération de dessin, tous les points précédents seront effacés.

Pour améliorer la situation, on peut mémoriser la liste des points : par exemple, dans une propriété `points` contenant un tableau de triplets : la position et la couleur de chaque point. Ensuite, chaque opération de dessin affichera l'entièreté des points mémorisés, reliés par des lignes, en faisant varier la couleur au besoin (`strokeStyle`).

```
property var points: []

onPaint: {
    if (mouseArea.pressed) {
        var colour;
        if ((mouseArea.pressedButtons & Qt.LeftButton)
                && ! (mouseArea.pressedButtons & Qt.RightButton)) {
            colour = "red";
        } else if (! (mouseArea.pressedButtons & Qt.LeftButton)
                && (mouseArea.pressedButtons & Qt.RightButton)) {
            colour = "green";
        } else {
            colour = "blue";
```

```
        }
        points.push([mouseArea.mouseX, mouseArea.mouseY, colour]);

        var ctx = getContext('2d');
        for (var i = 1; i < points.length; ++i) {
            ctx.beginPath();
            ctx.strokeStyle = points[i - 1][2];
            ctx.moveTo(points[i - 1][0], points[i - 1][1]);
            ctx.lineTo(points[i][0], points[i][1]);
            ctx.stroke();
        }
    }
}
```

Figure 23.6 : *Dessin en continu avec des lignes droites*



Note *> Cette manière de procéder n'est toujours pas parfaite : si l'utilisateur lâche la souris à un moment et reprend son dessin ailleurs, une ligne sera dessinée entre le point précédent et le nouveau. Il faudrait donc séparer chaque série de points.*

De même, au niveau de la performance, afficher aussi régulièrement un très grand nombre de courbes n'est pas idéal. Il vaudrait mieux ne réafficher tous les points que lorsque c'est nécessaire, c'est-à-dire lors d'un redimensionnement de la zone de dessin. De même, pour faire le lien avec le défaut précédent, il vaudrait mieux ne dessiner qu'un minimum de courbes (effectuer un minimum d'appels à `beginPath`*).*

Note *> Une autre manière d'éviter les effacements lors du redimensionnement serait*

de mémoriser l'image affichée, plutôt que les points qui la constituent.

4. Détection de collision

Dans les cas où le composant Canvas n'est pas entièrement rempli par un élément ou s'il contient plusieurs éléments, la gestion du clic est plus périlleuse : il faut détecter si l'utilisateur a bien cliqué sur un élément qui existe. Dans l'exemple précédent, l'affaire est entendue : le composant Canvas ne contient qu'un rectangle, qui remplit entièrement la zone qui lui est réservée ; un clic dans cette zone correspond forcément au rectangle.

Pour vérifier si l'utilisateur clique dans une forme, on effectue une *détection de collision* un peu particulière, entre un point et une forme. L'algorithme à employer dépend fortement de la forme géométrique considérée ; il cherche à répondre à une seule question : un point (*x*, *y*) est-il dans la forme ? L'algorithme proprement dit dépend de la structure de la forme considérée ; il se trouve généralement facilement sur Internet (par exemple, sur Wikipedia).

Note *> En toute généralité, une détection de collision permet de savoir si deux formes données ont une intersection — par exemple, entre deux carrés, entre une ellipse et un arbre.*

Par exemple, pour réécrire MosaiQ avec des cercles, la modification du code de dessin est assez simple. Il faut cependant faire attention au fait que Canvas ne propose pas de fonction pour directement remplir un cercle, contrairement à un rectangle : on doit créer un chemin, faire appel à la fonction `ellipse` pour mettre à jour le chemin (un cercle est un cas particulier d'une ellipse), fermer le chemin, puis remplir la forme ainsi créée. Les coordonnées passées en paramètres à la fonction `ellipse` correspondent au rectangle dans lequel s'inscrit l'ellipse (ici, plutôt le carré, pour obtenir un cercle) : le point en haut à gauche, la largeur, la hauteur.

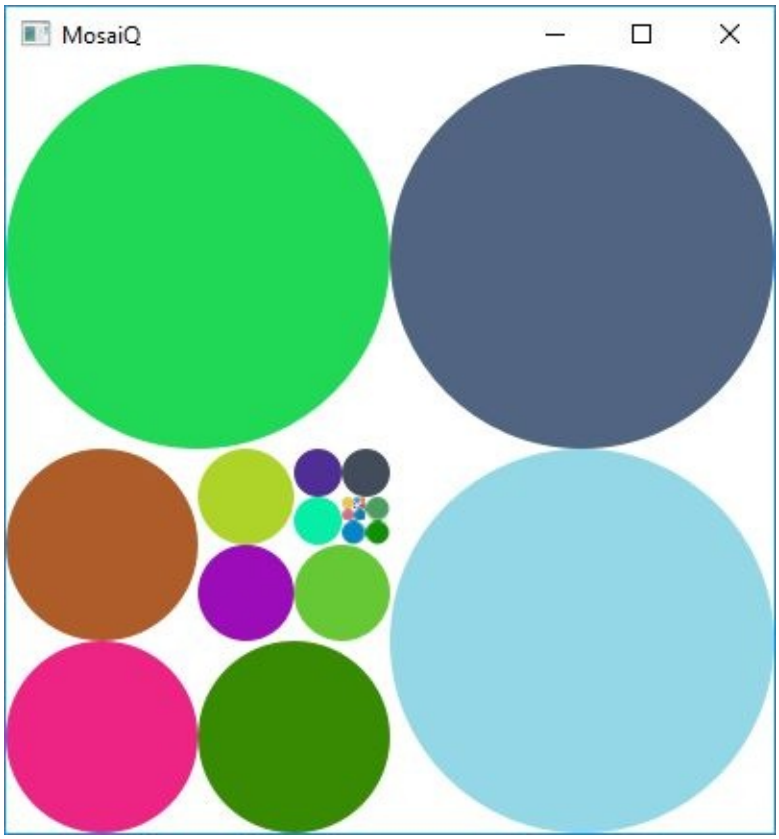
```
var ctx = getContext("2d");
ctx.fillStyle = rcolor;
ctx.beginPath();
ctx.ellipse(0, 0, width, height);
ctx.closePath();
ctx.fill();
```

Cependant, en l'état, le code ne fonctionne pas comme prévu : si l'utilisateur clique en dehors d'un cercle, la division se fait quand même ! La solution consiste à ajouter, dans le code qui gère les clics (gestionnaire de signal `onClicked` de MouseArea), une condition : le point cliqué est-il dans le cercle ? C'est là qu'interviennent les algorithmes de détection de collision. Pour un cercle, l'algorithme le plus courant

repart de la définition du cercle : est dans le cercle tout point qui se situe au plus à une distance `rsize` du centre du cercle.

```
onClicked: {
    if (Math.pow(mouse.x - rsize / 2, 2)
          + Math.pow(mouse.y - rsize / 2, 2)
          <= Math.pow(rsize / 2, 2)) {
        // Modifier le modèle mod.
        // …
    }
}
```

Figure 23.7 : *MosaiQ implémenté avec des cercles et une détection de collision*



Canvas 3D

Selon le même principe que Canvas, Qt Quick propose le module Qt Canvas 3D : il reprend la partie JavaScript de l'API HTML5 de WebGL (tout comme Canvas reprend l'API HTML5 Canvas). La différence principale par rapport à l'API HTML5 est que le dessin est effectué dans un composant Canvas3D, alors que HTML5 réutilise la balise `<canvas>`.

Les principes sont très similaires à ceux de Canvas : tout le rendu est effectué dans une zone délimitée par le composant Canvas3D. L'initialisation correspond au signal `initializeGL` ; ensuite, Qt Quick lance le signal `paintGL` régulièrement pour mettre à jour la scène. Les gestionnaires correspondants peuvent effectuer des appels à l'API WebGL (très similaire à OpenGL 3.0), y compris pour des *shaders*.

De manière générale, cependant, Qt Canvas 3D n'est pas actuellement la manière préférée de faire de la 3D avec Qt Quick : il vaut mieux utiliser Qt 3D (voir Chapitre Affichage 3D avec Qt 3D) — le seul désavantage de ce dernier étant son relatif jeune âge par rapport à Qt Canvas 3D. L'utilité principale de Canvas3D est maintenant de faciliter la migration d'applications HTML5 vers Qt Quick.

24

Affichage 3D avec Qt 3D

Niveau : avancé

Objectifs : intégrer des éléments 3D dans une application Qt Quick

Prérequis : Premiers pas avec Qt Quick, Présentation de JavaScript

L'idée d'introduire des éléments 3D simples à utiliser n'est pas neuve dans l'écosystème Qt, mais il a fallu attendre Qt 5.6 pour sa réalisation : Qt 3D. Il s'agit d'un moteur 3D complet et facile d'emploi : inutile d'apprendre à se servir d'OpenGL ou de DirectX pour intégrer une scène 3D dans une application. Globalement, Qt 3D est capable d'afficher une scène, composée d'une série de formes, la partie visible à l'utilisateur étant définie par une caméra.

***Note** > Cette organisation est similaire à celle de Graphics View ; par exemple, on peut définir plusieurs caméras pour une même scène et ainsi montrer simultanément à l'utilisateur plusieurs vues.*

Cependant, Qt 3D peut faire bien plus que tout ça : du rendu 2D, le stockage de la géométrie d'objets, le rendu avec des matériaux et des shaders… sans oublier une pléthore d'effets (occlusion ambiante, rendu à haute plage dynamique, rendu différé, utilisation de plusieurs textures pour un même polygone, instanciation, UBO, pour le moment). Le moteur est aussi prévu pour l'extensibilité : l'implémentation de toute nouvelle technique de rendu doit être aisée ; à cette fin, la scène à afficher est représentée à l'aide d'un graphe de trame (similaire au graphe de scène utilisé par Qt Quick). Tout est prévu dans Qt 3D pour réaliser des applications 3D, qu'elles soient simples ou complexes (comme un jeu vidéo complet).

De manière générale, Qt 3D se veut *extensible* : même si ces fonctionnalités ne sont pas prévues de base, il est possible d'implémenter un moteur physique, une détection de collision, de l'intelligence artificielle (par exemple, pour un jeu, un calcul de chemin), de l'audio 3D, la détermination de l'objet sélectionné par l'utilisateur à la souris, etc. Pour y arriver, toute la conception du moteur s'est orientée autour des concepts d'*entité* et de *composant*.

Pour rester à la pointe niveau performance, c'est-à-dire exploiter au mieux les cœurs disponibles sur la machine, Qt 3D décompose les calculs à effectuer en *tâches*, qui

peuvent avoir des dépendances entre elles : ainsi, deux tâches indépendantes pourront être effectuées en même temps. Ce mécanisme est raisonnablement simple dans ses concepts, mais permet néanmoins un grand gain de performance.

Note >

- *Le moteur Qt 3D est prévu de telle sorte qu'il puisse être utilisé aussi bien en Python qu'en QML. Les principes sous-jacents sont identiques, mais la syntaxe varie selon le langage.*

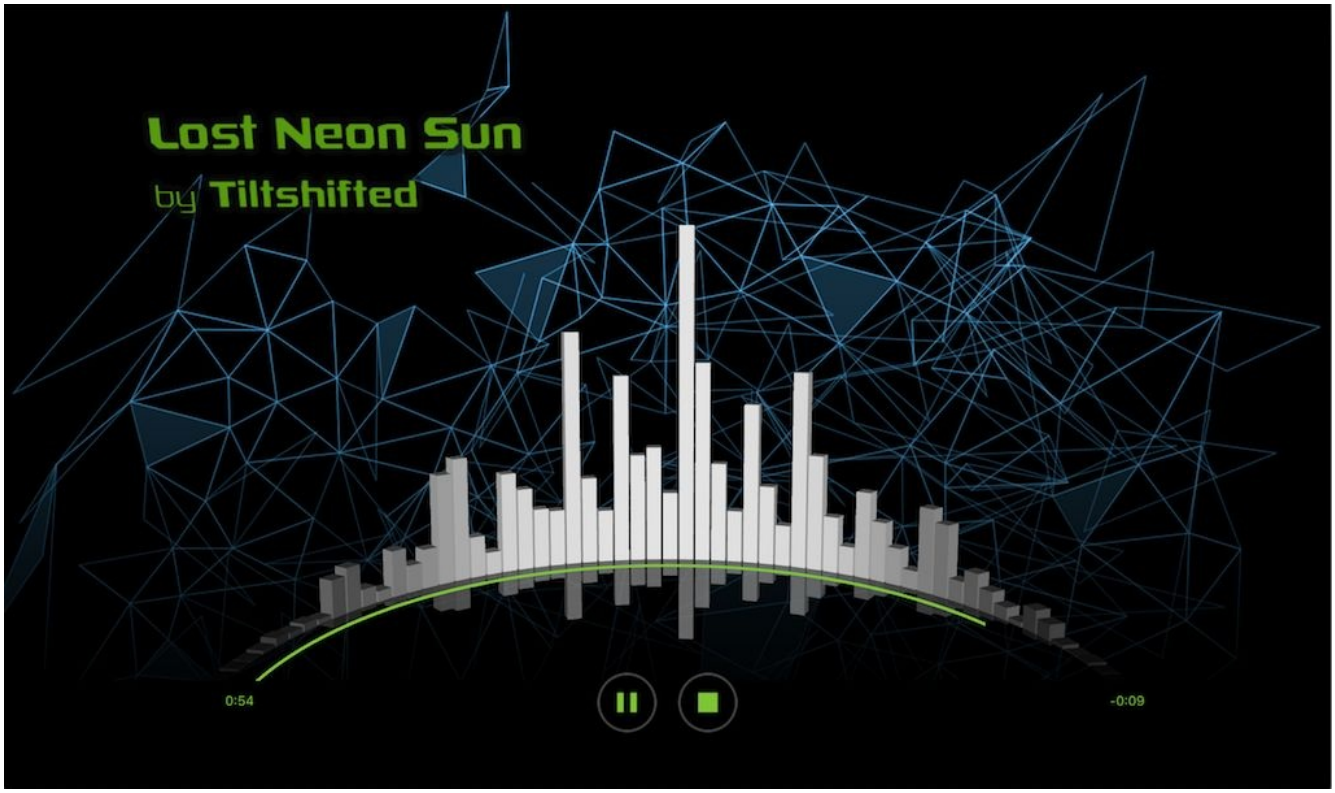
- *Qt 3D est relativement récent et, de ce fait, moins bien documenté que le reste de Qt Quick.*

- *Qt 3D n'est pas fourni avec PyQt, il faut l'installer séparément. La manière la plus simple de procéder est d'utiliser pip :* `pip3 install pyqt3d`*. Vu que le paquet PyQt3D est extrêmement récent, il est très probablement encore absent des dépôts de votre distribution Linux.*

Figure 24.1 : *Exemple d'utilisation : une implémentation du jeu Space Invaders*

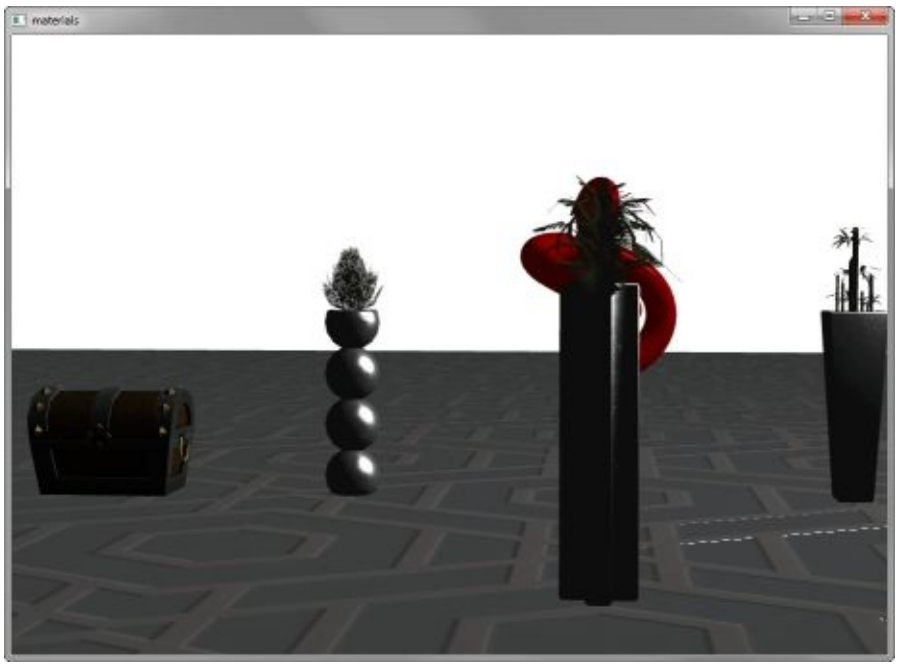
Ici, la 3D est utilisée pour superposer une série de plans (arrière-plan, ennemis en blanc, canon du joueur, etc.). Cette technique est régulièrement utilisée pour coder des jeux en deux dimensions (par exemple, avec Unity).

Figure 24.2 : *Exemple d'utilisation : une visualisation de la musique jouée*



La scène Qt 3D sert à afficher les bandes, tandis que les autres composants sont directement affichés avec Qt Quick. Les sources sont disponibles.

Figure 24.3 : *Exemple d'utilisation : quelques objets affichés en trois dimensions*



Différents matériaux sont utilisés pour l'éclairage, ce qui a notamment un impact sur les reflets, en fonction de la position des lumières. Les sources sont disponibles.

1. Système à entités

Qt 3D est entièrement conçu en tant que système à entités (*entity-component-system*, d'où l'acronyme ECS), une architecture très courante pour des jeux vidéo. Le principe de ce patron de conception est de limiter drastiquement l'utilisation de l'héritage dans un système : d'habitude, chaque élément affiché (par exemple, pour Space Invaders, un canon, un ennemi) possède des données (position, points de vie, etc.) et un certain comportement (pour l'affichage, la réaction aux entrées de l'utilisateur, notamment). Au contraire, avec un système à entités, le comportement est abstrait des données, de telle sorte qu'il peut être réutilisé sans héritage. Ainsi, on compte quatre types d'éléments :

- une *entité* (QEntity en Python, Entity en QML) identifie un objet du jeu, mais sans aucun comportement ou donnée (une entité ne sert qu'à agréger des *composants*). Par exemple, le canon du joueur et chacun des ennemis sera une entité à part entière. Une scène contiendra un très grand nombre d'entités, chacune étant une instance de QEntity ou d'Entity ;

- un *composant* (QComponent en Python, Component3D en QML) se greffe à une entité pour lui ajouter des données et indiquer au système qu'un certain comportement s'applique (l'implémentation du comportement étant gérée par un *aspect*). Par exemple, être affichable peut être indiqué par un composant (un canon, un ennemi), émettre un bruit également (pour le canon du joueur). Pour créer un nouveau composant, il faut hériter de QComponent ou de Component3D ;

- un *aspect* (QAbstractAspect en Python) est l'implémentation d'un *comportement* indiqué par un composant. Par exemple, le rendu à l'écran est implémenté comme un aspect (QRenderAspect) : pour dessiner une image à l'écran, le code de cet aspect passe en revue toutes les entités et, pour celles qui ont un composant de rendu, utilise les informations de ce composant pour afficher l'entité correspondante ;

- le *système* (QAspectEngine en Python) gère les interactions entre ces trois parties, notamment l'échange de messages lors du changement de valeur d'une propriété entre les différents composants.

Une application Qt 3D relativement simple n'aura que deux types d'éléments : des entités et des composants. Si elle est plus complexe, certains comportements devront être implémentés à l'aide d'aspects (ce qui ne peut pas se faire en JavaScript). Le système, quant à lui, est instancié lors de la création de la fenêtre Qt 3D.

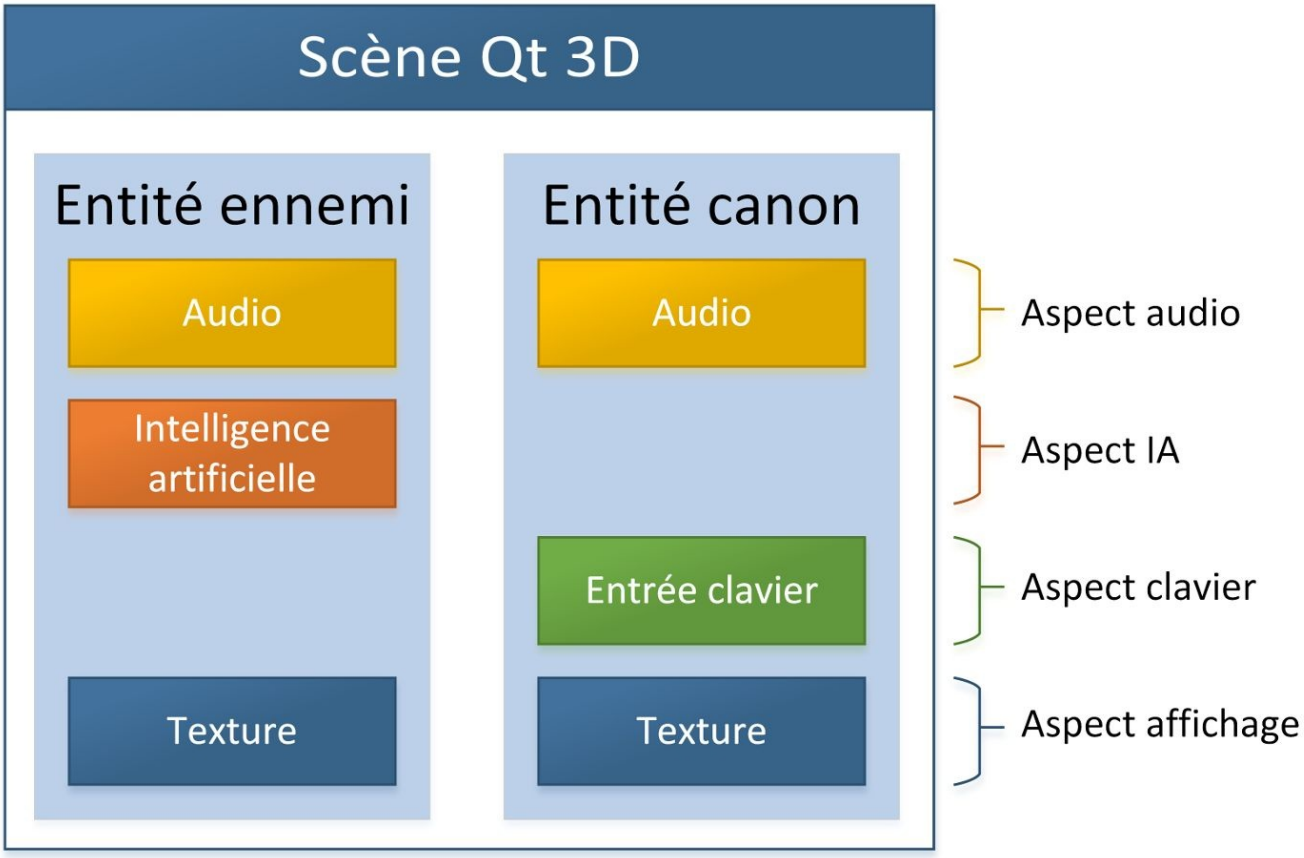
Par exemple, pour le jeu Space Invader (voir Figure 24.1), la scène comporterait une série d'entités :

- une par ennemi (en blanc) ;
- une par balle tirée (par le joueur ou l'ennemi) ;
- une par bloc de défense (en vert) ;
- une pour le canon du joueur ;
- une pour le sol.

Chacune de ces entités est affichée à l'écran : elles auront toutes un composant pour l'affichage. Les ennemis et le joueur émettent des sons lors d'un tir : ces entités auront un composant pour l'émission de bruits. Le canon est contrôlé par le joueur : ce comportement impose que l'entité canon possède un composant de contrôle par le clavier (ou autre). Les ennemis doivent décider quand effectuer un tir : cette décision correspond à un composant ajouté à chaque entité d'ennemi.

À l'exécution, le système passe régulièrement sur les aspects et les entités : chaque fois qu'il rencontre un composant dans une entité, il exécute l'aspect correspondant. Ce dernier peut décider de modifier des propriétés de l'entité, d'émettre un son, etc.

Figure 24.4 : *Space Invaders : entités ennemi et canon*

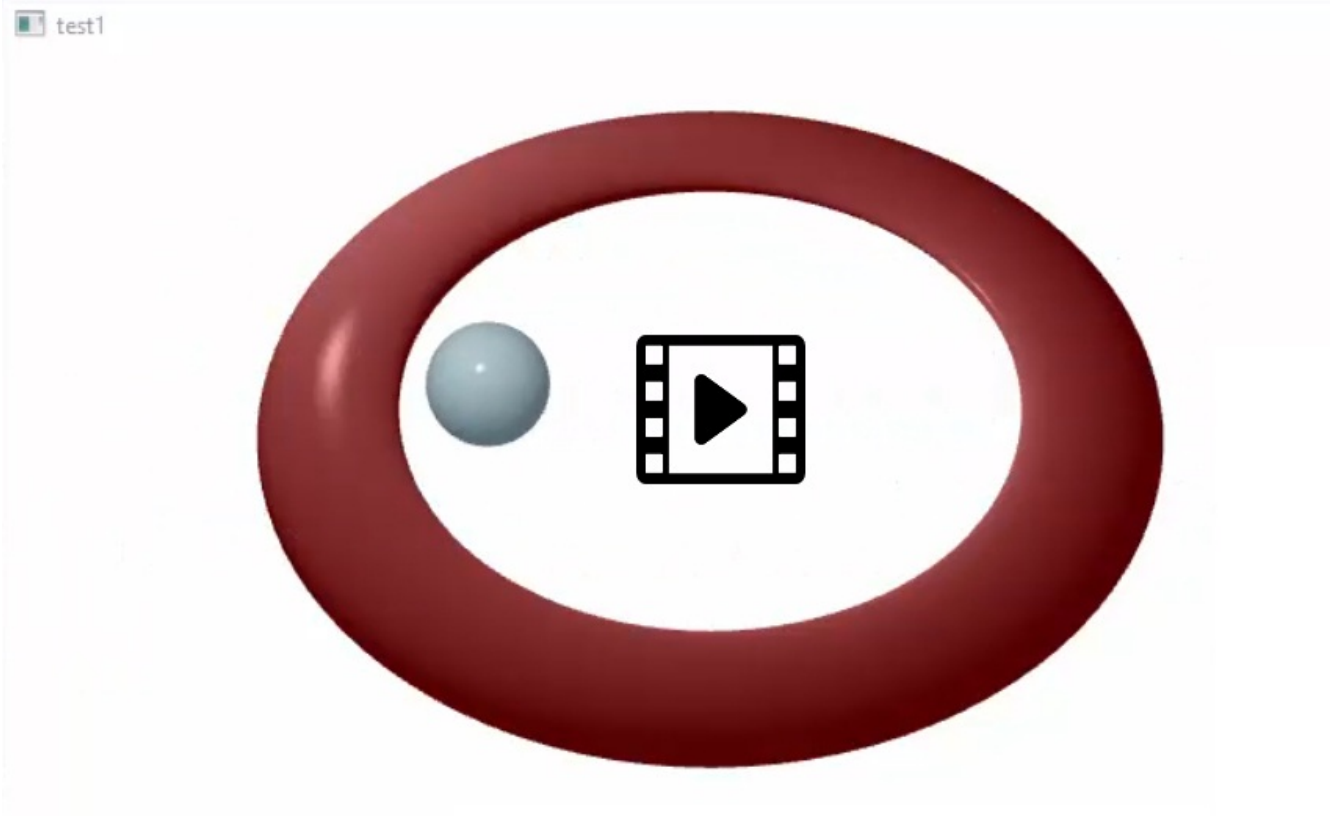

Scène Qt 3D
Entité ennemi
Audio
Intelligence artificielle
Texture
Entité canon
Audio
Entrée clavier
Texture
Aspect audio
Aspect IA
Aspect clavier
Aspect affichage

2. Utilisation basique

Une scène Qt 3D doit avoir, à sa racine, une entité. Elle sera associée aux composants nécessaires au rendu, y compris le positionnement de la caméra par rapport à la scène. Ensuite, le contenu de la scène est défini dans une série d'entités enfants. Ce premier exemple sera assez simple : une bille tourne autour d'un tore (voir la vidéo de la Figure 24.5). Bien que simple, cet exemple montre les principes pour créer une scène Qt 3D… et le degré d'intégration avec le reste de l'environnement Qt Quick : pour faire tourner la bille, il suffit d'utiliser une animation sur sa position.

Note *> Le code des exemples de ce chapitre est disponible dans le dossier* `qt3d`*. Le premier exemple développé dans cette section correspond au dossier* `basique` .

Figure 24.5 : *Scène basique avec Qt 3D : une bille tourne autour d'un tore (vidéo)*



La première étape consiste donc à définir une entité racine. Elle n'a d'autre but que de rassembler les autres entités de la scène et les composants nécessaires au rendu.

```
import QtQuick 2.7
```

```
import Qt3D.Core 2.0
```

❶ import Qt3D.Render 2.0 import Qt3D.Extras 2.0 Entity { ❷ components: [❸ RenderSettings { ❹ activeFrameGraph: ForwardRenderer { ❺ camera: Camera { ❻ id: camera position: Qt.vector3d(0.0, 0.0, -40.0) ❼ viewCenter: Qt.vector3d(0.0, 0.0, 0.0) ❽ fieldOfView: 45 ❾ } } }] // … }

❶ Pour utiliser Qt 3D, il faut importer de nouveaux modules. `Qt3D.Core` contient les éléments principaux, nécessaires pour le fonctionnement du système à entités. `Qt3D.Render` s'occupe exclusivement du rendu, avec des techniques de rendu implémentées dans `Qt3D.Extras` .

❷ Une entité est définie par le composant Entity. Celui-ci ne peut contenir que deux choses : des composants (propriété `components`) et des enfants (notamment des entités, qui feront alors partie de la scène). Il ne s'agit d'enfants qu'au sens de Qt Quick (comme un modèle est créé au niveau d'une fenêtre parce qu'il est utilisé par plusieurs vues), pas dans une quelconque hiérarchie de Qt 3D.

❸ Les composants d'une entité sont définis à l'aide de la propriété `components`, sous la forme d'une liste. Cette liste peut contenir une série de composants, comme ici, ou bien une série d'*identifiants* de composants (il n'est pas possible de mélanger les deux). Dans ce dernier cas, les composants sont déclarés comme enfants de l'entité (souvent la racine ou bien une entité parente de toutes les utilisations des composants ainsi déclarés).

❹ Un seul composant est ici défini et il paramétrise le rendu. Le composant RenderSettings contient principalement un graphe de trame, c'est-à-dire la manière d'effectuer le rendu. Ici, le nœud racine du graphe de trame est ForwardRenderer ❺, la technique la plus courante de rendu actuellement : chaque objet est affiché indépendamment des autres, les lumières et ombres sont calculées de manière distincte (les jeux utilisent régulièrement une technique de rendu différé, où les lumières sont calculées sur la scène complète). [27]

Le principal paramètre pour effectuer le rendu est la caméra, c'est-à-dire l'emplacement du spectateur qui visualise la scène. Cette caméra possède un certain nombre de paramètres :

- la position de la caméra ❼ indique où se situe la caméra dans la scène ;
- la direction de la caméra ❽ est donnée par le point vers lequel elle est dirigée dans la scène ;
- le champ de vue ❾ s'exprime en degrés et indique à quel point la vue est large : à position égale, plus le champ de vue est large, plus la caméra voit une grande partie de la scène.

❻

Au sens strict, une caméra est une entité qui possède un composant CameraLens, mais il n'y a pas besoin de ce niveau de complexité ici : Camera fournit une abstraction largement suffisante à un certain nombre de besoins, y compris pour des mouvements du champ de vision.

2.1. Entités et composants de la scène

Ensuite, on peut s'occuper d'un des deux éléments de la scène — le tore, par exemple. Celui-ci doit avoir une géométrie fixée (le tore), un matériau (sa couleur, sa manière de réagir à l'éclairage) et une position (indiquée par une transformation). Ainsi, l'entité représentant le tore devra avoir trois composants.

```
Entity { // Racine
    components: [ … ]

    Entity { // Tore
        components: [
            TorusMesh {
```

❶ radius: 5.5 rings: 100 slices: 20 }, PhongMaterial { ❷ ambient: Qt.darker("maroon", 1.5) }, Transform { ❸ scale3D: Qt.vector3d(1.5, 1.5, 0.5) rotation: fromAxisAndAngle(Qt.vector3d(1, 0, 0), 45) }] } // … }

Le tore est défini par une géométrie, qui correspond au composant TorusMesh. Le paramètre principal est `radius`, qui définit le rayon du tore. Ensuite, les paramètres `rings` et `slices` définissent le nombre de polygones composant le tore : si ce nombre est trop faible, l'apparence ne sera pas satisfaisante (le tore ne sera pas rond, mais anguleux) ; s'il est trop élevé, la scène prendra beaucoup de temps de calcul.

❶

Le matériau définit la couleur à appliquer sur la géométrie et sa réaction face à la lumière. Le composant PhongMaterial est basé sur le modèle d'éclairage de Phong. En termes simples, ce modèle définit trois couleurs :

- la couleur d'ambiance (`ambiant`), qui ne dépend pas de l'éclairage externe ;
- la couleur de diffusion (`diffuse`), émise par les surfaces rugueuses (relativement uniforme sur la surface de l'objet) ;
- la couleur spéculaire (`specular`), émise par les surfaces brillantes (qui présentent des points lumineux très intenses). Le degré de brillance est contrôlé par le paramètre `shininess`.

PhongMaterial propose des valeurs par défaut pour chacun de ces paramètres.

Le positionnement de l'objet est contrôlé par une transformation. Ici, deux transformations sont combinées :

- une mise à l'échelle `scale3D`, indiquée par trois nombres : chacun correspond à la mise à l'échelle dans chacune des trois dimensions (les trois nombres sont rassemblés par `Qt.vector3d`[28]). La valeur `1.0` garde une taille constante dans cette dimension, une valeur inférieure diminue la taille (elle aplatit le tore selon cet axe), une valeur supérieure l'augmente. Ici, les valeurs données aplatissent le tore selon l'axe vertical, pour se rapprocher d'un frisbee avec un trou au centre ;
- une rotation `rotation`, indiquée par un objet spécifiant la rotation[29]. La fonction `fromAxisAndAngle` donne une manière simple de représenter une rotation, avec un axe (les trois premiers nombres, à nouveau rassemblés par `Qt.vector3d`) et un angle (en degrés).

Une transformation peut aussi mémoriser une translation (`translation`), à nouveau indiquée par trois nombres (rassemblés par `Qt.vector3d`), chacun indiquant le déplacement dans une dimension.

***Note** > Qt 3D s'appuie sur des notions classiques de synthèse d'images, qu'il n'y a pas lieu d'approfondir dans le cadre de cet ouvrage. Si vous n'êtes pas très à l'aise avec celles-ci, nous vous invitons à consulter le livre Synthèse d'images avec OpenGL(ES), écrit par Pierre Nerzic, éd. D-BookeR (2017). Vous y trouverez en*

particulier des explications sur le modèle de Phong (chapitre Calculs d'éclairement) et les transformations (chapitre Transformations géométriques).

La sphère constitue la dernière des trois entités de la scène. Elle aura trois composants, comme le tore, mais aussi une animation sur l'angle de rotation (gérée avec NumberAnimation). Cette animation n'est pas un composant au sens de Qt 3D, mais bien un composant Qt Quick habituel ; en tant que telle, elle est ajoutée comme enfant de l'entité sphère (comme ce qui se ferait en dehors de Qt 3D).

```
Entity { // Racine
    components: [ … ]

    Entity { … } // Tore

    Entity { // Sphère
        components: [
            SphereMesh {
```

❶ radius: 2 }, PhongMaterial { ambient: Qt.darker("lightblue", 2) }, Transform { id: sphereTransform property real angle: 0.0 ❷ matrix: { ❸ var m = Qt.matrix4x4(); m.rotate(angle, Qt.vector3d(0, 1, 0)); m.translate(Qt.vector3d(20, 0, 0)); return m; } }] NumberAnimation { ❹ target: sphereTransform property: "angle" duration: 5000 from: 0; to: 360 loops: Animation.Infinite running: true } } }

❶ Précédemment, on utilisait une géométrie de tore (TorusMesh) ; ici, comme il s'agit d'une bille, on utilise une géométrie de sphère (SphereMesh). Contrairement au cas du tore, les paramètres par défaut produisent une sphère de très bonne apparence ; cependant, on pourrait jouer sur les mêmes paramètres `rings` et `slices` pour améliorer le rendu s'il n'était pas bon ou pour améliorer la performance au détriment de la qualité du rendu.

❷ L'angle de la transformation est une propriété qui changera régulièrement à l'exécution : c'est grâce à elle que la bille pourra faire le tour du tore.

À la différence de la transformation du tore, on utilise ici une matrice pour représenter la transformation de la sphère[30]. Cette manière d'écrire une transformation reprend les informations des translations, des rotations et des mises à l'échelle[31]. On crée une matrice avec `Qt.matrix4x4()`, puis on la modifie en ajoutant des transformations élémentaires :

❸
- une rotation avec la fonction `rotate`, qui prend en argument un angle et une rotation (comme pour le tore) ;
- une translation avec la fonction `translate`, qui prend en argument trois nombres.

(Ces opérations ne sont pas définies dans la documentation de Qt Quick, mais bien dans celle de Qt.)

❹ Pour finir, une animation s'occupe de faire varier l'angle de la bille (la propriété `angle` de `sphereTransform`) autour du tore, entre 0 et 360° (c'est-à-dire un tour complet). Un tour dure cinq secondes (`duration` est exprimé en millisecondes). L'animation se répète sans fin (`loops: Animation.Infinite`) et se lance en même temps que l'application (`running: true`).

***Note** > Bien évidemment, il est possible d'instancier d'autres types de géométrie que des sphères ou des tores. En toute généralité, le composant Mesh peut charger un fichier de géométrie, grâce à sa propriété `source`.*

***Note** > Le composant Transform définit déjà une propriété `angle`, mais cet exemple redéfinit une propriété qui porte le même nom ❷. Ainsi, la propriété `angle` de Transform est cachée. Ceci évite un problème de dépendance circulaire : spécifier une transformation avec `matrix` ❸ impose la valeur à toute une série de propriétés (`scale`, `angle`, etc.) ; si on donne la valeur d'`angle` en même temps, elle doit donc prendre plusieurs valeurs simultanément (celle donnée directement dans le code et celle donnée implicitement par `matrix`). De même, donner un `angle` crée, de manière interne, une nouvelle `matrix` de transformation.*

Si on enlevait les mots clés `property real`, on aurait le message d'erreur suivant :

```
QML Transform: Binding loop detected for property "matrix"
```

***Attention** > La scène Qt 3D n'est pas encore utilisable : la racine est une entité, qui n'a pas de représentation à l'écran telle quelle. (En C++, on peut utiliser la classe Qt3DQuickWindow pour lancer directement ce fichier QML, mais elle n'existe pas encore dans PyQt3D.)*

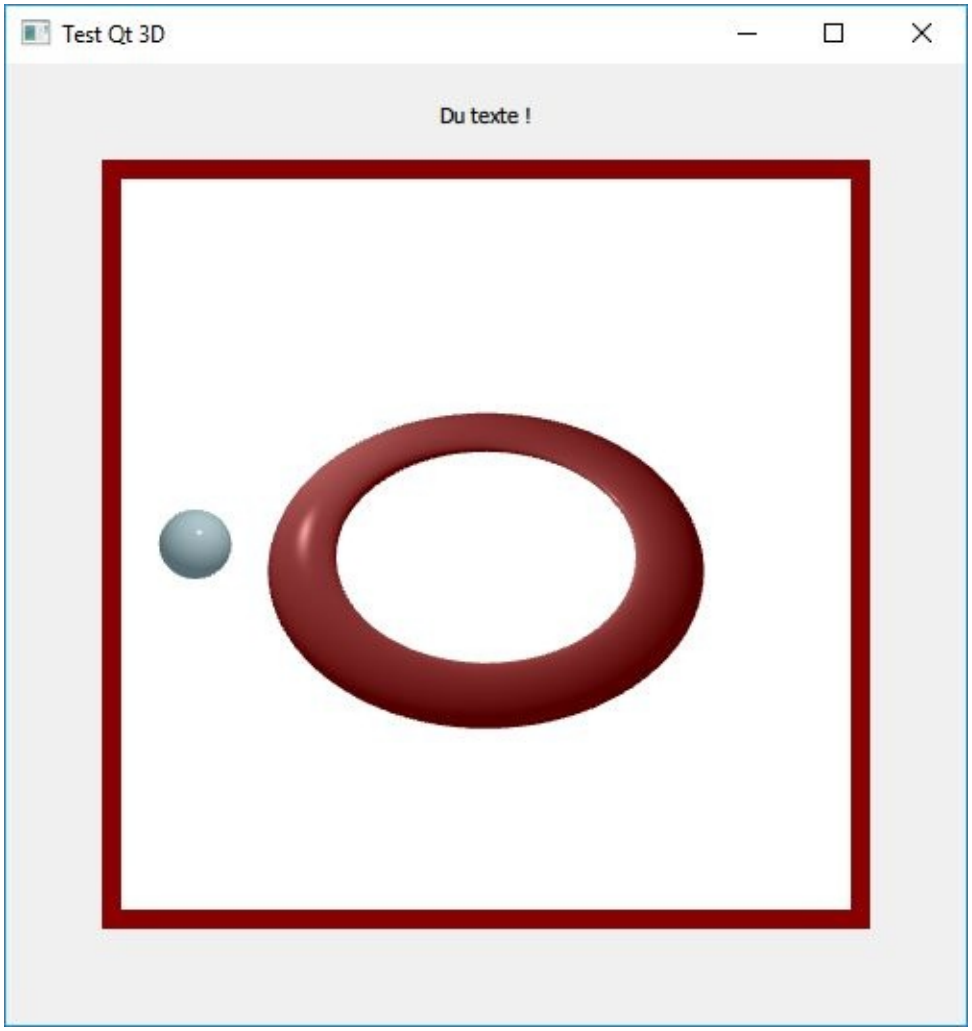
2.2. Intégration à une application Qt Quick

existante

Pour intégrer une partie Qt 3D à une application Qt Quick, la seule chose à faire est d'entourer l'entité racine dans un composant qui s'occupera de l'afficher : Scene3D. À titre d'exemple, on affichera un peu de texte, puis la scène Qt 3D encadrée : pour ce faire, un premier rectangle sera créé ; ensuite, la scène 3D occupera une bonne partie de ce rectangle, en laissant des marges.

```
import QtQuick 2.7
import QtQuick.Controls 1.4
import QtQuick.Scene3D 2.0
import Qt3D.Core 2.0
import Qt3D.Render 2.0
import Qt3D.Extras 2.0

ApplicationWindow {
    // …

    Text {
        text: "Du texte !"
        // …
    }

    Rectangle {
        anchors.fill: parent
        anchors.margins: 50
        color: "darkRed"

        Scene3D {
            anchors.fill: parent
            anchors.margins: 10

            Entity {
                // …
                // La scène Qt 3D, comme précédemment.
            }
        }
    }
}
```

Figure 24.6 : *Scène Qt 3D intégrée à une application Qt Quick*



***Note** > Un lanceur Python habituel continue de fonctionner avec Qt 3D (comme à la Section 1, Lancement d'une fenêtre Qt Quick par Python).*

---

[27] Pour plus d'informations, voir l'article Forward Rendering vs. Deferred Rendering. Vous pouvez aussi vous reporter au livre Synthèse d'images avec OpenGL(ES), chapitre Calculs d'éclairement, écrit par Pierre Nerzic, éd. D-BookeR (2017).

[28] Cette fonction crée un vecteur à trois dimensions. Ici, le vecteur n'est utilisé que pour rassembler trois nombres, mais il est plus souvent utilisé de manière plus habituelle (comme en mathématiques et en physique).

[29] Plus précisément, un quaternion.

[30] Mathématiquement, il s'agit d'une matrice carrée d'ordre 4. Elle travaille avec des coordonnées homogènes, d'où la quatrième dimension, malgré leur application dans des situations avec trois dimensions.

[31] Précision à l'adresse des matheux : toutes les transformations affines peuvent être représentées de la sorte, ce qui inclut aussi des dilatations ou des transvections.

3. Réaction aux stimuli de l'utilisateur

Bien évidemment, une scène Qt 3D peut réagir à des stimuli de la part de l'utilisateur, comme toute application Qt Quick. Par contre, les mécanismes habituels (à base de MouseArea pour la souris, par exemple) ne fonctionnent pas. Avec Qt 3D, il faut ajouter un composant à l'entité racine pour indiquer les paramètres concernant les entrées de la part de l'utilisateur :

```
components: [
    RenderSettings {
        // …
    },
    InputSettings {
        eventSource: window
    }
]
```

Ensuite, l'entité à la racine de la scène Qt 3D doit posséder au moins un périphérique (comme KeyboardDevice pour le clavier) pour récupérer une série d'événements de la part de l'utilisateur. Chaque entité de la scène peut alors faire référence à ces périphériques pour en récupérer les événements, grâce à de nouveaux composants (pour le clavier, KeyboardHandler) — cette référence correspond à la propriété `sourceDevice`.

Note *> Jusqu'à présent, les composants utilisés appartenaient aux modules Qt3D.Core, Qt3D.Render et Qt3D.Extras. Pour l'interaction avec l'utilisateur, il faudra aussi utiliser le module Qt3D.Input.*

3.1. Clavier

La première étape pour récupérer des informations du clavier est de définir le périphérique à la racine :

```
Entity {
    components: [
        RenderSettings { … },
        InputSettings { … }
    ]

    KeyboardDevice {
```

```
        id: keyboard
    }
```

Ensuite, le traitement des informations en provenance du clavier n'est pas très différent d'une interface Qt Quick traditionnelle : les mêmes gestionnaires d'événements sont présents, mais pas au même endroit. Avec Qt 3D, tout se fait au niveau de KeyboardHandler, ajouté comme composant à l'entité qui doit réagir. Par exemple, pour que le tore réagisse à l'appui sur la touche Tabulation pour passer le focus à la sphère (cette notion est la même que pour une application Qt Quick traditionnelle) :

```
Entity {
    components: [
        TorusMesh { … },
        PhongMaterial { … },
        Transform { … },
        KeyboardHandler {
            id: torusKeyboard
            sourceDevice: keyboard
            onTabPressed: sphereKeyboard.focus = true
        }
    ]
    // …
}
```

La sphère est modifiée de la même manière, mais en faisant passer le focus au tore.

Pour que l'impact soit bien visible à l'utilisateur, les deux entités (tore et sphère) auront une taille qui augmentera quand ils auront le focus (par exemple, elle doublera). Pour ce faire, la mise à l'échelle de leur transformation (`scale`) dépendra de cette condition. Ainsi, pour le tore :

```
Transform {
    scale3D: // …
    rotation: // …

    scale: torusKeyboard.focus ? 1 : 2
    Behavior on scale {
        NumberAnimation {
            duration: 250
        }
    }
}
```

L'instruction `Behavior on scale` adoucit la transition entre les deux tailles (voir Section 5, Transition lors d'un changement de valeur).

Côté sphère, l'implémentation est légèrement différente, puisqu'on utilise une matrice

de transformation : étant donné que cette matrice `matrix` est calculée en fonction de la propriété `scale` de Transform et que la valeur de la matrice est déjà imposée, on ne peut pas modifier ce champ directement. La solution déjà avancée est de définir une nouvelle propriété `scale`, qui remplacera celle héritée de Transform.

```
Transform {
    property real scale: sphereKeyboard.focus ? 1 : 2
    Behavior on scale {
        NumberAnimation {
            duration: 250
        }
    }

    property real angle: 0.0
    matrix: {
        var m = Qt.matrix4x4();
        m.rotate(angle, Qt.vector3d(0, 1, 0));
        m.translate(Qt.vector3d(20, 0, 0));
        m.scale(scale);
        return m;
    }
}
```

Figure 24.7 : *Interaction avec le clavier : à chaque appui sur la touche Tabulation, un des deux éléments devient plus gros, l'autre élément reprend sa taille initiale (vidéo)*

3.2. Sélection d'objets à l'écran

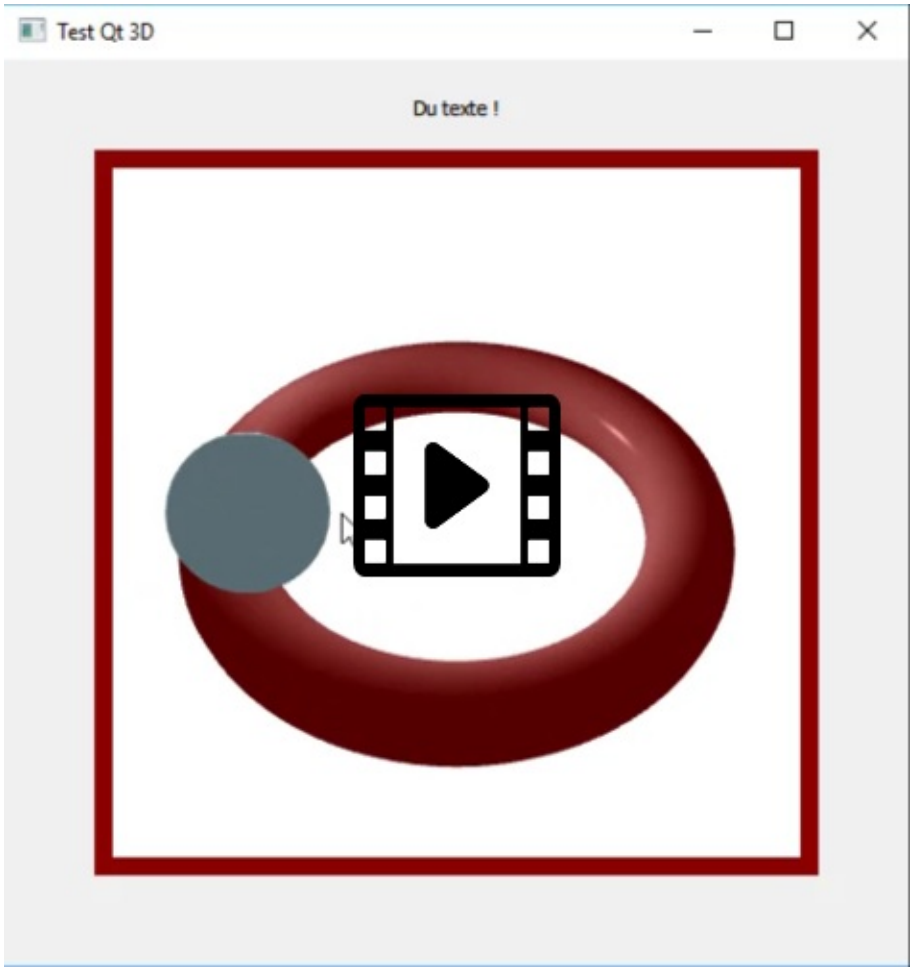
Dans une application 3D, l'utilisateur souhaite souvent sélectionner des éléments dans une scène et interagir avec eux (afficher un menu contextuel, changer la position de l'élément, etc.). Dans le vocabulaire de Qt 3D, cette sélection à l'écran s'appelle *picking*. L'implémentation est un peu plus simple que pour les interactions avec le clavier : il n'y a pas besoin de faire référence à un objet qui représente un périphérique d'entrée (la souris, dans le cas présent).

Pour qu'une entité affichée soit sélectionnable, il suffit de lui ajouter un composant ObjectPicker. Ce dernier propose les mêmes gestionnaires de signal que MouseArea, qui correspondent aux interactions avec cet objet. Par exemple, pour implémenter le changement de volume par sélection d'objet (au lieu de la touche Tabulation, comme précédemment), le code est extrêmement simple (ici, pour le tore) :

```
Entity {
    components: [
        TorusMesh { … },
        PhongMaterial { … },
        Transform {
            // …
            scale: root.sphereFocus ? 1 : 2
            // …
        },
        ObjectPicker {
            onClicked: root.sphereFocus = false
        }
    ]
}
```

La propriété `sphereFocus` est définie au niveau global (l'entité racine de la scène, `root`) : à la différence du clavier, la notion de focus n'a pas de sens avec la souris (elle est remplacée par la position du pointeur de la souris, qui indique l'élément qui doit recevoir les événements, chose qui n'existe pas avec le clavier). L'élément choisi doit donc être stocké d'une autre manière.

Figure 24.8 : *Interaction avec la souris : sélectionner un objet le fait grossir (vidéo)*



3.3. Déplacement de la caméra

Qt 3D propose un contrôleur qui implémente toute une série d'interactions avec la souris et le clavier : OrbitCameraController. Son principe est de faire tourner la caméra autour d'un objet à visualiser (la caméra est "en orbite", d'où le nom). Son utilisation est extrêmement simple : il suffit d'ajouter ce composant Qt Quick au niveau de l'entité racine et de faire référence à la caméra à contrôler de la sorte. En déplaçant la souris avec l'un ou l'autre bouton enfoncé, on contrôlera la vue *autour* de l'objet.

```
Entity {
    id: root

    components: [
        RenderSettings { … },
        InputSettings { … }
    ]

    OrbitCameraController {
        camera: camera
    }

    // …
}
```

D'autres contrôleurs sont déjà définis. Par exemple, dans un jeu, on préférera faire avancer le personnage dans un monde (plutôt que le faire tourner autour d'un objet). Dans ce cas, on peut utiliser FirstPersonCameraController, qui fonctionne selon le même principe qu'OrbitCameraController pour l'implémentation. En déplaçant la souris avec le bouton gauche enfoncé, on déplacera la caméra *sur un plan* vertical.

```
Entity {
    id: root

    components: [
        RenderSettings { … },
        InputSettings { … }
    ]

    FirstPersonCameraController {
        camera: camera
    }

    // …
```

```
}
```

Sans recourir à ces contrôleurs tout faits, on peut aussi utiliser les méthodes de Camera, qui permettent de déplacer (`translate`), de faire tourner (`rotate`, l'argument pouvant être obtenu par `rotation`) ou encore d'incliner (`tilt`) la caméra manuellement.

Comparaison de JavaScript et de Python

Cette annexe compare des exemples de code JavaScript et Python dans des utilisations très courantes. Ces tableaux sont prévus pour servir de référence au quotidien.

Tableau .2 : Expressions : comparaison de JavaScript et de Python

	JavaScript	Python
Arithmétique de base	`a + b - c * d / e`	`a + b - c * d / e`
Exponentiation	`Math.pow(a, b)`	`5 ** 2`
Valeurs booléennes	`true` et `false`	`True` et `False`
Opérateurs de comparaison	`a >= b`	`2 >= 4`
Connecteurs logiques	`a && b`, `a || c`, `! a`	`a and b`, `a or c`, `not a`
Opérateur ternaire	`a ? b : c`	`b if a else c`
Conversion de chaîne de caractères en nombre entier	`parseInt(s)`	`int(s)`
Conversion de chaîne de caractères en nombre à virgule flottante	`parseFloat(s)`	`float(s)`
Conversion de nombre en chaîne de caractères	`String(a)`	`str(a)`
Interpolation de chaîne de caractères	`"Fichier traité : " + a + " sur " + b`	`'Fichier traité : %i sur %i' % (a, b)`

Tableau .3 : Structures de données Python et JavaScript

	JavaScript	Python
Définition d'un tableau/liste	`[1, 2, 3]`	`[1, 2, 3]`

Définition d'un dictionnaire (JSON)	`{`
	`  "a": 2,`
	`  "b": true,`
	`  "c": "string"`
	`}`
		`{`
		`  "a": 2,`
		`  "b": True,`
		`  "c": "string"`
		`}`

Définition d'une pile (liste d'éléments, gérée selon la politique du *premier arrivé, dernier sorti* — LIFO)	`var stack = ['a', 'b'];`
	`stack.push('c', 'd');`
	`  // ['a', 'b', 'c', 'd']`
	`stack.push('e');`
	`  // ['a', 'b', 'c', 'd', 'e']`
	`stack.pop(); // 'e'`
		`stack = ['a', 'b']`
		`stack.append('c')`
		`stack.append('d')`
		`stack.append('e')`
		`stack.pop() # 'e'`

Définition d'une file (liste d'éléments, gérée selon la politique du *premier arrivé, premier sorti* — FIFO)	`var queue = ['a', 'b'];`
	`queue.unshift('c', 'd');`
	`  // ['c', 'd', 'a', 'b']`
	`queue.unshift('e');`
	`  // ['e', 'c', 'd', 'a', 'b']`
	`queue.pop(); // 'b'`
		`from collections import deque`
		`queue = deque(['a', 'b'])`
		`queue.append('d')`
		`queue.append('c')`
		`queue.append('e')`
		`queue.pop(0) # 'a'`

Tableau .4 : Constructions fréquentes : comparaison de JavaScript et de Python

	JavaScript	**Python**
Affichage de texte en console	`console.log("…")` (spécifique à Qt Quick)	`print("…")`

Condition	`if (condition) {`
	`  // …`
	`}`
		`if condition:`
		`  # …`

	`var array = [1, 2, 3];` `for (var i = 0; i <`	`array = [1, 2, 3]` `for i in`

Boucle sur un tableau	`array.length; i++) {`
	`    var element = array[i];`
	`    // …`
	`}`
	
	Syntaxe alternative avec une fonction anonyme :
	
	`var array = [1, 2, 3];`
	`array.forEach(function (element, index, array) {`
	`    // …`
	`});`
		`range(len(array)):`
		`    element = array[i]`
		`    # …`
		
		Syntaxe alternative, sans compteur explicite (nettement plus courante) :
		
		`array = [1, 2, 3]`
		`for element in array:`
		`    # …`

Boucle sur une série de nombres	`for (var i = 1; i < 5; i++) {`
	`    // …`
	`}`
		`for i in range(1, 5):`
		`    # …`

Cas spécial si la boucle n'a effectué aucune itération	`var found = false;`
	`for (var i = a; i < b; i++) {`
	`    found = true;`
	`    // …`
	`}`
	
	`if (! found) {`
	`    // Aucune itération effectuée.`
	`    // …`
	`}`
		`for i in range(a, b):`
		`    # …`
		`else:`
		`    # Aucune itération effectuée.`
		`    # …`

Compréhension de liste	`a_list.map(function(x) { return f(x); })`	`[f(x) for x in a_list]`
Avec condition	`a_list.filter(function(x) { return g(x); })` `      .map(function(x) { return f(x); })`	`a = [f(x) for x in a_list if g(x)]`

Index

Symboles

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

V

W

X

Z