PoP Série 6

Heure 1 : Usage de GTKmm 4 pour l'interface graphique utilisateur : événements du clavier et de la souris

Heure 2 : Faire le tutoriel puis les exercices de la semaine 6 du MOOC

Série semaine 6 : Héritage multiple (disponible sur Moodle)

H1 - Usage de GTKmm 4 pour l'interface graphique utilisateur

Code source : Événements du clavier et de la souris

Ces deux exercices réutilisent l'exemple 1.1 Layout de la semaine dernière avec bouton et dessin. Seules les nouveautés par rapport à cette référence sont détaillées ci-dessous.

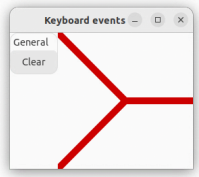
Exercice 1 (niveau 0) : **Usage d'événements provenant des touches du clavier**

Cet exercice introduit la gestion d'événements provenant du clavier, montrant également la possibilité de changer le texte d'un bouton, ce qui permet d'utiliser un seul bouton au lieu de deux. Les événements du clavier sont utilisés pour modifier l'état du bouton et d'un paramètre du dessin.

Exercice 2 (niveau 0) : **Usage d'événements provenant de la souris**

Cet exercice montre comment utiliser le changement d'état de deux boutons de la souris (gauche, droit) et utilise aussi les déplacements de la souris. Les coordonnées du point central sont mises à jour avec les coordonnées obtenues grâce aux changements de position de la souris.

Exercice 1 (niveau 0) : Usage d'événements provenant des touches du clavier



Cet exercice modifie l'exercice « Layout » de la semaine précédente qui présentait une interface graphique avec deux boutons et une surface de dessin. On y introduit la gestion d'événements provenant du clavier, plus particulièrement les touches `'c'`, `'d'`, `'w'` et `'q'`. Comme pour les boutons, on peut effectuer une grande variété d'actions grâce à des variables d'état.

1.1 Examinons les éléments de l'interface de la classe (`.h`) :

```
 6  class MyEvent : public Gtk::Window
 7  {
 8  public:
 9      MyEvent();
10
11  private:
```

```
12      // GUI layout
13      Gtk::Box m_Main_Box;
14      Gtk::Box m_Panel_Box;
15      Gtk::Box m_Buttons_Box;
16      Gtk::Frame m_Panel_Frame;
17      Gtk::Button m_Button_Draw_Clear;
18      Gtk::DrawingArea m_Area;
19
20      bool draw ; // current drawing state
21      bool thick_linewidth ; // another state variable
22
23      //Button Signal handlers:
24      void on_button_clicked_draw_clear();
25
26      // DrawingArea signal handler:
27      void on_draw(const Cairo::RefPtr<Cairo::Context>& cr,
28                   int width, int height);
29
30      // keyboard event signal handler:
31      bool on_window_key_pressed(guint keyval, guint keycode,
32                                 Gdk::ModifierType state);
33  };
```

On remarque qu'il n'y a plus qu'un seul bouton, `m_Button_Draw_Clear`, qui peut effectuer les 2 actions de l'exemple précédent (effacer / dessiner) en agissant sur la variable d'état `draw` — via ce bouton ou via les touches `'c'` ou `'d'`. Une seconde variable d'état `thick_linewidth` sera modifiée par la touche `'w'` et utilisée par le signal handler `on_draw`. Pour pouvoir les utiliser, il faut déclarer le signal handler du clavier (lignes 31–32).

1.2 Dans l'implémentation (`.cc`), le constructeur ajoute les lignes 37–41 pour travailler avec le clavier. On y reconnaît, à la ligne 40, notre signal handler `on_window_key_pressed` :

```
37      // Handling Keyboard Events.
38      auto controller = Gtk::EventControllerKey::create();
39      controller->signal_key_pressed().connect(
40                 sigc::mem_fun(*this, &MyEvent::on_window_key_pressed), false);
41      add_controller(controller);
```

1.3 L'utilisation de la variable d'état `thick_linewidth` dans le signal handler `on_draw` se résume à modifier la largeur du trait avant de dessiner les lignes rouges :

```
62          if(thick_linewidth) cr->set_line_width(10.0);
63          else cr->set_line_width(2.0);
```

1.4 La diminution du nombre de boutons est rendue possible grâce au changement du label du bouton, ce qui permet de toujours afficher l'action possible à tout moment. En effet, chaque action inverse systématiquement l'état de la variable `draw` (ligne 47) et change le label du bouton avec la méthode `set_label` (lignes 48–49). On doit faire un appel à `queue_draw` (ligne 50) pour demander un rafraîchissement de la fenêtre :

```
44  void MyEvent::on_button_clicked_draw_clear()
45  {
46      std::cout << "changing drawing state through the button" << std::endl;
47      draw = !draw;
48      if(draw) m_Button_Draw_Clear.set_label("CLEAR!");
49      else m_Button_Draw_Clear.set_label("DRAW !");
50      m_Area.queue_draw();
51  }
```

Ce que nous avons fait avec le signal handler du bouton, nous pouvons aussi le faire avec le signal handler du clavier pour les touches `'c'` et `'d'` (lignes 84–91) :

```
80  bool MyEvent::on_window_key_pressed(guint keyval, guint, Gdk::ModifierType state)
81  {
82      switch(gdk_keyval_to_unicode(keyval))
83      {
84        case 'c':
```

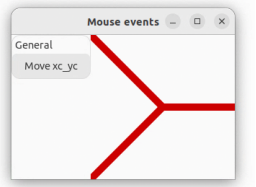
```
85        case 'd':
86          std::cout << "changing drawing state through the keyboard" << std::endl;
87          draw = !draw;
88          if(draw) m_Button_Draw_Clear.set_label("CLEAR!");
89          else m_Button_Draw_Clear.set_label("DRAW !");
90          m_Area.queue_draw();
91          return true;
92        case 'w':
93          std::cout << " changing state variable thick_linewidth" << std::endl;
94          thick_linewidth = !thick_linewidth;
95          m_Area.queue_draw();
96          return true;
97        case 'q':
98          std::cout << "Quit" << std::endl;
99          hide(); // equivalent to exit(0)
100         return true;
101     }
102     // the event has not been handled
103     return false;
104 }
```

On remarque le traitement de la touche `'w'` (lignes 92–96) qui inverse l'état de la variable `thick_linewidth` à chaque appel et qui demande un nouvel affichage. Enfin, le traitement de la touche `'q'` (lignes 97–100) fait quitter le programme avec un appel à `hide()`, ce qui est équivalent à un appel à `exit(0)`.

Activité : Ajouter d'autres réactions liées à d'autres touches du clavier, en ajoutant éventuellement une variable d'état pour mémoriser l'état courant.

Exercice 2 (niveau 0) : Usage d'événements provenant de la souris

Cet exercice utilise aussi un seul bouton pour changer une variable d'état `move_xc_yc`. Celle-ci active ou désactive l'usage des événements de déplacement de la souris. On ajoute également comme variables d'état pour le dessin les coordonnées `xc` et `yc` du point où se rejoignent les lignes rouges.



2.1 Examinons les éléments de l'interface de la classe (`.h`) :

```
6  class MyEvent : public Gtk::Window
7  {
8  public:
9      MyEvent();
10
11 private:
12     // GUI layout
13     Gtk::Box m_Main_Box;
14     Gtk::Box m_Panel_Box;
15     Gtk::Box m_Buttons_Box;
16     Gtk::Frame m_Panel_Frame;
17     Gtk::Button m_Button_Move_xc_yc;
18     Gtk::DrawingArea m_Area;
19
20     // used to enable using mouse move data
21     bool move_xc_yc ;
22     int xc, yc; // drawing parameter
23
24     //Button Signal handlers:
25     void on_button_clicked_move_xc_yc();
26
```

```
27      // DrawingArea signal handler:
28      void on_draw(const Cairo::RefPtr<Cairo::Context>& cr,
29                   int width, int height);
30
31      // mouse event signal handlers:
32      void set_mouse_controller();
33      void on_drawing_left_click(int n_press, double x, double y);
34      void on_drawing_right_click(int n_press, double x, double y);
35      void on_drawing_move(double x, double y);
36  };
```

Les variables d'état sont rassemblées aux lignes 21–22. Les signal handlers des boutons de souris et du mouvement de souris se trouvent à la fin (lignes 33–35). La méthode `set_mouse_controller` (ligne 32) met en œuvre le principe d'abstraction pour rendre le constructeur plus lisible : elle regroupe les initialisations liées à la souris.

2.2 Dans l'implémentation (`.cc`), le constructeur initialise la variable d'état `move_xc_yc` à `false` (ligne 15) et les coordonnées `(xc, yc)` au milieu de la surface de dessin (ligne 37). Il se termine par l'appel de la méthode `set_mouse_controller` qui initialise l'usage des boutons gauche et droit de la souris ainsi que du mouvement de la souris (noter l'appel sur l'attribut de la zone de dessin, lignes 82–84) :

```
66  void MyEvent::set_mouse_controller()
67  {
68      auto left_click = Gtk::GestureClick::create();
69      auto right_click = Gtk::GestureClick::create();
70      auto move = Gtk::EventControllerMotion::create();
71
72      left_click->set_button(GDK_BUTTON_PRIMARY);
73      right_click->set_button(GDK_BUTTON_SECONDARY);
74
75      left_click->signal_pressed().connect(
76          sigc::mem_fun(*this, &MyEvent::on_drawing_left_click));
77      right_click->signal_pressed().connect(
78          sigc::mem_fun(*this, &MyEvent::on_drawing_right_click));
79      move->signal_motion().connect(
80          sigc::mem_fun(*this, &MyEvent::on_drawing_move));
81
82      m_Area.add_controller(left_click);
83      m_Area.add_controller(right_click);
84      m_Area.add_controller(move);
85  }
```

2.3 Le fonctionnement du signal handler du bouton `m_Button_Move_xc_yc` est le même que pour l'exercice précédent. Le signal handler de dessin `on_draw` a été simplifié puisqu'il n'y a plus de variable d'état `draw` : le dessin est toujours effectué.

Examinons maintenant les signal handlers des boutons gauche et droit de la souris. Dans cet exemple, on leur a assigné des tâches très différentes :

— Le bouton gauche récupère les coordonnées de la souris (exprimées dans l'espace de dessin) et les affecte aux attributs `xc` et `yc`. La conséquence, lorsque l'événement créé par `queue_draw` sera traité, sera de déplacer le point de rencontre des lignes rouges à l'endroit où se trouve le curseur :

```
87  void MyEvent::on_drawing_left_click(int n_press, double x, double y)
88  {
89      cout << "mouse left click detected and processed" << endl;
90      cout << "x = " << x << " y = " << y << endl;
91      xc = x;
92      yc = y;
93      m_Area.queue_draw();
94  }
```

— Le bouton droit a la même action que le bouton `m_Button_Move_xc_yc` :

```
96  void MyEvent::on_drawing_right_click(int n_press, double x, double y)
97  {
```

```
 98      cout << "mouse right click detected and processed" << endl;
 99      move_xc_yc = !move_xc_yc;
100      if(move_xc_yc) m_Button_Move_xc_yc.set_label("Do Not Move!");
101      else m_Button_Move_xc_yc.set_label("Move (xc,yc)");
102      m_Area.queue_draw();
103  }
```

2.5 Le signal handler des mouvements de la souris est indépendant de l'usage des boutons de la souris. Il est automatiquement appelé dès que la souris bouge légèrement. Ici on prend en compte la position du curseur si la variable d'état `move_xc_yc` est `true`. Sinon, on recentre le point `(xc, yc)` au milieu de la surface de dessin :

```
105  void MyEvent::on_drawing_move(double x, double y)
106  {
107      static char c('A');
108      cout << c << " => mouse move detected and processed"
             << endl;
109      if(++c > 'Z') c = 'A'; // reset
110
111      if(move_xc_yc)
112      {
113          if(x > 0 and x < m_Area.get_width())
114              xc = x;
115
116          if(y > 0 and y < m_Area.get_height())
117              yc = y;
118      }
119      else
120      {
121          xc = m_Area.get_width() / 2;
122          yc = m_Area.get_height() / 2;
123      }
124      m_Area.queue_draw();
125  }
```

```
A => mouse move detected and processed
B => mouse move detected and processed
C => mouse move detected and processed
D => mouse move detected and processed
E => mouse move detected and processed
F => mouse move detected and processed
G => mouse move detected and processed
H => mouse move detected and processed
mouse left clic detected and processed
x = 73   y = 96
mouse right clic detected and processed
I => mouse move detected and processed
J => mouse move detected and processed
K => mouse move detected and processed
```

Activité : Modifier les signal handlers de la souris pour explorer d'autres comportements.