Event Processing

Leow Wee Kheng
CS3249 User Interface Development
Events are created in response to user inputs.
Questions:
- Where are the event handlers (in which part of application)?
- Who receive events?
Different frameworks process events differently.

Let's examine each of them.
X Window Event Processing

- XtAppMainLoop() runs event loop.
- Event is sent to widget by calling action routine.
- Action routine calls application's callback if registered.
Qt application class

- `QCoreApplication`: for non-GUI applications.
- `QApplication`: for GUI applications, inherits `QCoreApplication`.
- `qApp`: points to unique Qt application instance.
QCoreApplication::exec() runs event loop.
- Gets native window event from event queue.
- Translates into QEvent (or subclass).
- Sends QEvent to QObject by calling QObject::event().

QObject::event() 
- Main event handler.
- Forwards event to specific event handler.
In Qt, event ≠ signal.

- Events are useful when implementing a widget.
  Signals are useful when using a widget.
- Event handling is lower-level mechanism.
  Signal-slot is higher-level mechanism.
- Qt provides 5 levels of event processing:
  1. Reimplement specific event handlers.
     • Change behaviour of event handlers.
  2. Reimplement `QObject::event()` main event handler.
     • Catch events before they reach specific event handlers.
  3. Install event filer on an object.
     • Events for the object are first sent to its event filter.
  4. Install event filter on `QApplication` object.
     • All events for all objects are first sent to its event filter.
  5. Subclass `QApplication` and reimplement `notify()`.
     • Catch all events before they are sent to any event filer.
Event propagation

- Events can be propagated to parent if not handled.

```cpp
MyWindow::mouseEvent(QMouseEvent *event)
{
    if (don’t want to handle)
        QWidget::mouseEvent(event);
    else ...
}
```
Cocoa application class

- NSApplication
- NSApp: NSApplication object.
Responder Model

- NSApplication, NSWindow, NSView are subclasses of NSResponder, which can receive and handle events.
- Event messages
  - Messages that correspond to input event, e.g., mouse click.
- Action messages
  - Messages describing higher-level command, e.g., copy.
- Responder chain
  - A series of responder objects that handle message.
Class Hierarchy

```
    NSObject
   /     \
  NSResponder    NSEvent
     \       /  
   NSApplication  NSWindow  NSView
```
Event Routing

The diagram illustrates the responder chain for event processing in NSApplication, NSWindow, NSView, and Control objects. Events are queued and then routed through the responder chain. If an event is not handled, it is forwarded to the next object in the chain. The responder chain is indicated with red arrows.

- Event Queue
- NSEvent
- NSApplication
- NSWindow
- NSView
- Control
- target
- action
- NSObject
- method

responder chain: forward events not handled
Event Delegation

- Allows an object to delegate responsibility to another.
- Delegate
  - Receives messages from another object when events occur.
  - Helps to perform tasks for sender object.
- Change object's behaviour without creating subclass.
Notification

- Broadcast messages to objects in application.
- Notification center
  - Object of NSNotificationCenter class, has default center.
 Notification
  ○ Sender defines names of notifications to post as NSString.
  ○ Receiver registers with (default) notification center
    • notification to receive
    • method to invoke
  ○ Sender posts notification (NSNotification) to notification center.
  ○ Notification center sends notification to receiver by calling receiver's method.
Window API is developed in C.

- **WinMain()**
  - contains message (event) loop

- **WindowProc()**
  - main message (event) handler
int WINAPI WinMain(HINSTANCE instance, HINSTANCE notused, LPSTR commandLine, int showOptions)
{
    WNDCLASS WindowClass;
    ... // set WindowClass parameters
    RegisterClass(&WindowClass);

    window = CreateWindow(...);
    ShowWindow(window, showOptions);
    UpdateWindow(window);

    // message loop
    MSG msg;
    while(GetMessage(&msg, 0, 0, 0) == TRUE)
    {
        TranslateMessage(&msg);
        DispatchMessage(&msg);
    }
    return static_cast<int>(msg.wParam);
}
LRESULT CALLBACK WindowProc(HWND window, UINT msgId, WPARAM wParam, LPARAM lParam)
{
    switch(msgId)
    {
    case WM_CREATE:
        ... // create window
        return 0;

    case WM_DESTROY:
        PostQuitMessage(0);
        return 0;

    default: // call default message handler
        return DefWindowProc(window, msgId, wParam, lParam);
    }
}
MFC encapuslates Windows API in C++ classes.

- `CWinApp::Run()` runs message (event) loop.
- `CWinApp` and `CWnd` have message (event) handlers.
Class Hierarchy

```
COBJECT
  \--------
  |       |
  |       |
  |       |
  CObject
  \--------
        \-
        |  |
        |  CCmdTarget
        \-
              \-
              |  |
              |  CWInThread
              \-
                                \-
                                |  |
                                |  CWinApp
                                \-
                                                \-
                                                |  |
                                                |  MyApp
                                                \-
                                                \-
                                                |  |
                                                |  CWnd
                                                \-
                                                      \-
                                                      |  |
                                                      |  CFrameWnd
                                                      \-
                                                            \-
                                                            |  |
                                                            |  MyWindow
```
Event Receiver

Two kinds

- Top-level window
  - Examples: Windows API / MFC, ...

- Widget / object
  - Examples: X, Qt, Cocoa
Top-Level Window Handles Events
Advantages
- Top-level window contains all UI elements.
- Easy to synchronise states of UI elements.
- Simple mechanism.

Disadvantages
- Top-level window is tightly coupled to UI elements.
- UI elements' functions are incomplete; some are in top-level window (low cohesion).
- Cannot define part of the window as widget for reuse.
Widget Handles Events

Window

Spin Box Pointer

Slider Pointer

Spin Box

Slider

Spin Box Handler

Slider Handler

Enter your age

35
Advantages
- Top-level window can be loosely coupled to UI elements.
- UI elements' functions are more complete (high cohesion).
- Can define part of the window as widget for reuse.
- Powerful mechanism.

Complication
- How to coordinate actions of UI elements?
  - Centralised at top-level window
    - Simple but tight coupling with UI elements.
  - Decentralised to UI elements
    - Possible tight coupling between UI elements, or need observer mechanism.
- X Window's solution: callback
  - Application synchronises states of UI elements.
    - Easy to use, but tightly coupled.

- Cocoa's solution: target-action
  - Top-level window synchronises states of UI elements.
    - Easy to use, but tightly coupled.

- Qt's solution: signal-slot ★
  - Widgets synchronise their states by signals and slots.
    - Easy to use, and yet loosely coupled.
## Comparison

<table>
<thead>
<tr>
<th></th>
<th>Qt</th>
<th>X11/Motif</th>
<th>Cocoa</th>
<th>MFC</th>
</tr>
</thead>
<tbody>
<tr>
<td>event loop</td>
<td>QApplication::exec()</td>
<td>XtAppMainLoop()</td>
<td>in NSApplication</td>
<td>CWinApp::Run()</td>
</tr>
<tr>
<td>event receiver</td>
<td>QWidget</td>
<td>widget, application</td>
<td>target object, top-level window</td>
<td>top-level window</td>
</tr>
<tr>
<td>event handler</td>
<td>QWidget, event handler</td>
<td>widget's action routine, application's callback</td>
<td>target's action function</td>
<td>top-level window, message handler</td>
</tr>
<tr>
<td>synchronisation</td>
<td>signal-slot</td>
<td>application</td>
<td>target top-level window</td>
<td>top-level window</td>
</tr>
</tbody>
</table>
Qt Event Handlers

- Reimplement event handlers
  - We have used this technique before.
  - We'll see more of it in later lectures.

```cpp
void MyEditor::closeEvent(QCloseEvent *event)
{
    if (okToContinue())
        event->accept();
    else
        event->ignore();
}
```

- closeEvent() is reimplemented in MyEditor.
- QCloseEvent is a subclass of QEvent.
Qt Event Filter

- **Unchecked**
- **Hidden**
- **Checked**
- **Hidden**
Two ways to create volume render dialog:

- **Subclass**
  - VolumeRenderDialog as subclass of QWidget
  - QWidget does not emit close signal when hiding.
  - Define closed() signal in VolumeRenderDialog.
  - Use closed() to uncheck action.

- **Composition**
  - VolumeRenderDialog is a QWidget in main window.
  - Use close button's clicked() signal to uncheck action.
  - Use event filter to catch close event and uncheck action.
Subclass Method

class VolumeRenderDialog:: public QWidget
{
  Q_OBJECT

public:
  VolumeRenderDialog();
  ...

protected:
  void closeEvent(QCloseEvent *event);

signals:
  void closed();
  ...
};
Hide by clicking close button:

```cpp
void VolumeRenderDialog::createWidgets()
{
    QPushButton *closeButton = new QPushButton("close");
    ...
    connect(closeButton, SIGNAL(clicked()),
            this, SLOT(hide()));
    connect(closeButton, SIGNAL(clicked()),
            this, SIGNAL(closed())); // emit signal
}
```

Hide by clicking close window button:

```cpp
void VolumeRenderDialog::closeEvent(QCloseEvent *event)
{
    event->accept();
    emit closed();
}
```
class MainWindow:: public QMainWindow
{
    Q_OBJECT

public:
    MainWindow();
    ...

private:
    QAction *volumeRenderAction;
    VolumeRenderDialog *volumeRenderDialog;
    ...
};
void MainWindow::createWidgets()
{
    volumeRenderDialog = new VolumeRenderDialog;
    ...
}

void MainWindow::createActions()
{
    volumeRenderAction = new QAction("Volume Render", this);
    volumeRenderAction->setCheckable(true);
    ...
    connect(volumeRenderAction, SIGNAL(toggled(bool)),
            volumeRenderDialog, SLOT(setVisibility(bool)));
    connect(volumeRenderDialog, SIGNAL(closed()),
            VolumeRenderAction, SLOT(toggle()));
}
Composition Method

class MainWindow:: public QMainWindow
{
    Q_OBJECT

public:
    MainWindow();
    ...

protected:
    bool eventFilter(QObject *object, QEvent *event);

private:
    QAction *volumeRenderAction;
    QWidget *volumeRenderDialog;
    ...
};
Hide by clicking close button:

```cpp
void MainWindow::createWidgets()
{
    volumeRenderDialog = new QWidget;
    volumeRenderDialog->installEventFilter(this);

    QPushButton *closeButton = new QPushButton("close");
    ...
    connect(closeButton, SIGNAL(clicked()),
            volumeRenderDialog, SLOT(hide()));
    ...}
```
Hide by clicking close button:

```cpp
void MainWindow::createActions()
{
    volumeRenderAction = new QAction("Volume Render", this);
    volumeRenderAction->setCheckable(true);
    ...
    connect(volumeRenderAction, SIGNAL(toggled(bool)),
            volumeRenderDialog, SLOT(setVisibility(bool)));
    connect(closeButton, SIGNAL(clicked()),
            volumeRenderAction, SLOT(toggle()));
}
```
Hide by clicking close window button:

```cpp
bool MainWindow::eventFilter(QObject *object, QEvent *event)
{
    if (object == volumeRenderDialog)
        if (event->type() == QEvent::Close)
            { 
            volumeRenderAction->setChecked(false);
            return false; // Let volumeRenderDialog
            // handle close event 

        }

    // Let parent widget handle event.
    return QMainWindow::eventFilter(object, event);
}
```
Staying Responsive

- Event processing should be fast.
  - Other events are waiting to be dispatched and processed.
  - Slow event processing can freeze the whole GUI.

- But some processes just need more time.
  - Example: Saving a large spread sheet into disk.

- Possible solutions
  - Show wait cursor to inform the user to just wait.
  - Call dispatcher to dispatch events regularly.
  - Run long operations when system is idle (further reading).
  - Use multithreading (later lecture).
  - Use lazy evaluation, level-of-details (later lecture).
bool Spreadsheet::writeFile(const QString &fileName)
{
    QFile file(fileName);
    QTextStream out(&file);

    for (int row = 0; row < RowCount; ++row)
    {
        for (int col = 0; col < ColumnCount; ++col)
        {
            QString str = formula(row, col);
            if (!str.isEmpty())
                out << quint16(row) << quint16(col) << str;
        }

        qApp->processEvents();  // Dispatch some events.
    }

    return true;
}
Calling `qApp->processEvents()` is a bit dangerous
- User may close the application while it is writing.

Solution:

```cpp
cqApp->processEvents(QEventLoop::ExcludeUserInputEvents);
```

- Ignore key and mouse inputs.
For long-running process, can show progress dialog.

```cpp
bool Spreadsheet::writeFile(const QString &fileName)
{
    QFile file(fileName);
    QTextStream out(&file);

    QProgressDialog progress(this);
    progress.setLabelText(tr("Saving %1").arg(fileName));
    progress.setRange(0, RowCount);
    progress.setModal(true);
```
for (int row = 0; row < RowCount; ++row) {
    progress.setValue(row); // Update progress bar.
    qApp->processEvents(); // Dispatch some events.

    if (progress.wasCanceled()) {
        file.remove();
        return false;
    }

    for (int col = 0; col < ColumnCount; ++col) {
        QString str = formula(row, col);
        if (!str.isEmpty())
            out << quint16(row) << quint16(col) << str;
    }
}
return true;
Timer Event

- Timer events are delivered at “regular interval”.
  - Allow applications to perform processing at regular interval.
  - Implement blinking cursors, animations, video player, etc.
- Illustrate with an animated banner
/ Ticker.h

#ifndef TICKER_H
#define TICKER_H

#include <QWidget>

class Ticker : public QWidget
{
    Q_OBJECT
    Q_PROPERTY(QString text READ text WRITE setText)

public:
    Ticker(QWidget *parent = 0);
    QString text() const { return myText; }
    void setText(const QString &newText);
    QSize sizeHint() const;

};
Reimplement 4 event handlers.
// Ticker.cpp

#include <QtGui>
#include "Ticker.h"

Ticker::Ticker(QWidget *parent): QWidget(parent)
{
    offset = 0;
    myTimerId = 0; // Indicate no timer has started.
}

void Ticker::setText(const QString &newText)
{
    myText = newText;
    update(); // Repaint.
    updateGeometry(); // Update if size has changed.
}
QSize Ticker::sizeHint() const
{
    return fontMetrics().size(0, text());
}

- **fontMetrics()** returns **QFontMetrics object.**
  - Use it to check info about the widget's font.
- **fontMetrics().size()** returns pixel length of **text().**
  - First argument is not needed; so set to 0.
void Ticker::paintEvent(QPaintEvent * /* event */) {
    QPainter painter(this);
    int textWidth = fontMetrics().width(text());
    if (textWidth < 1)
        return;

    int x = -offset; // Start position offset to the left.
    while (x < width()) // Repeat while have space
    {
        painter.drawText(x, 0, textWidth, height(),
                         Qt::AlignLeft | Qt::AlignVCenter, text());
        x += textWidth;
    }
}

-offset  textWidth  x
void Ticker::showEvent(QShowEvent * /* event */) {
    myTimerId = startTimer(30);
}

- Starts timer when widget is visible.
- Qt will generate a timer event once every 30 msec.

void Ticker::hideEvent(QHideEvent * /* event */) {
    killTimer(myTimerId);  // Stops timer.
}
void Ticker::timerEvent(QTimerEvent *event)
{
    if (event->timerId() == myTimerId)
    {
        ++offset;
        if (offset >= fontMetrics().width(text()))
            offset = 0;
        scroll(-1, 0);
    }
    else
        QWidget::timerEvent(event);  // propagate
}

- scroll(-1, 0) scrolls text displayed in widget left by one pixel.  
  - It generates paint event only for newly revealed 1-pixel strip.  
  - More efficient than calling update().
- offset is increased and reset to 0
- Without resetting offset

![Ticker example](image)

```plaintext
Have a great day! ++ Hello, world! Have a gre
```
// main.cpp

#include <QApplication>
#include "Ticker.h"

int main(int argc, char *argv[]) {
    QApplication app(argc, argv);
    Ticker *ticker = new Ticker;
    ticker->setText("++ Hello, world! Have a great day! ");
    ticker->show();
    return app.exec();
}
Notes on using QTimer:

- Timer events are delivered after, not before, period is up.
- Timer events are delivered only when event loop is running.
  - When an operation is executing, timer event is not delivered until control returns to event loop.
- Timer resolution depends on operating systems:
  - Some support 1 msec interval, some support 15 msec interval.
Summary

- Handle events in response to user actions.
- Event receiver
  - Top-level window: common type.
  - Any object / widget: more sophisticated type.
- Qt event handling
  - Reimplement event handler to change behaviour.
  - Install event filter to catch event.
- If long-running, clear event queue regularly.
- Use timer event for regular activities.
Further Reading

- User timer event to indicate system idle: [Blan2008] chap. 7.
References