Université de la Rochelle - Licence 3 Informatique
Programmation Évènementielle

TP 2 - Paint Event, événement souris et graphisme

© B. Besserer, R. Péteri, S. Bourbia

Année universitaire 2025-2026

1 Fenêtre principale avec zone de dessin

1.1 Événement de type Paint

On reprend la création d'une fenêtre principale en utilisant une écriture orientée objet comme suggéré au TP1. Une classe correspondant à la fenêtre principale est dérivé de la classe QWidget. Dans la zone "vide" de ce widget (souvenet appelé zone "client") nous allons effectuer des opérations graphiques.

Le code de base est le suivant (`TP2.py` sous moodle)

```
from PyQt5.QtWidgets import *
from PyQt5.QtGui import *
from PyQt5.QtCore import Qt

class myMainWindow(QWidget):
    def __init__(self, parent=None):
        QWidget.__init__(self, parent=parent)

        # attributs de la fenetre principale
        self.setGeometry(300,300,800,600)
        self.setWindowTitle('My main window')

        self.initUI()    # appel d'une méthode dédiée à la création de l'IHM

    def initUI(self):
        # code ci-dessous pour remplir... peut-être effacé pour la suite du TP
        w = self.width()
        h = self.height()
        print(str(w)+","+str(h))

if __name__ == '__main__':
    app = QApplication(sys.argv)
    w = myMainWindow()
    w.show()
    app.exec_()
```

Il faut un objet QPainter (contexte de dessin) pour effectuer des tracés graphiques. Lors du TP1, nous avons procédé différement en effectuant les opérations graphiques d'abord dans un objet en mémoire (Pixmap) avant d'attacher cette Pixmap à un widget QLabel. Maintenant, nous allons accéder directement au contexte QPainter d'un widget, à condition que celui-ci **reçoive un message de type Paint** ("rafraîchissement" de la zone graphique) et il faut écrire du code en réaction à ce message.
Pour cela, nous allons définir une méthode paintEvent pour notre classe myMainWindow :

```
def paintEvent(self, event):
  ... votre code ...  # mettre ici le code ne nécessitant pas de contexte graphique (le cas échéant)
  qp = QPainter()     # suite à un événement paint il est possible de récupérer l'objet contexte graphique
  qp.begin(self)
  ... votre code ...  # mettre ici le code qui effectue un tracé graphique (qui utilise qp)
  qp.end()            # fin tracé graphique
```

1. Pour visualiser et compter les événements Paint que votre fenêtre principale reçoit depuis son apparition, écrivez un code qui affiche sur la **console** "Paint event N°" suivi de la valeur du comptage. Il faut donc incrémenter une variable à chaque appel de la méthode paintEvent et l'afficher. Attention, pensez à la portée de la variable (même s'il est possible de définir des variables globales sous Python, on vous le déconseille ;

définissez plutôt une donnée membre de votre classe). Vérifiez si le comptage des événements est effectif : lorsque votre fenêtre était cachée et redevient visible, ou bien en icône et redevient visible, ou lorsque vous redimensionnez la fenêtre, etc...

2. Ce texte ("Paint event N°") ne sera plus seulement affiché dans la console, mais aussi "écrit" (plus exactement dessiné) au milieu de la fenêtre principale. Il faut utiliser une méthode drawText() de l'objet QPainter. Cette méthode dispose d'un paramètre définissant la position spatiale du texte, par exemple le centrage dans une zone rectangulaire :

```
qp.drawText(QRect, Qt.AlignCenter, str)
```

Modifier votre code pour écrire votre ligne de texte au milieu de la fenêtre principale.

Il y a plusieurs façon d'obtenir la zone rectangulaire correspondant à la zone client de votre fenêtre principale, mais de toute façon, il faut que le texte reste centré en cas de redimensionnement de la fenêtre.

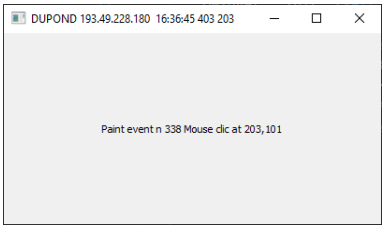
3. Ensuite, récupérez la position d'un clic souris. Pour cela définissez la méthode :

```
def mouseReleaseEvent(self, event):
  p = event.pos()         # ici, par exemple, on recupere la position du clic en tant que QPoint
  self.x = event.x()      # OU ici en tant que valeur entière
  self.y = event.y()
```

Il faut bien sûr définir `self.x` et `self.y` comme données membre de la classe. Modifiez votre affichage pour ajouter l'affichage des coordonnées du clic. Pour information, vous pouvez depuis votre code provoquer un "rafraîchissement" de la fenêtre ou d'un widget avec la méthode `update()`, c'est-à-dire provoquer un événement de type Paint sans que l'utilisateur ait manipulé la fenêtre ou le widget. **Mais attention, il ne faut en aucun cas appeler la méthode `update()` dans la méthode `paintEvent(self, event)`, sinon c'est un bouclage sans fin...**

Pour valider, vous montrerez à l'enseignant :
- la fenêtre immédiatement après le lancement : nombre d'événements de type Paint au minimum, aucun clic souris mémorisé et taille de la fenêtre telle que définie dans le constructeur de la classe (800×400).
- puis après redimensionnement de moitié (**approximativement**) et en ayant cliqué au centre de cette fenêtre (**approximativement**, voir exemples). Évidemment, le compteur d'événements de type Paint sera quelconque, selon vos actions.

1



1.2 Tracé graphique

On va créer et placer deux boutons sur le fond de la fenêtre principale. Le code pour la création de ces boutons sera placé dans la méthode `initUI` qui regroupera tout le code de création et d'initialisation de l'interface utilisateur (initUI = *init User Interface*). **N'utilisez pas** de gestionnaire de géométrie genre VBoxLayout, il faudra donc préciser le widget parent du bouton : `bu1 = QPushButton("Dessine", self)` et donner l'emplacement et la taille des boutons (voir TP1 pour la création et la façon d'attacher à une méthode telle ou telle action effectuée sur le bouton).

Vous placerez les boutons en haut à gauche, consultez la copie d'écran du point de validation 2. Le premier bouton (nommé Dessine) doit servir à valider le dessin d'une figure géométrique sur le fond de votre fenêtre, le second bouton (nommé Efface) permettra d'effacer cette figure. La présence ou non du tracé devra être conservée lorsque par exemple l'application est mise en icône puis restaurée, idéalement définir une variable booléene `doPaint`, membre de votre classe, qui mémorise cet état et qui sera modifiée par les actions sur Dessine et Efface. Lorsqu'un evenement de type Paint doit être traité, le dessin de la grille sera effectif selon la valeur de cette variable booléenne

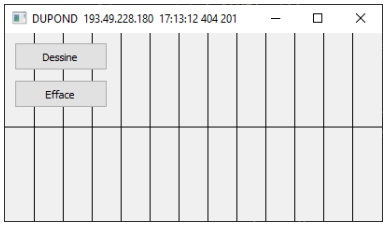
(pour info : effacer = aucune action de dessin entre `qp.begin()` et `qp.end()`). Les boutons Dessine et Efface seront évidemment toujours affichés. Le texte au centre de la fenêtre du 1.1 disparait mais la taille de la fenetre sera toujours affichée dans la barre de titre.

Pour tracer la grille :
- Récupérer la géométrie (h,w) de la zone client et dessinez une ligne horizontale qui sépare la zone client en deux (méthode drawLine)
- Tracez 12 lignes verticales (la largeur de la fenêtre est donc divisé en 13 colonnes égales.
- Les actions Dessine et Efface doivent aussi se déclencher par l'appui sur la touche D (pour Dessine) et l'appui sur la touche E (pour Efface). La méthode qui traite les événements de frappe clavier est la suivante :

```
def keyPressEvent(self, event):
  if event.key() == Qt.Key_D:
    ... votre code ....   # code à executer si la touche est D (majuscule ou minuscule)
  ... votre code ...
```

N'oubliez pas de forcer le rafraîchissement de la fenêtre après une action sur une touche clavier.

Pour valider, il faudra montrer : 2
- la fenêtre immédiatement après le lancement, sans la grille, à la taille définie dans le constructeur de la classe (800×400).
- la fenêtre après redimensionnement de moitié (**approximativement**) et avec le tracé de la grille



2 Grille alphabétique et détection de la lettre

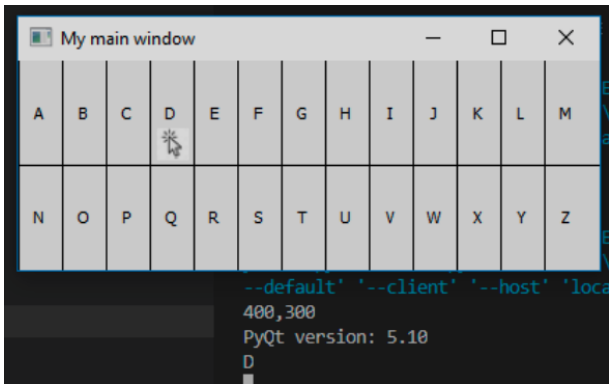
Sur la base de ce tracé de grille, vous devez maintenant dessiner les lettres de l'alphabet (obligatoirement en majuscule) dans chaque case.

– Vous pourrez utiliser le fait que les valeurs du code ASCII des lettres sont consécutives. L'instruction `toto = ord('A')` permet de récupérer le code ASCII de la lettre A, si vous faites `toto = toto+1`, la variable toto contiendra le code ASCII du ''B'. Si vous faites `print(chr(toto))`, la lettre B s'affichera sur la console.

Mais avant de commencer à coder le dessin de votre grille, sachez que la suite de l'exercice, on vous demandera de retrouver la case dans laquelle un clic souris a eu lieu (et donc la lettre correspondante)

Vous pouvez supprimer les boutons Dessine et Efface mais gardez les actions sur les touches D et E. Lorsque votre tracé (grille avec les caractères) est fonctionnelle y compris au niveau du redimensionnement (Remarque : on pourra bloquer la taille minimale de la fenêtre avec `setMinimumSize(QSize(400,200))`), il faut que la récupération du caractère cliqué fonctionne également. Pour cela, après avoir récupéré la position X,Y du clic, il faut déterminer dans quelle case le clic à eu lieu, et en récupérer la lettre (là aussi, plusieurs façon de faire : des tests dans une boucle ou l'utilisation des coordonnées pour calculer un index, qui via un tableau permettra de récupérer la lettre, ou encore l'utilisation du code ASCII).

La lettre en question sera affiché sur la console, voir ci-dessous. Faite valider le fonctionnement.



3

3 Lien avec une seconde fenêtre

La grille va servir de zone de saisie et piloter l'affichage d'une page Web affichée dans une seconde fenêtre. Attention, pour réaliser cette partie qui utilise un widget QtWebEngineWidgets.QWebEngineView qui correspond à un widget de rendu HTML+CSS+Javascript, il faut vérifier :
- Que vous utilisez une version de python >3.5
- Que `PyQtWebEngine` est installé (sinon: `pip3 install PyQtWebEngine`).
Pour les VMs de l'Université, il faudra executer: `pip3 install - -upgrade pip`, puis `.local/bin/pip3 install PyQtWebEngine`.
- Si vous êtes sous Windows avec une version 32 bits de python+pyQt, avec une version pyQt <5.11, il faut ajouter la ligne `from PyQt5 import QtWebEngineWidgets`
- Si vous êtes sur Windows 64 bits avec une version récente de Python 64 bits, il faut installer le module PyQtWebEngine, et dans ce cas ajouter la ligne `from PyQt5.QtWebEngineWidgets import QWebEngineView`.

En cas de problème, consulter l'intervenant. Vous pouvez depuis votre programme afficher la version de pyQt, pour cela, ajoutez : `from PyQt5.Qt import PYQT_VERSION_STR` et dans le code vous pouvez afficher la version sur la console avec `print("PyQt version:", PYQT_VERSION_STR)`

Ce point étant vérifié, définissez une seconde classe de fenêtre :

```
class myBrowserWindow(QWidget):
  ... votre code ....
```

Cette seconde classe disposera d'un widget fils de type "navigateur" : `self.myBrowserWidget = QWebEngineView(self)`, occupant tout l'espace (on peut ici donner une taille fixe avec la methode `SetFixedSize(800, 600)`

Depuis votre programme affichant la grille, créer et afficher une instance de la classe myBrowserWindow provoque l'ouverture d'une seconde fenêtre intégrant ce widget, qui va afficher une page donnant la liste des Pokemons par ordre alphabétique. On charge une page en utilisant la méthode `load()` de ce widget :

```
url = 'https://bulbapedia.bulbagarden.net/wiki/List_of_Pokémon_by_name'
self.myBrowserWidget.load(QUrl(url))
```

Pour cette seconde classe, vous allez également implémenter une méthode, que l'on va appeler `changeLetter`. En effet, il faut "relier" les deux fenêtres : en cliquant dans la grille, on détermine une lettre. Cette lettre sera envoyée à la seconde fenêtre, qui va modifier son adresse URL et actualiser sa page. Par exemple, si l'on clique dans la case D, le caractère D sera transmis en tant que paramètre via un signal (au sens Qt) à la méthode `changeLetter(param)` qui va effectuer la concaténation :

```
def changeLetter(self,letter):
  newurl = url + '#' + param
  ... suite du code ....
```

de manière à former l'adresse : (voir la fin de la ligne, avec la balise #D)
`https://bulbapedia.bulbagarden.net/wiki/List_of_Pokémon_by_name#D`, et on effectue un load avec cette nouvelle adresse.

Il faut donc aussi définir un signal pour la classe myMainWidget (celle qui trace la grille):

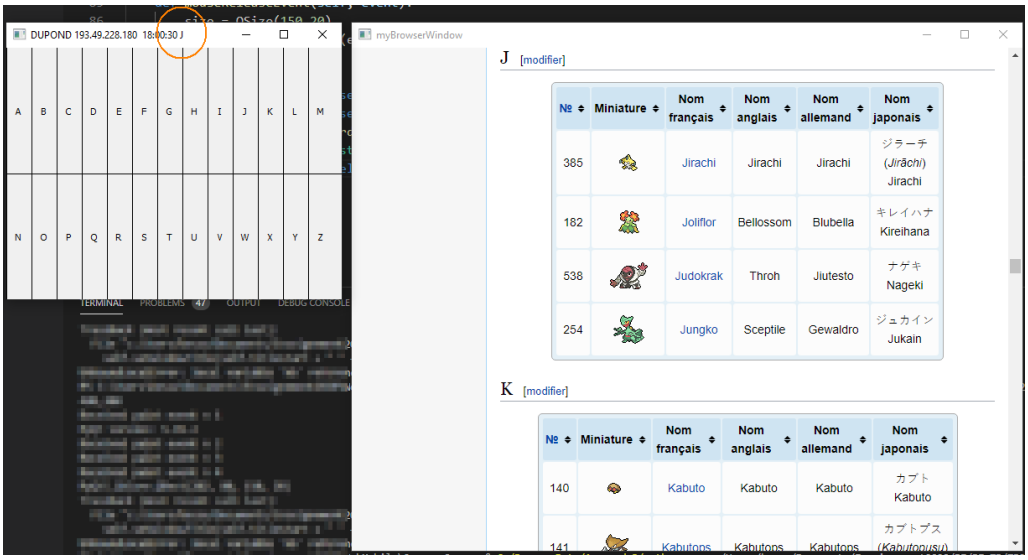
```
sendLetter = pyqtSignal(str) # le signal se nomme sendLetter et transmet une chaine de caractères
                    }
```

pour émettre le signal (lorsque la lettre correspondant au click a été déterminée) :

```
self.sendLetter.emit(message)    # message etant la chaine de caratère à transmettre
```

Et il faut finir par créer le lien entre le signal emis par la fenêtre de classe MyMainWidget et une méthode de la fenêtre de classe myBrowserWindow. On écrit donc : `w.sendLetter.connect(b.changeLetter)`
Ici les instances de la classe myMainWidget et de la classe myBrowserWindow se nomment respectivement w et b.
Pour valider ce point, montrer à l'enseignant le comportement de votre programme avec les deux fenêtres l'une à coté de l'autre, en ayant préalablement cliqué dans une lettre de votre choix (pour laquelle il existe des Pokemons dont les noms commencent avec cette lettre), voir exemple.

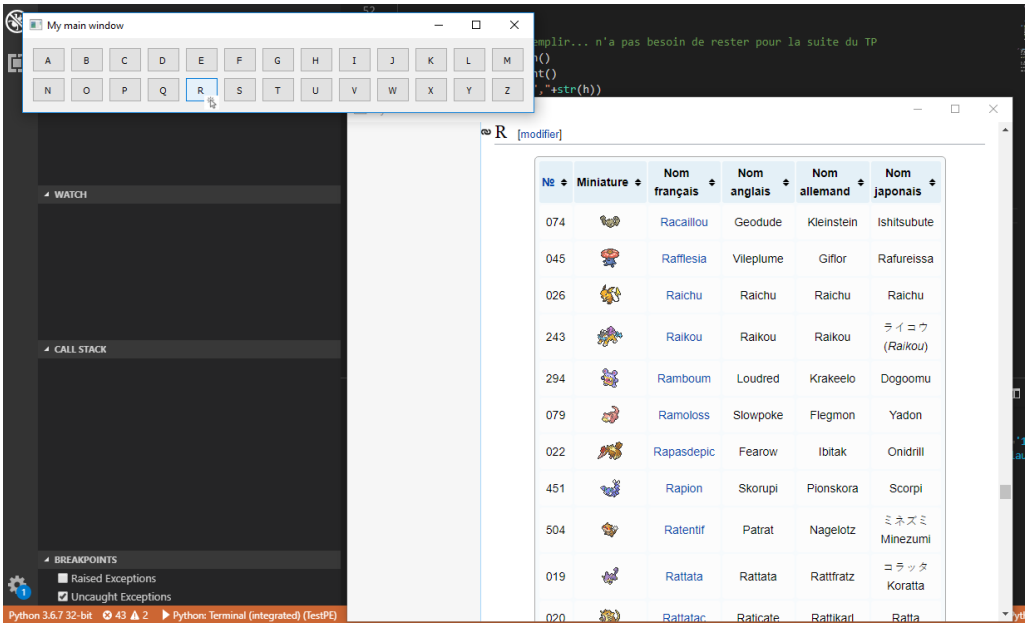
4



3.1 Version avec gestionnaire de géométrie

Au lieu de tracer une grille, placez un gestionnaire de géométrie de type QGridLayout et créez 26 boutons avec les 26 lettres en guise de texte dans ces boutons. Lors de la création de ces boutons, associez l'événement click de chaque bouton à la même méthode, nommée par exemple getLetterFromClick. Dans cette méthode, vous pouvez récupérer le texte (donc la lettre) pour ensuite transmettre celle-ci via votre signal :

```
def getLetterFromClick(self):
  print (self.sender().text())
```



5