PoP C++ Série 7 niveau 0

Usage de GTKmm :
Fenêtre de dialogue / event du clavier et souris / Timer

Adapté du manuel de référence en-ligne de GTKmm 3

Exercice 1.(niveau 0) : **fenêtre de dialogue pour choisir un nom de fichier**

Cet exercice simplifie l'exemple FileChooserDialog fourni par le manuel GTKmm 3. Par contre on a ajouté un Gtk ::Label dont on change le texte selon l'interaction avec la fenêtre de dialogue. Le programme principal définit un widget ExampleWindow dérivé de la classe Window

```
#include "examplewindow.h"
#include <gtkmm/application.h>

int main(int argc, char *argv[])
{
  auto app = Gtk::Application::create(argc, argv, "org.gtkmm.example");

  ExampleWindow window;

  //Shows the window and returns when it is closed.
  return app->run(window);
}
```

L'interface du module **exampleWindow** définit un ButtonBox, un Button avec son signal handler, et un Label :

```
#ifndef GTKMM_EXAMPLEWINDOW_H
#define GTKMM_EXAMPLEWINDOW_H

#include <gtkmm.h>

class ExampleWindow : public Gtk::Window
{
public:
  ExampleWindow();
  virtual ~ExampleWindow();

protected:
  //Signal handlers:
  void on_button_file_clicked();

  //Child widgets:
  Gtk::ButtonBox m_ButtonBox;
  Gtk::Button    m_Button_File;
  Gtk::Label     m_Label_Info;
};

#endif //GTKMM_EXAMPLEWINDOW_H
```

La première partie de l'implémentation contient le constructeur. On dispose le Button et le Label verticalement dans un ButtonBox (équivalent à Box vu dans la série6 niveau 0). On connecte le Button à son signal handler.

```
#include "examplewindow.h"
#include <iostream>


ExampleWindow::ExampleWindow()
: m_ButtonBox(Gtk::ORIENTATION_VERTICAL),
  m_Button_File("Choose File"),
  m_Label_Info("Init")
{
  set_title("Gtk::FileSelection example");

  add(m_ButtonBox);

  m_ButtonBox.pack_start(m_Button_File);
  m_Button_File.signal_clicked().connect(sigc::mem_fun(*this,
              &ExampleWindow::on_button_file_clicked) );

  m_ButtonBox.pack_start(m_Label_Info);

  show_all_children();
}

ExampleWindow::~ExampleWindow()
{
}
```

L'aspect intéressant est le signal handler du Button. Cette méthode commence par déclarer une instance **dialog** de **Gtk ::FileChooserDialog** qui servira de support pour gérer le choix d'un nom de fichier dans une fenêtre temporaire. Cet objet de dialogue dispose d'une méthode **run()** qui va capter l'activité de l'interface, c'est-à-dire qu'on ne pourra rien faire d'autre tant que ce dialogue sera en cours.

Le caractère « temporaire » de l'instance **dialog** est spécifié par l'appel de la méthode **set_transient_for**() qui suit. Cela garantit le placement de la fenêtre de dialogue au-dessus de la fenêtre de l'application.

On doit explicitement demander quels boutons de dialogue on veut : ici on demande un button CANCEL et un bouton OPEN qui confirme le choix.

Les constantes **`Gtk::RESPONSE_CANCEL`** et **`Gtk::RESPONSE_OK`**, fournies en second paramètres, sont testées plus loin dans le code.

Dans cet exemple nous avons encadré l'appel de la méthode **run**() par une modification du Label à l'aide de la méthode **set_text**(). Il suffit de déplacer la fenêtre de dialogue pour constater que le Label a bien changé avant l'appel de **run**().

```
void ExampleWindow::on_button_file_clicked()
{
  Gtk::FileChooserDialog dialog("Please choose a file",
          Gtk::FILE_CHOOSER_ACTION_OPEN);
  dialog.set_transient_for(*this);

  //Add response buttons the the dialog:
  dialog.add_button("_Cancel", Gtk::RESPONSE_CANCEL);
  dialog.add_button("_Open", Gtk::RESPONSE_OK);

  m_Label_Info.set_text("choosing a file");

  //Show the dialog and wait for a user response:
  int result = dialog.run();

  m_Label_Info.set_text("Done choosing a file");

  //Handle the response:
  switch(result)
  {
    case(Gtk::RESPONSE_OK):
    {
      std::cout << "Open clicked." << std::endl;

      //Notice that this is a std::string, not a Glib::ustring.
      std::string filename = dialog.get_filename();
      std::cout << "File selected: " <<  filename << std::endl;
      break;
    }
    case(Gtk::RESPONSE_CANCEL):
    {
      std::cout << "Cancel clicked." << std::endl;
      break;
    }
    default:
    {
      std::cout << "Unexpected button clicked." << std::endl;
      break;
    }
  }
}
```

Il est très important de récupérer dans la variable **result** la valeur entière renvoyée par la méthode **run**() car c'est elle qui pilote la suite du programme avec un switch. On y retrouve les deux constantes utilisées au moment de la création des buttons CANCEL et OPEN.

La constante liée à OPEN est **`Gtk::RESPONSE_OK`.** Attention il n'y a pas eu d'ouverture de fichier à ce stade. Cette réponse veut simplement dire qu'on peut récupérer le nom de fichier choisi. La string renvoyée par **get_filename()** contient le chemin absolu depuis la racine de l'arborescence des fichiers comme on peut le constater quand on l'affiche dans le terminal.

Activité : utiliser le filename pour ouvrir un fichier et l'afficher dans le terminal.

Complément : Comment passer de **OPEN** à **SAVE** ?

Le même objet **Gtk ::FileChooserDialog** sert pour les deux opérations ; il faut seulement le configurer avec un symbole différent selon l'utilisation.

Pour OUVRIR : on indique le symbole **`Gtk::FILE_CHOOSER_ACTION_OPEN`** dans la déclaration du **Gtk ::FileChooserDialog** :

```
Gtk::FileChooserDialog dialog("Please choose a file",
        Gtk::FILE_CHOOSER_ACTION_OPEN);
```

Ensuite, on ajoute le Button de dialogue qui signale l'action **_Open** dans la fenêtre de dialogue :

```
dialog.add_button("_Open", Gtk::RESPONSE_OK);
```

Pour SAUVEGARDER : on indique le symbole **`Gtk::FILE_CHOOSER_ACTION_SAVE`** dans la déclaration du **Gtk ::FileChooserDialog** :

```
Gtk::FileChooserDialog dialog("Please choose a file",
        Gtk::FILE_CHOOSER_ACTION_SAVE);
```

Ensuite, on ajoute le Button de dialogue qui signale l'action **_Save** (au lieu de **_Open**) dans la fenêtre de dialogue :

```
dialog.add_button("_Save", Gtk::RESPONSE_OK);
```

Exercice 2.(niveau 0) : **gestion d'événements clavier et souris**

Vous trouverez les détails des explications sur la gestion d'événements provenant des Buttons dans l'exercice1 de la série6 niveau0. On se concentre ici sur les quelques différences introduites par rapport à cet exercice de référence.

2.1) Le programme principal définit un widget de la classe que nous avons définie. Contrairement à l'exemple de la série6 ici on ne définit pas de taille de fenêtre ; on laisse GTKmm calculer la bonne taille à partir de son contenu.

```
#include "myevents.h"
#include <gtkmm/application.h>
#include <gtkmm/window.h>

int main(int argc, char** argv)
{
   auto app = Gtk::Application::create(argc, argv, "org.gtkmm.example");

   MyEvent eventWindow;
   eventWindow.set_resizable(false);

   return app->run(eventWindow);
}
```

2.2) Passons à l'interface du module **myevents** :

Il contient d'abord un modèle de structure **Point** pour faciliter le dessin à partir d'un ensemble de Points qui est mémorisé dans la classe **MyArea**.

Vient ensuite l'interface des deux classes dont voici les différences par rapport à la série6:

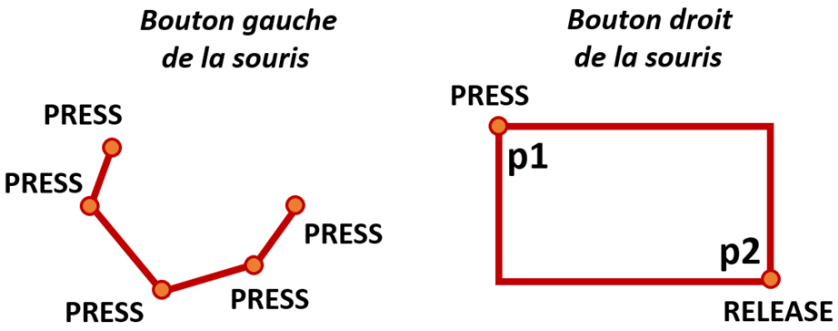
- La classe **MyArea** est étendue avec 4 méthodes publiques :
  - **reset**() : ré-initialise les éléments de dessin ; les Points **p1** et **p2** sont à l'origine et le vector **line** est vidé
  - **reset_rect(Point p) :** le Point p ré-initialise les 2 coins opposés **p1** et **p2** du rectangle avec le Point p
  - **finalize_rect(Point p)** : définit le second coin **p2** du rectangle avec le Point **p**
  - **add_point(Point p) :** ajoute le Point à la fin du vector **line**
- une méthode protected de dessin **draw_frame** : pour visualiser l'espace de **Myarea**
- et 3 attributs privés :
  - 2 Points **p1** et **p2**, et
  - un vector de Points **line**.

- La classe **MyEvent** est étendue avec 2 signal_handlers pour les boutons de la souris :
  - **on_button_press_event(GdkEventButton * event);**
  - **on_button_release_event(GdkEventButton * event);**
- un pour les événements du clavier :
  - **on_key_press_event(GdkEventKey * key_event);**
- Deux des attributs **Gt ::Box** ont été renommés pour avoir un layout principal horizontal au lieu de vertical :

```
#ifndef GTKMM_EXAMPLE_MYEVENTS_H
#define GTKMM_EXAMPLE_MYEVENTS_H

#include <gtkmm.h>

struct Point
{
    double x;
    double y;
};

class MyArea : public Gtk::DrawingArea
{
public:
    MyArea();
    virtual ~MyArea();
    void clear();
    void draw();
    void reset();
    void reset_rect(Point p);
    void finalize_rect(Point p);
    void add_point(Point p);

protected:
    //Override default signal handler:
    bool on_draw(const Cairo::RefPtr<Cairo::Context>& cr) override;
    void draw_frame(const Cairo::RefPtr<Cairo::Context>& cr);

private:
    bool empty;
    Point p1,p2;
    std::vector<Point> line;
    void refresh();
};

//====================================
// includes keyboard and mouse events
class MyEvent : public Gtk::Window
{
public:
    MyEvent();
    virtual ~MyEvent();

protected:
    //Button Signal handlers:
    void on_button_clicked_clear();
    void on_button_clicked_draw();

    // Mouse event signal handlers:
    bool on_button_press_event(GdkEventButton * event);
    bool on_button_release_event(GdkEventButton * event);

    // Keyboard signal handler:
    bool on_key_press_event(GdkEventKey * key_event);

    // interface components
    Gtk::Box m_Box, m_Box_Left, m_Box_Right;
    MyArea          m_Area;
    Gtk::Button     m_Button_Clear;
    Gtk::Button     m_Button_Draw;
};
#endif // GTKMM_EXAMPLE_MYEVENTS_H
```

L'implémentation définit d'abord MyArea.

- Le fonctionnement des méthodes **clear , draw** et **refresh** reste le même (voir série6).
- Les nouvelles méthodes publiques servent à effectuer deux types de dessin selon le bouton (de la souris) qui est utilisé (Fig ci-dessous):



  - **<u>Bouton gauche de la souris</u>** : À chaque clic (PRESS), la méthode **add_point** ajoute un point à l'attribut **line** dont les coordonnées sont données par les coordonnées de la souris. L'attribut **line** est dessiné comme une ***polyline*** dans la méthode on_draw().

  - **<u>Bouton droit de la souris</u>**:
    - Quand on appuie (PRESS) la méthode **reset_rect** re-initialise les points **p1** et **p2** avec la valeur du Point où se trouve le pointeur de la souris.

    - Si on reste dans l'état appuyé en déplaçant la souris et qu'on relache le bouton (RELEASE) la méthode **finalize_rect** affecte **p2** avec la valeur du Point où se trouve le pointeur de la souris au moment où on relache le bouton. Les points p1 et p2 sont utilisés pour dessiner un rectangle dans la méthode on_draw().

- La quatrième méthode **reset** sert à vider le vector **line** et initialiser **p1** et **p2** à (0,0).

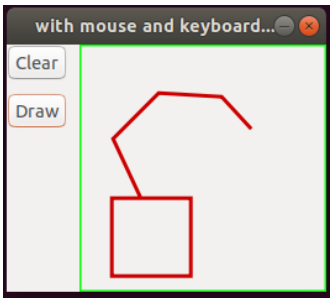
Remarquer dans le code que les 4 nouvelles méthodes appellent toutes la méthode **refresh** pour créer un événement qui va produire une mise à jour du dessin pour visualiser le nouvel affichage graphique. Sans cette demande on ne verrait pas l'effet de l'action des boutons.

La dernière nouveauté de MyArea est la méthode draw_frame qui matérialise l'espace de dessin avec une bordure verte (une autre manière serait de dessiner le fond de la MyArea avec la méthode paint() pour obtenir un fond d'une couleur différente de la colonne où existent les boutons).
C'est une bonne pratique d'effectuer cette visualisation du « cadre » de MyArea pour mieux comprendre ensuite la transformation de coordonnées à effectuer sur les valeurs obtenues avec les signal handlers des boutons de la souris dans l'implémentation de MyEvent.

<u>Activité</u> : utiliser **paint()** pour obtenir un fond blanc pour l'espace de MyArea.

```
#include <iostream>
#include "myevents.h"
#include <cairomm/context.h>

using namespace std;

MyArea::MyArea(): empty(false),p1({0.,0.}),p2({0.,0.})
{
}

MyArea::~MyArea()
{
}

void MyArea::clear()
{
    empty = true;
    refresh();
}

void MyArea::draw()
{
    empty = false;
    refresh();
}

void MyArea::reset()
{
    p1 = {0.,0.};
    p2 = {0.,0.};
    line.clear();
    refresh();
}

void MyArea::reset_rect(Point p)
{
    p1 = p;
    p2 = p;
    refresh();
}

void MyArea::finalize_rect(Point p)
{
    p2 = p;
    refresh();
}

void MyArea::add_point(Point p)
{
    line.push_back(p);
    refresh();
}

void MyArea::refresh()
{
    auto win = get_window();
    if(win)
    {
        Gdk::Rectangle r(0,0, get_allocation().get_width(),
                              get_allocation().get_height());

        win->invalidate_rect(r,false);
    }
}

void MyArea::draw_frame(const Cairo::RefPtr<Cairo::Context>& cr)
{
    //display a rectangular frame around the drawing area
```

```
    cr->set_line_width(3.0);
    // draw green lines
    cr->set_source_rgb(0.0, 1.0, 0.0);
    cr->rectangle(0,0,get_allocation().get_width(),
                      get_allocation().get_height());
    cr->stroke();
}

bool MyArea::on_draw(const Cairo::RefPtr<Cairo::Context>& cr)
{
    draw_frame(cr);

    if(not empty)
    {
         cout << " Drawing what has been defined with mouse events " << endl ;
         cr->set_line_width(3.0);

         // draw red lines
         cr->set_source_rgb(0.8, 0.0, 0.0);

         cr->rectangle(p1.x, p1.y, p2.x-p1.x, p2.y-p1.y);

         if(line.size()>0)
              cr->move_to(line[0].x,line[0].y);

         for(size_t i(1) ; i< line.size() ; ++i)
              cr->line_to(line[i].x,line[i].y);

         cr->stroke();
    }
    else
    {
         cout << "Empty !" << endl;
    }

    return true;
}


//=============================================

MyEvent::MyEvent() :
  m_Box(Gtk::ORIENTATION_HORIZONTAL,10),
  m_Box_Left(Gtk::ORIENTATION_VERTICAL, 10),
  m_Box_Right(Gtk::ORIENTATION_VERTICAL, 10),
  m_Button_Clear("Clear"),
  m_Button_Draw("Draw")
{
    // Set title and border of the window
    set_title("with mouse and keyboard...");
    set_border_width(0);

    // Add outer box to the window (because the window
    // can only contain a single widget)

    add(m_Box);

    //Put the inner boxes and the separator in the outer box:
    m_Box.pack_start(m_Box_Left);
    m_Box.pack_start(m_Box_Right);

    m_Box_Left.pack_start(m_Button_Clear,false,false);// keep fixed width
    m_Box_Left.pack_start(m_Button_Draw,false,false); // and aligned to left;

    m_Area.set_size_request(200,200);
    m_Box_Right.pack_start(m_Area);


    // Connect the clicked signal of the button to
    // their signal handler
```

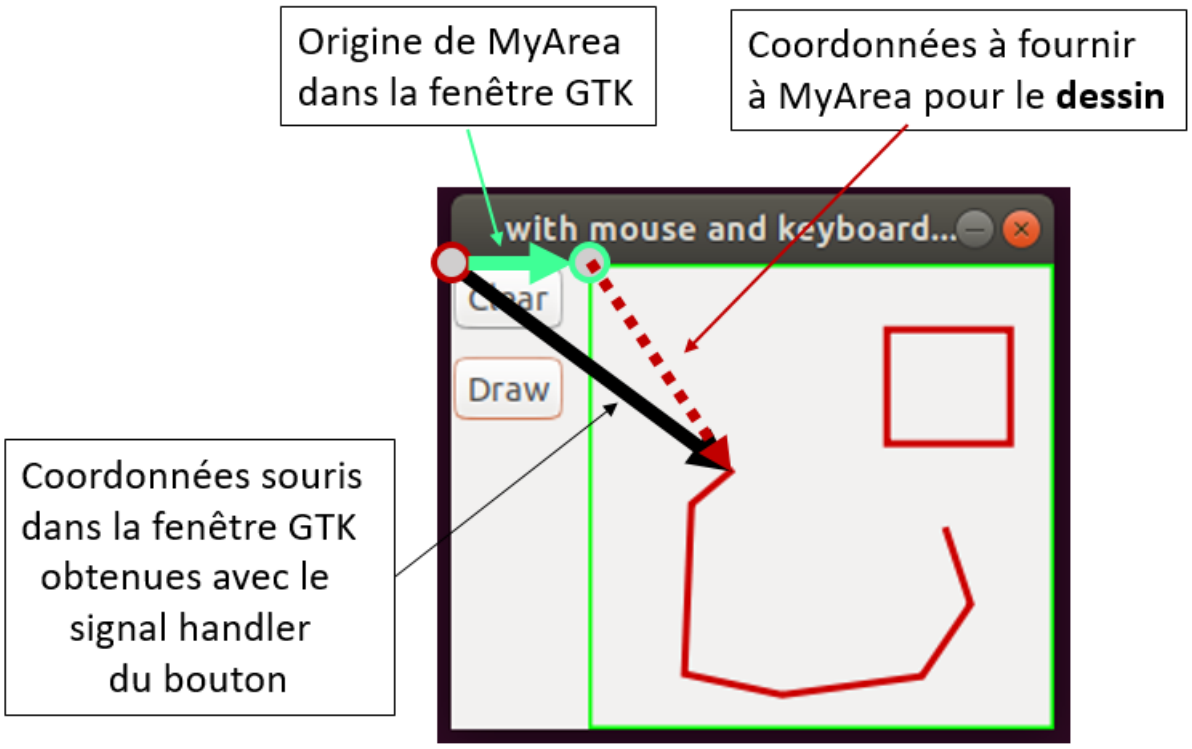


```
    m_Button_Clear.signal_clicked().connect(sigc::mem_fun(*this,
                                &MyEvent::on_button_clicked_clear) );

    m_Button_Draw.signal_clicked().connect(sigc::mem_fun(*this,
                                 &MyEvent::on_button_clicked_draw) );

    // Show all children of the window
    show_all_children();
}

MyEvent::~MyEvent()
{
}

//Button Signal handlers:
void MyEvent::on_button_clicked_clear()
{
    cout << "Clear" << endl;
    m_Area.clear();
}

void MyEvent::on_button_clicked_draw()
{
    cout << "Draw" << endl;
    m_Area.draw();
}

// Mouse event signal handlers:
bool MyEvent::on_button_press_event(GdkEventButton * event)
{
    if(event->type == GDK_BUTTON_PRESS)
    {
         // raw mouse coordinates in the window frame
        double clic_x = event->x ;
        double clic_y = event->y ;

         cout << "mouse x = " << clic_x << "\t mouse y = " << clic_y << endl;

         // origin of the drawing area
         double origin_x = m_Area.get_allocation().get_x();
         double origin_y = m_Area.get_allocation().get_y();

         cout << "m_Area x = " << origin_x << "\t m_Area y = " << origin_y << endl;

         // get width and height of drawing area
         double width = m_Area.get_allocation().get_width();
         double height= m_Area.get_allocation().get_height();

         cout << "width  = " << width << "\t height = " << height << endl;

         // retain only mouse events located within the drawing area
         if(clic_x >= origin_x && clic_x <= origin_x + width &&
            clic_y >= origin_y && clic_y <= origin_y + height)
         {
              // Point that we are allowed to use expressed with drawing area coord.
              Point p({clic_x - origin_x, clic_y -origin_y});

              // there is no symbol to designate mouse buttons...
              if(event->button == 1) // Left mouse button
              {
                  // add one point to the polyline vector
                  m_Area.add_point(p);
              }
              else if(event->button == 3) // Right mouse button
              {
                  m_Area.reset_rect(p);
              }
         }
    }
    return true;
```

```
}

bool MyEvent::on_button_release_event(GdkEventButton * event)
{
    if(event->type == GDK_BUTTON_RELEASE)
    {
        // raw mouse coordinates in the window frame
        double clic_x = event->x ;
        double clic_y = event->y ;

        cout << "mouse x = " << clic_x << "\t mouse y = " << clic_y << endl;

        // origin of the drawing area
        double origin_x = m_Area.get_allocation().get_x();
        double origin_y = m_Area.get_allocation().get_y();

        cout << "m_Area x = " << origin_x << "\t m_Area y = " << origin_y << endl;

        // get width and height of drawing area
        double width = m_Area.get_allocation().get_width();
        double height= m_Area.get_allocation().get_height();

        cout << "width  = " << width << "\t height = " << height << endl;

        // retain only mouse events located within the drawing area
        if(clic_x >= origin_x && clic_x <= origin_x + width &&
           clic_y >= origin_y && clic_y <= origin_y + height)
        {
            // Point that we are allowed to use expressed with drawing area coord.
            Point p({clic_x - origin_x, clic_y -origin_y});

            // there is no symbol to designate mouse buttons...
            if(event->button == 3) // Right mouse button
            {
                m_Area.finalize_rect(p);
            }
        }
    }
    return true;
}

// Keyboard signal handler:
bool MyEvent::on_key_press_event(GdkEventKey * key_event)
{
    if(key_event->type == GDK_KEY_PRESS)
    {
        switch(gdk_keyval_to_unicode(key_event->keyval))
        {
            case 'r':
                cout << "Reset" << endl;
                m_Area.reset();
                break;

            case 'q':
                cout << "Quit" << endl;
                exit(0);
                break;
        }
    }

    return Gtk::Window::on_key_press_event(key_event);
}
```

Comparé à l'agencement de la série6, nous avons choisi ici d'avoir les boutons verticalement dans une première colonne suivie par l'espace de dessin (voir image à coté du constructeur).

La raison est de faire apparaître le contexte où l'origine de la fenêtre GTKmm ne coïncide pas avec l'origine de l'espace du dessin. Cette différence apparaît sous la forme du vecteur vert dans la figure ci-dessous. Cela va imposer une transformation de coordonnées décrite plus loin.



Gestion des événements des boutons de la souris :

Il n'y a rien à initialiser au niveau du constructeur de MyEvent en ce qui concerne les 3 *signal handlers* que nous avons ajoutés à son interface.

Examinons les 2 *signal handlers* associée aux boutons de la souris ; ils se distinguent par le type de l'événement détecté : **PRESS** ou **RELEASE** (et non pas par le bouton utilisé).

Quand on **appuie** sur l'un des 3 boutons de la souris le signal handler PRESS est appelé.

Quand on **relache** l'un des 3 boutons de la souris le signal handler RELEASE est appelé.

Le signal handler a un seul paramètre **event** qui permet d'obtenir les coordonnées (x,y) du pointeur de la souris au moment où le bouton est appuyé (PRESS) ou relaché (RELEASE).

Attention : il s'agit des coordonnées « fenêtre» indiquées avec le vecteur noir ci-dessus.

C'est pourquoi l'attribut MyArea demande le vecteur vert ci-dessus indiquant son origine dans la fenêtre avec les méthodes **get_x** et **get_y** .

Il sera soustrait des coordonnées obtenues pour avoir des coordonnées à utiliser pour le dessin (vecteur pointillé rouge).

Le code source de **`on_button_press_event`** et de **`on_button_release_event`** fait afficher toutes ces valeurs à titre documentaire.

Avant toute action avec les coordonnées calculées, il faut vérifier que le point cliqué est bien *dans l'espace de dessin* correspondant à MyArea afin de pouvoir voir le résultat de l'action. C'est le but du test sur **clic_x** et **clic_y** dans le code.

Après cette validation il faut décider comment utiliser les coordonnées de dessin et pour cela il faut savoir *quel est le bouton appuyé*. Cela se fait en testant le champ **button** accessible avec le paramètre **event** ; on a la correspondance suivante[1] :

gauche = 1, milieu = 2, droit = 3.

Il ne reste plus qu'à appeler les bonnes méthodes de MyArea selon le type d'événement et le bouton utilisé :

- PRESS sur le bouton **gauche** produit un appel de **add_point**
- PRESS sur le bouton **droit** produit un appel de **reset_rect**
- RELEASE sur le bouton **droit** produit un appel de **finalize_rect**

Activité : faire dessiner une ligne en pointillé au lieu d'une ligne brisée continue en utilisant l'event RELEASE du bouton gauche. Un trait plein commence toujours avec l'event PRESS et se termine avec l'event RELEASE. On peut ré-utiliser le même vector line. Il faudra adapter la méthode on_draw pour obtenir l'effet désiré.

Gestion des événements des touches du clavier :

Le dernier *signal handler* à examiner est celui lié au clavier. On obtient un code qu'il faut convertir vers le standard UNICODE avec la méthode **`gdk_keyval_to_unicode`**.

On peut alors utiliser un simple switch avec autant de case que de touches clavier qui nous intéressent. Nous avons retenu '**q'** pour quitter le programme en appelant **exit**(0) et '**r'** comme **reset** pour réinitialiser **p1**, **p2** et **line** dans MyArea.

Activité : ajouter d'autres réactions liées à d'autres touches du clavier (changement incrémental d'épaisseur de ligne, de couleur, …).

[1] La doc GTKmm n'offre pas de symboles pour nommer ces magic numbers.

Exercice 3 (niveau 0) : **contrôle du temps avec un Timer**

Cet exercice simplifie autant que possible l'exemple Timerexample fourni par le manuel GTKmm 3 dans le sens où on gère un seul Timer au lieu d'un nombre quelconque. Le programme principal définit un widget SimpleExampleWindow dérivé de la classe Window

```
#include "basictimer.h"
#include <gtkmm/application.h>

int main (int argc, char *argv[])
{
  auto app = Gtk::Application::create(argc, argv, "org.gtkmm.example");

  BasicTimer example;
  return app->run(example);
}
```

Le but est de faire exécuter une action particulière à intervalle régulier, dans cet exemple toutes les 0,5 s. L'exemple illustre également comment arrêter puis relancer le Timer.

```
#ifndef GTKMM_EXAMPLE_TIMEREXAMPLE_H
#define GTKMM_EXAMPLE_TIMEREXAMPLE_H

#include <gtkmm.h>
#include <iostream>

class BasicTimer : public Gtk::Window
{
public:
  BasicTimer();

protected:
  // button signal handlers
  void on_button_add_timer();
  void on_button_delete_timer();
  void on_button_quit();

  // This is the standard prototype of the Timer callback function
  bool on_timeout();

  // Member data:
  Gtk::Box m_Box;
  Gtk::Button m_ButtonAddTimer, m_ButtonDeleteTimer, m_ButtonQuit;

  // Keep track of the timer status (created or not)
  bool timer_added;

  // to store a timer disconnect request
  bool disconnect;

  // This constant is initialized in the constructor's member initializer:
  const int timeout_value;
};

#endif // GTKMM_EXAMPLE_TIMEREXAMPLE_H
```

Interface du module :

Les actions de lancer puis d'arrêter le Timer sont demandées à l'aide des Buttons **m_ButtonAddTimer** et **m_ButtonDeleteTimer**. Chacun dispose de son signal handler qui va respectivement contrôler ces actions de lancer ou arrêter le Timer.

La Box **m_Box** sert à disposer les Buttons horizontalement.

Le booléen **timer_added** sert à mémoriser si un Timer a déjà été créé pour éviter des actions incorrectes (ex : créer plusieurs Timers ou détruire un Timer qui n'existe pas).

Le booléen **disconnect** va servir de relai pour une demande d'arrêt du Timer.

La *constante* entière **timeout_value** est un nombre de millisecondes qui sera initialisée à la déclaration de l'instance de SimpleTimerExample.

Implémentation du module :

Le constructeur précise les label des Buttons dans la liste d'initialisation (Start, Stop et Quit). On y trouve aussi la valeur de la période du Timer, ici **500ms**. Enfin the Timer n'existe pas au lancement du programmet ; le booléen **timer_added** est initialisé à **false** ainsi que le booléen **disconnect**.

Le constructeur dispose les Buttons dans la Box horizontalement (conformément au paramètre de la liste d'initialisation) et les connectent à leur signal handler.

```
#include "basictimer.h"

BasicTimer::BasicTimer() :
  m_Box(Gtk::ORIENTATION_HORIZONTAL, 10),
  m_ButtonAddTimer("_Start", true),
  m_ButtonDeleteTimer("_Stop", true),
  m_ButtonQuit("_Quit", true),
  timer_added(false),
  disconnect(false),
  timeout_value(500) // 500 ms = 0.5 seconds
{
  set_border_width(10);

  add(m_Box);
  m_Box.pack_start(m_ButtonAddTimer);
  m_Box.pack_start(m_ButtonDeleteTimer);
  m_Box.pack_start(m_ButtonQuit);

  // Connect the three buttons:
  m_ButtonQuit.signal_clicked().connect(sigc::mem_fun(*this,
              &BasicTimer::on_button_quit));
  m_ButtonAddTimer.signal_clicked().connect(sigc::mem_fun(*this,
              &BasicTimer::on_button_add_timer));
  m_ButtonDeleteTimer.signal_clicked().connect(sigc::mem_fun(*this,
              &BasicTimer::on_button_delete_timer));

  show_all_children();
}

void BasicTimer::on_button_add_timer()
{
  if(not timer_added)
  {
        Glib::signal_timeout().connect( sigc::mem_fun(*this,
              &BasicTimer::on_timeout), timeout_value );

        timer_added = true;

        std::cout << "Timer added" << std::endl;
  }
  else
```

```
  {
        std::cout << "The timer already exists: nothing more is created"
                  << std::endl;
  }
}

bool BasicTimer::on_timeout()
{
  static unsigned int val(1);

  if(disconnect)
  {
        disconnect = false; // reset for next time a Timer is created
        return false; // End of Timer
  }

  std::cout << "This is timer value : " << val << std::endl;
  ++val; // tic the clock

  return true; // keep the Timer working
}
void BasicTimer::on_button_delete_timer()
{
  if(not timer_added)
  {
    std::cout << "Sorry, there is no active timer at the moment."
              << std::endl;
  }
  else
  {
    std::cout << "manually disconnecting the timer " << std::endl;
    disconnect  = true;
    timer_added = false;
  }
}

void BasicTimer::on_button_quit()
{
  hide();
}
```

La nouveauté de cet exemple provient des signal handlers des Buttons Add et Delete :

- Pour **m_ButtonAddTimer** le signal handler **on_button_add_timer()** s'assure que le booléen est à false avant de créer le Timer. Celui-ci est associé à une fonction callback **on_timeout()** qui sera automatiquement appelée chaque fois que la période de 500ms est écoulée. Cela fait on change l'état du booléen **timer_added** à vrai pour ne pas créer d'autres Timer par erreur.
- Pour **m_ButtonDeleteTimer** le signal handler **on_button_delete_timer()** s'assure qu'il y a bien un Timer en testant **timer_added**. Une demande de désactivation est mémorisée en faisant passer le booléen **disconnect** à true. Le booléen **timer_added** doit bien sûr repasser à faux pour permettre de re-créer le Timer ultérieurement avec l'autre Button.
- La fonction callback **on_timeout** gère le Timer en fonction des demandes exprimées par les Buttons.

- Si une demande de fin d'existence du Timer a été enregistrée en faisant passer le booléen **disconnect** à **true**, alors la fonction renvoie **false** ce qui a pour effet de supprimer le Timer. Auparavant il faut seulement remettre le booléen **disconnect** à **false** au cas où un Timer est créée par une demande du Button addTimer.
- Si par contre il n'y a pas de demande de fin du Timer, on se contente d'afficher la valeur d'une variable **static** locale initialisée à 1 et qui est incrémentées à chaque appel. Remarquer que la fonction renvoie **true**, ce qui exprime qu'on veut que le Timer continue son activité.

Activité : changer la valeur de la période et constater la qualité de la synchronisation avec le temps du monde réel (wall-clock time). *On se satisfera de cette qualité pour le projet*. Par exemple, commencez en indiquant une période de 1000ms puis regardez votre montre au moment où vous appuyez sur Start et lorsque le programme affiche 10. Continuez en diminuant la période : (100ms, affiche 100), (10ms, affiche 1000),(1ms, affiche 10000).

Que constatez-vous ?

Exercice 4 (niveau 0) :

execution d'une simulation aussi vite que possible avec idle

Cet exercice simplifie autant que possible l'exemple fourni par le manuel GTKmm 3 dans le sens où on gère une seule fonction idle.

On voit aussi comment contrôler le lancement et l'arret de la simulation par l'intermédiaire d'un booléen dont la valeur est modifiée à l'aide d'un bouton.

Le programme principal définit un widget IdleExampleWindow dérivé de la classe Window

```
#include "idle.h"
#include <gtkmm/application.h>

int main (int argc, char *argv[])
{
  auto app = Gtk::Application::create(argc, argv, "org.gtkmm.example");

  IdleExample example;
  return app->run(example);
}
```

Interface du module :

On remarque la déclaration de la méthode **on_idle**(), écrite par nous, pour être appelée quand il n'y aura aucun événement à traiter. Elle précède les signal handler des bouton Quit et StartSim ; ce second bouton va contrôler le booléen **started** qui sera utilisé par **on_idle().**

```
#ifndef GTKMM_EXAMPLE_IDLEEXAMPLE_H
#define GTKMM_EXAMPLE_IDLEEXAMPLE_H

#include <gtkmm.h>
#include <iostream>

class IdleExample : public Gtk::Window
{
public:
  IdleExample();

protected:
  // Signal Handlers:
  bool on_idle();
  void on_button_clicked_Quit();
  void on_button_clicked_StartSim();

  // Member data:
  Gtk::Box m_Box;
  Gtk::Button m_Button_Quit;
  Gtk::Button m_Button_StartSim;

private:
  bool started;
};

#endif // GTKMM_EXAMPLE_IDLEEXAMPLE_H
```

Implémentation du module :

```
#include <cstdlib>
#include "idle.h"
using namespace std;

IdleExample::IdleExample() :
  m_Box(Gtk::ORIENTATION_HORIZONTAL, 5),
  m_Button_Quit("Quit", true),
  m_Button_StartSim("Start-Stop Simulation with idle function", true),
  started(1)
{
  set_border_width(5);
  // Put buttons into container
  add(m_Box);

  m_Box.pack_start(m_Button_Quit, false,false);
  m_Box.pack_start(m_Button_StartSim, false,false);

  // Connect the signal handlers:
  m_Button_Quit.signal_clicked().connect( sigc::mem_fun(*this,
              &IdleExample::on_button_clicked_Quit) );

  // Connect the signal handlers:
  m_Button_StartSim.signal_clicked().connect( sigc::mem_fun(*this,
              &IdleExample::on_button_clicked_StartSim) );

  // updating a simulation in idle signal handler
  // called as quickly as possible
  Glib::signal_idle().connect( sigc::mem_fun(*this,
                               &IdleExample::on_idle) );

  show_all_children();
}

void IdleExample::on_button_clicked_Quit()
{
  cout << "The End" << endl;
  exit(0);
}

void IdleExample::on_button_clicked_StartSim()
{
  started = !started ; // change the simulation activation at each call
}

// This idle callback function is executed as often as possible,
// hence it is ideal for processing intensive tasks.
bool IdleExample::on_idle()
{
  static unsigned count(0);
  if(started)
  {
      cout << "Mise à jour de la simulation numéro : " << ++count <<endl;
  }
  return true;  // return false when done
}
```

Le constructeur précise les label des Buttons dans la liste d'initialisation.
Le booléen **started** est à 1 = **true** à l'initialisation pour montrer que la simulation peut s'exécuter à pleine vitesse dès le lancement du programme. Il va servir de relai entre la simulation effectuée par **on_idle**() et le bouton **StartSim**.

Le constructeur dispose les Buttons dans la Box horizontalement (conformément au paramètre de la liste d'initialisation) et les connectent à leur signal handler.

Le point important est d'indiquer à GTK que notre méthod **on_idle**() est la fonction à appeler lorsqu'il n'y a pas d'événement ; ce qui est fait avec cette instruction :

```
Glib::signal_idle().connect( sigc::mem_fun(*this, &IdleExample::on_idle) );
```

L'implémentation continue avec les signal handlers. Le second est le plus intéressant ; chaque appel du signal handler **on_button_clicked_StartSim**() fait passer le booléen **started** dans l'état opposé.

Cet attribut **started** est aussi visible pour la méthode **on_idle**() comme on peut le voir dans son implémentation. S'il est true, on affiche un message avec la valeur du compteur **count** qui est incrémentée. Bien sûr pour un vrai projet on aurait ici un appel d'une fonction ou méthode faisant *une seule* mise à jour d'une simulation.

Remarquer que la méthode **on_idle** renvoie true pour cet exemple de façon à mettre en place une boucle infinie de simulation. Si la tâche à réaliser par **on_idle**() était différente, on pourrait quitter cette méthode en renvoyant false quand on estime que sa tâche est terminée. Dans ce cas, elle ne sera plus du tout appelée.

Activités :

- Changer **on_idle** pour que cette méthode renvoie false lorsque la valeur de count est supérieure à une valeur donnée, par exemple 1000000. Normalement La simulation ne continue plus au-delà de cette valeur de compteur.

- Ajouter un bouton **Step** qui fait avancer la simulation d'une seule mise à jour. Pour cela il faut prévoir un autre attribut **step** initialisé à false. Cet attribut passe à true quand on clique sur le bouton Step. Ensuite il faut mettre à jour la méthode **on_idle** :
    - L'action du bouton Step ne doit est prise en compte que si le booléen started est à false (simulation en pause).
    - Dans ce cas, Si le booléen **step** est à true alors faite une seule mise à jour, ET refaite passer le booléen **step** à false