EPFL

Architecture d'un programme interactif graphique

Gestion des événements de la souris et du clavier

Objectifs:

- compléter la maîtrise de la boucle d'interaction

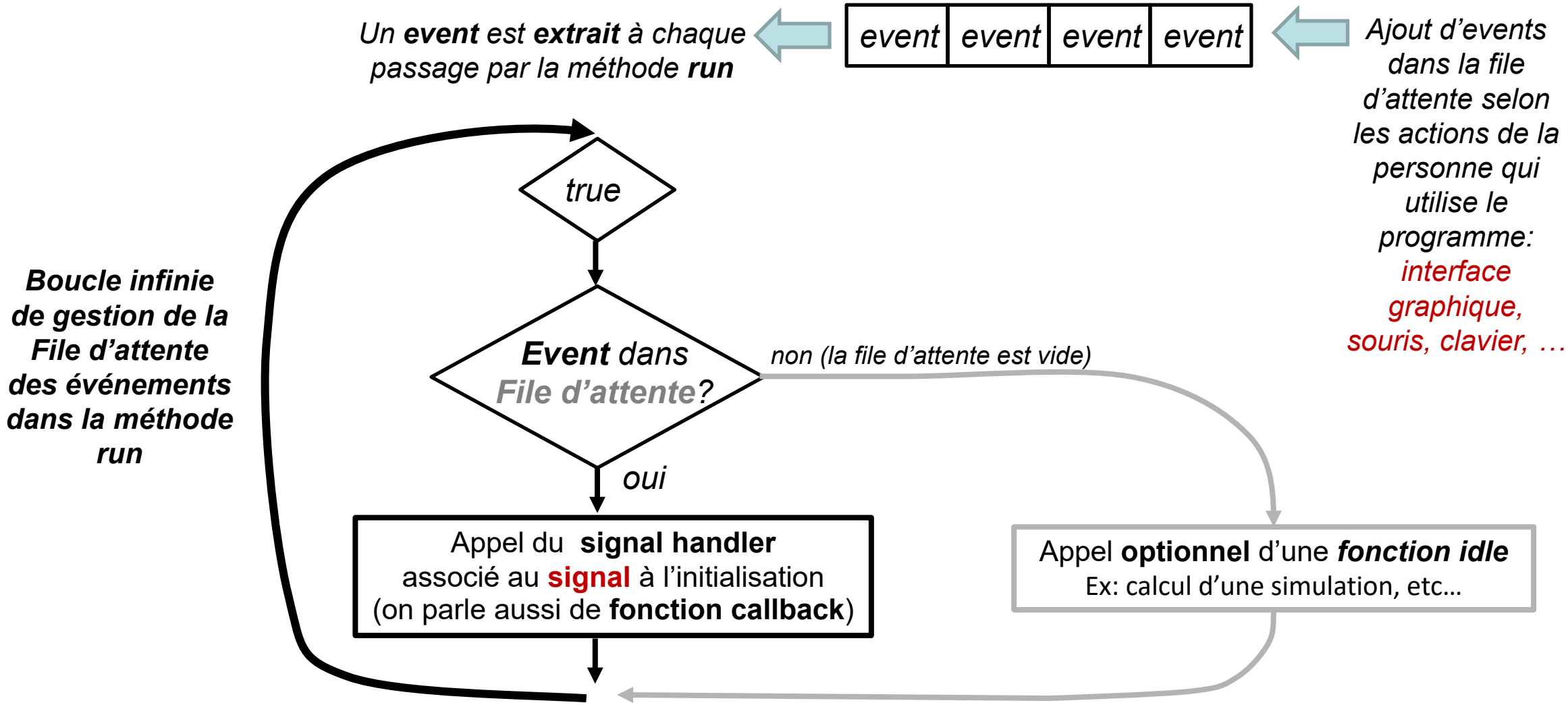
Plan:

- Rappel et but de l'exemple
- Conversion de coordonnées vers l'espace du dessin
- Eléments importants de l'implementation
- Comment adapter l'exemple à un contexte de projet

Rappel : La programmation par événements

Pseudocode de la boucle infinie de gestion de la file d'attente des événements dans la méthode run(…)

Chaque ***événement*** **=** ***event*** **=** ***signal*** est mémorisé dans une **File d'attente d'événements** selon son instant de création



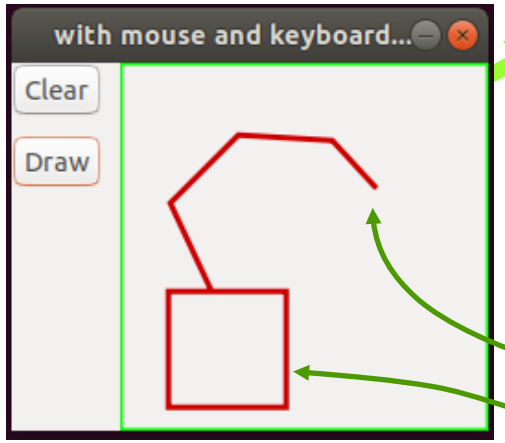
Répartition des rôles dans l'exemple myevents.cc

L'interface **myEvents** est dérivée de **Window**

En plus des Buttons déjà présentés en sem6, elle déclare trois signals handlers pour tirer parti des boutons de la souris et du clavier:

1

on_button_press_event(…)
on_button_release_event(…)
on_key_press_event(…)



myArea est dérivée de **DrawingArea**

2

En plus des éléments déjà présenté en sem6, on trouve:

- des attributs pour mémoriser un dessin
- Des méthodes associées à l'action des boutons de la souris et du clavier:
  - **add_point(…)**
  - **reset_rect(…) et finalize_rect(…)**
  - **reset()**

3 Lien entre un *event* et réaction du programme:

L'action des boutons de la souris et du clavier modifie les informations du dessin mémorisées dans l'instance de myArea ***ET*** demande un nouvel affichage à l'aide de la méthode **refresh()**

Exemple:

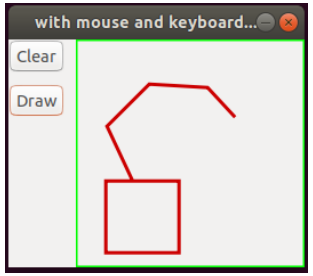
frappe!

Appel du signal handler : **on_key_press_event(…)** ⇨ Appel de : **reset()** ⇨

1) Ré-init les attributs du dessin
2) Appel de **refesh()** ⇨

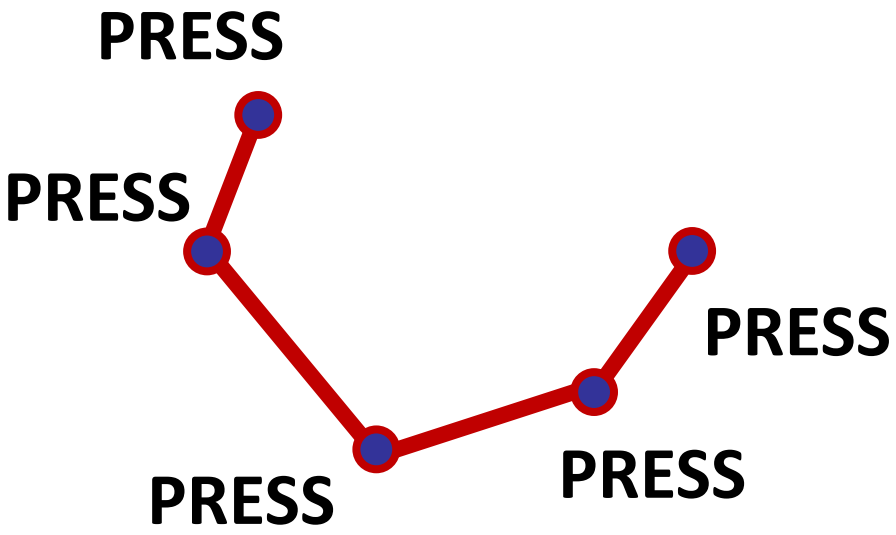
La méthode **refresh()** *efface* la surface entière de la fenêtre graphique, ce qui produit un **event** qui va causer l'appel de **on_draw()**

Usage des boutons de la souris pour le dessin :



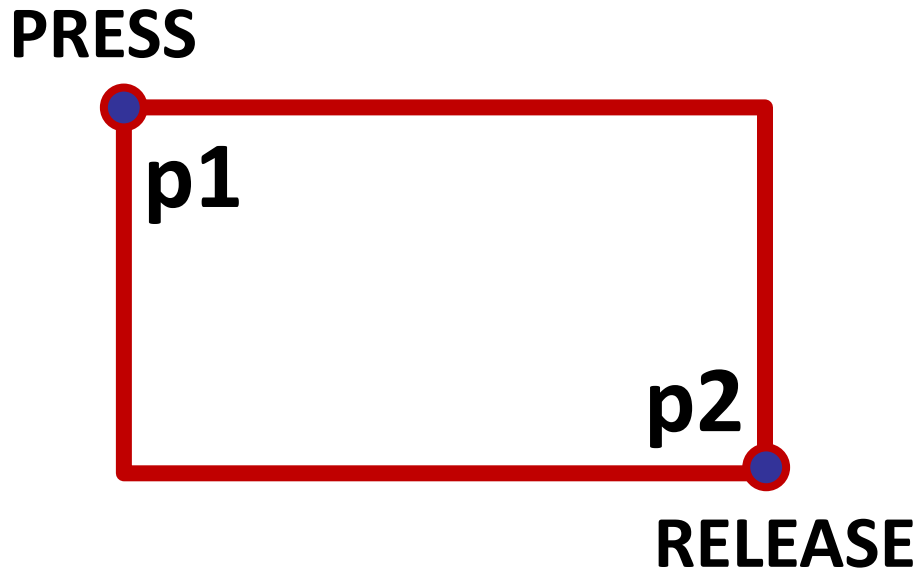
Bouton gauche de la souris:

Chaque clic ajoute un point à une liste qui sera visualisée comme une polyline



Bouton droit de la souris:

Appuyer sur le bouton définit **p1**, bouger puis relacher pour définir **p2**. Ils seront visualisés comme un rectangle



Conversion : des coordonnées *fenêtre GTKmm* aux coordonnées *DrawingArea*

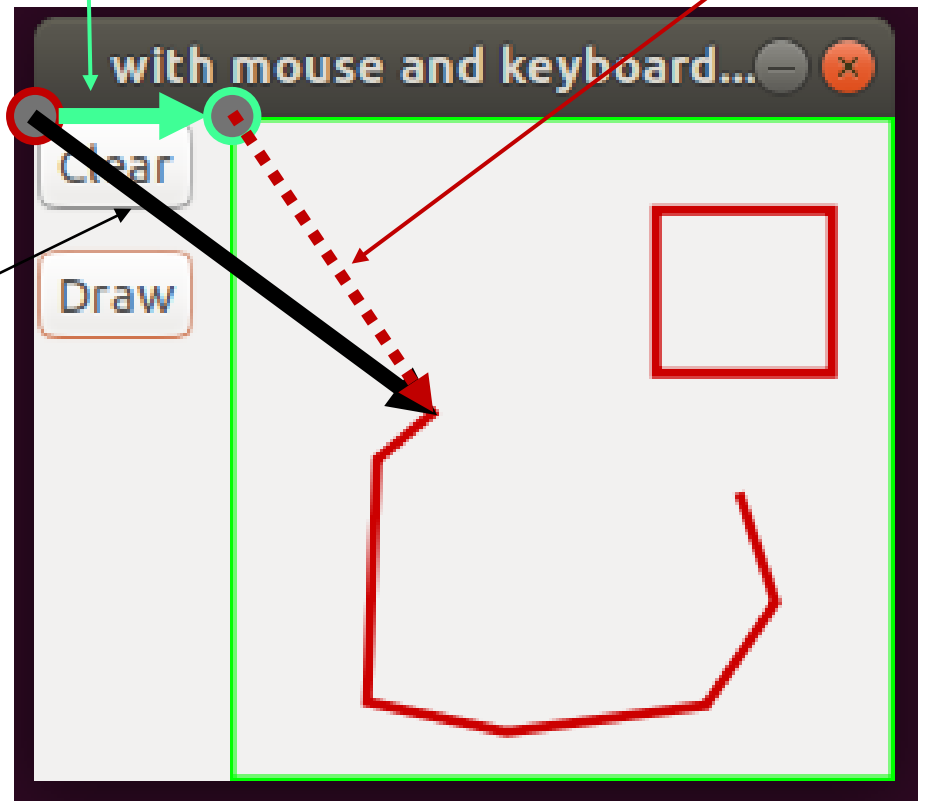
CEPENDANT, Si l'origine de l'espace de dessin DrawingArea ne coïncide pas avec l'origine de la fenêtre GTKmm, il faut convertir (Xf,Yf) dans le système de coordonnées de DrawingArea

Un event de la souris permet d'obtenir les coordonnées (**Xf,Yf**) du pointeur de la souris dans le système de coordonnées de la *fenêtre* GTKmm

Origine (Xo,Yo) de MyArea dans la fenêtre GTK

Coordonnées (Xd,Yd) à fournir à MyArea pour le **dessin**

Coordonnées (Xf,Yf) de la souris dans la fenêtre GTKmm, obtenues avec le signal handler du bouton



$$\begin{cases} Xd = Xf - Xo \\ Yd = Yf - Yo \end{cases}$$

En vert et bleu les nouveautés
par rapport à l'exemple du cours précédent

myevents.h

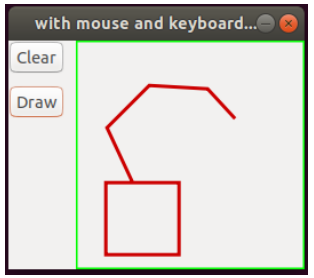
```
…
class MyEvent : public Gtk::Window
{
public:
  MyEvent();
  virtual ~MyEvent();

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;
};
…
```

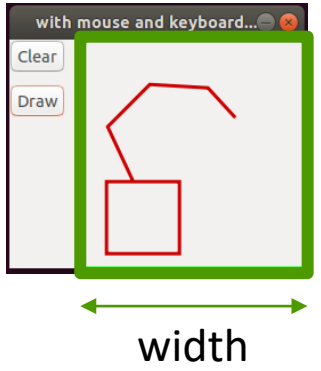



```
…
class MyArea : public Gtk::DrawingArea
{
public:
  MyArea();
  virtual ~MyArea();
  void clear();
  void draw();
  void reset();
  void reset_rect(Point p);
  void finalize_rect(Point p);
  void add_point(Point p);

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();
};
…
```

myevents.cc




```
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 ;

        // origin of the drawing area + width and height
        double origin_x = m_Area.get_allocation().get_x();
        double origin_y = m_Area.get_allocation().get_y();
        double width    = m_Area.get_allocation().get_width();
        double height   = m_Area.get_allocation().get_height();

        // 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});

            if(event->button == 3) // Right mouse button
            {
                m_Area.finalize_rect(p);
            }
        }
    }
    return true;
}
```

EPFL

```
…
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();
}
```

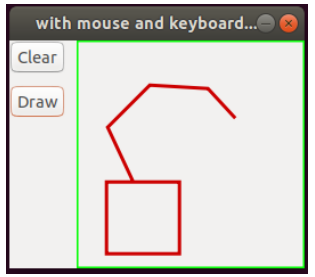
myevent.cc / **code du widget de dessin**

```
…
bool MyArea::on_draw(const Cairo::RefPtr<Cairo::Context>& cr)
{
    …
        cr->set_line_width(3.0);
        cr->set_source_rgb(0.8, 0.0, 0.0);  // red  line
        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();
    …
    return true;
}
…
```



L'appel de **refresh()** produit un event = ***signal***
qui lui-même produira l'appel de **on_draw()**

Comment adapter cet exemple à l'échelle d'un projet

- A l'échelle d'un projet l'ensemble des données manipulées (ici p1, p2, line) constituent un **Modèle** qui doit être géré dans son sous-système propre

- Les signal handlers de l'interface graphique, de la souris ou du clavier doivent :
  - Appeler des méthodes du **Modèle** pour mettre à jour ses données
  - Produire un *event* de rafraichissement de l'affichage graphique si l'état du **Modèle** change

- Le signal handler **on_draw()** effectue les conversions de coordonnées entre :
  - les coordonnées fenêtre (**Xf,Yf**) et les coordonnée (**Xd,Yd**) du Drawing Area
  - les coordonnées (**Xd,Yd**) du Drawing Area et celles du Modèle (**Xm,Ym**) à l'aide du cadrage de l'espace du Modèle [Xmin,Xmax]x[Ymin,Ymax]

De l'espace du Modèle à la fenêtre du dessin	**Xd = width *(Xm –Xmin)/(Xmax-Xmin)** **Yd= height *(Ymax – Ym)/(Ymax-Ymin)**
De la fenêtre du dessin à l'espace du Modèle	**Xm = (Xd/width)*(Xmax–Xmin) + Xmin** **Ym = Ymax – (Yd/height)*(Ymax-Ymin)**