EPFL

Architecture d'un programme interactif graphique

Comparaison de **idle** et d'un **timer** pour piloter une simulation

Objectifs:

- Programmation par événement : Mise à jour d'un Modèle
- mise à jour asynchrone avec idle() : exemple sans dessin
- mise à jour asynchrone avec idle() : exemple avec dessin
- mise à jour synchronisée à un timer

Remarques:

Revoir le cours de la semaine6 sur la gestion d'événements du clavier (pas détaillé ici)
Tout le code présenté dans ce cours est fourni et détaillé dans la série7 niveau 0 exercices 3, 4 et 5.

Les exercices suivants de la série7 niveau 0 ne sont pas détaillés en cours:

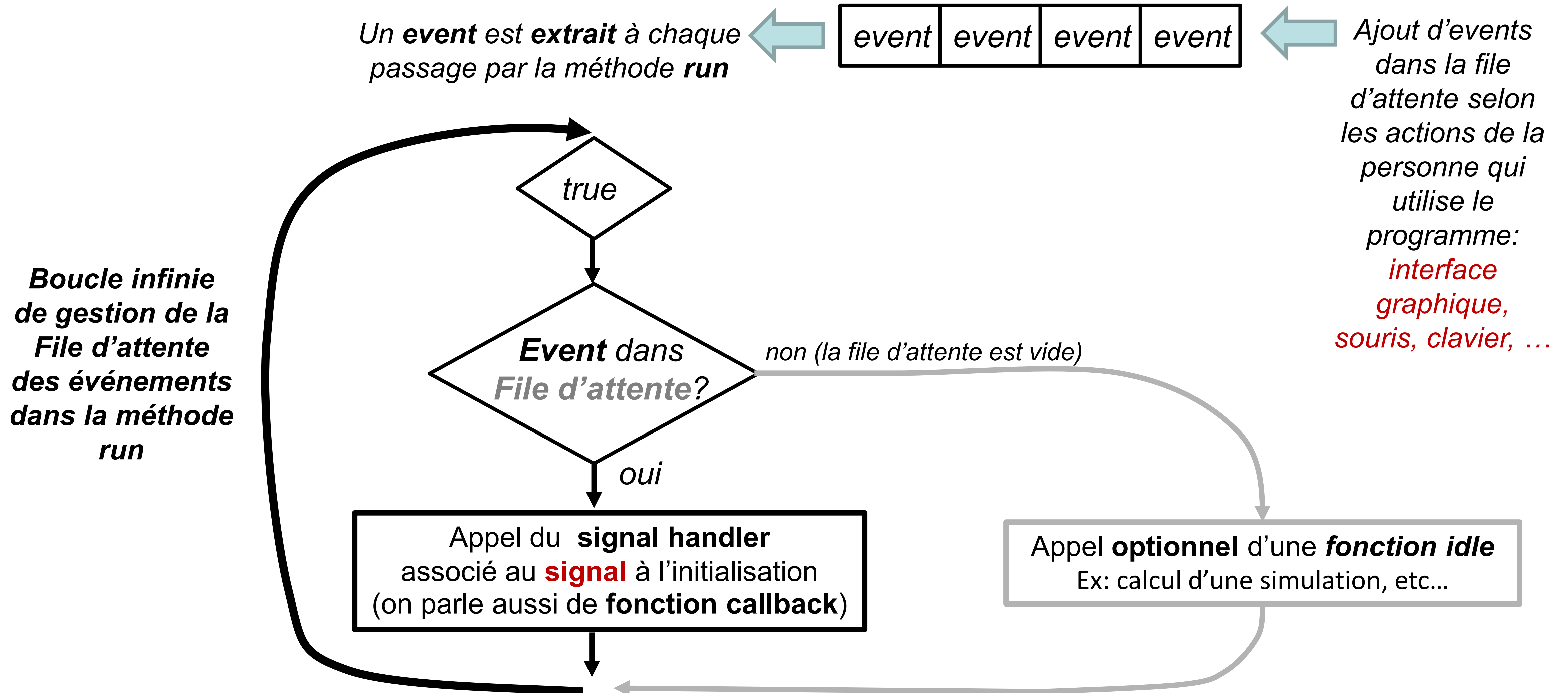
L'exercice1 explique le code pour ouvrir un fichier avec GTKmm => projet
L'exercice2 détaille comment tirer parti de la souris => optionnel pour 2021-22

EPFL

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



EPFL En vert et bleu les points intéressants

Gestion du temps avec **on_idle()** [cas sans dessin]

idleKeyb.h

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

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

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

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

private:
  bool started;  // start-stop de la fonction idle
  bool step;     // un seul pas de simulation
};
…
```

Méthodes:

- **on_idle()** : appelée si aucun événement à traiter
- Pour le bouton pour démarrer/arrêter la simulation
- Pour traiter les événements du clavier

Attributs:

- **started** : état de la simulation active/inactive
- **step**: pour activer une seule mise à jour quand la simulation est dans l'état stoppé

EPFL En vert et bleu les points intéressants

idleKeyb.h

Gestion du temps avec **on_idle()** [cas sans dessin]

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

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

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

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

private:
  bool started;  // start-stop de la fonction idle
  bool step;     // un seul pas de simulation
};
…
```

Méthodes:

- **on_idle()** : appelée si aucun événement à traiter
- Pour le bouton pour démarrer/arrêter la simulation
- Pour traiter les événements du clavier

Attributs:

- **started** : état de la simulation en_cours/stoppée
- **step**: pour activer une seule mise à jour quand la simulation est dans l'état stoppé

idleKeyb.cc : constructeur

```
…
IdleExample::IdleExample() :
  m_Box(Gtk::ORIENTATION_HORIZONTAL, 5),
  m_Button_Quit("Quit", true),
  m_Button_StartSim("Start-Stop Simulation with idle function", true),
  started(true), step(false)
{
  set_border_width(5);
  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) );

  // Events from keyboard
  add_events(Gdk::KEY_RELEASE_MASK);

  show_all_children();
}
…
```

Initialisation des attributs dans la liste d'initialisation

Connection de nos méthodes callback aux signaux auxquels elles doivent réagir

EPFL

idleKeyb.cc (suite)

Alternance de l'état started entre actif et inactif quand on appuie sur le bouton ou sur la touche 's'

L'attribut step passe à vrai seulement si on appuie sur la touche '1' et si la simulation est dans l'état inactif

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

void IdleExample::on_button_clicked_StartSim()
{
  started = !started ;
}
```

```
…
bool IdleExample::on_key_press_event(GdkEventKey * key_event)
{
  if(key_event->type == GDK_KEY_PRESS)
  {
     switch(gdk_keyval_to_unicode(key_event->keyval))
     {
        case 's':
           cout << " key 's' pressed !" << endl;
           started = !started ;
           return true;
        case '1':
           cout << " key '1' pressed !" << endl;
           if(!started)
              step = true;
           else
           {
              cout << " NOT VISIBLE … !" <<endl;
              //~ exit(0);
           }
           return true;
        case 'q':
           cout << "Quit" << endl;
           exit(0);
           break;
     }
  }
  return Gtk::Window::on_key_press_event(key_event);
}
```

```
…
// 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;
  }
  else if (step)
  {
      step = false;
      cout << "Mise à jour de la simulation numéro : " << ++count << endl;
  }

  return true;
}…
```

Le compteur qui représente la simulation est incrémenté seulement si started est vrai.

Le compteur est aussi incrémenté si started est faux et step est vrai.
Ensuite step passe immédiatement à faux
pour qu'il n'y ait pas de mise à jour au prochain appel de on_idle()

idleKeyb.cc (suite)

Ce mécanisme fonctionne très efficacement s'il n'y a pas de dessin à effectuer

EPFL

Gestion du temps avec **on_idle()** [cas avec dessin]

Contexte: l'application est complétée avec un attribut **m_Area** pour effectuer le dessin

Comment demander le dessin depuis on_idle() ? Indirectement avec un appel de la méthode refresh() sur **m_Area**

Cependant on observe que le temps de traitement de la demande de dessin est loin d'être négligeable ; plusieurs mises à jour de la simulation sont faite avant de pouvoir observer la simulation

```
#include <chrono>
#include <thread>

bool IdleExample::on_idle()
{
  static unsigned count(0);
  m_Area.refresh(); // la méthode refresh() doit être "public" pour pouvoir effectuer cet appel.
  if(started)
  {
        std::this_thread::sleep_for(std::chrono::milliseconds(25)); // sleep de 25 ms
        cout << "Mise à jour de la simulation numéro : " << ++count << endl;
  }
  else if (step)
  {
        step = false;
        cout << "Mise à jour de la simulation numéro : " << ++count << endl;
  }

  return true;
}
```

Solution: mettre en pause pendant X ms pour donner le temps d'être traité à l'événement qui va appeler on_draw()

EPFL basicTimer.h

Gestion du temps avec **un timer()** [cas avec dessin]

```
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();

  // 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
  const int timeout_value;
};
```

Méthodes:

- **on_timeout()** : appelée après une durée prédéfinie en ms
- Pour le bouton pour démarrer/arrêter le timer

Attributs:

- **timer_added et disconnect** : création et gestion d'un seul timer
- **Timeout_value** : un événement est produit qui appelle la callback **on_timeout** après cette durée en ms.

basicTimer.cc : constructeur

Initialisation des attributs dans la liste d'initialisation

Connection de nos méthodes callback aux signaux auxquels elles doivent réagir (sauf celle du timer ; cf slide suivant)

```
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();
}
```

basicTimer.cc (suite)

```
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;
}

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;
  }
}
```

Connection de la callback du timer à sa création (quand on appuie sur le bouton «add») ; l'attribut timeout_value est alors fourni

Initialisation des attributs dans la liste d'initialisation

EPFL

callback automatiquement appelée quand la durée timeout_value est écoulée

basicTimer.cc (suite)

```
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 simulation update number : " << val << std::endl;
  ++val; // tic the simulation clock


  // A call to make a single update of the simulation is expected here
  // Then a call to refresh the visualization (if any) would be done here

  // Keep going with the timer ; launch the next countdown

  return true;
}
```

Le compteur qui représente la simulation est incrémenté