FIEC

FACULTAD DE INGENIERÍA EN ELECTRICIDAD Y COMPUTACIÓN

PROGRAMACIÓN APLICADA A LA AUTOMATIZACIÓN

M.Sc. Alexander Prieto León

Conferencia XII

Unidad III Programación visual y orientada a eventos para la creación de HMI.

3.4 Creación de widgets personalizados.

Objetivos:

- Crear widgets personalizados.
  - Crear nuevos widgets a partir de otros que se parezcan y tengan funcionalidades similares.
  - Crear nuevos widgets a partir de QWidget.
  - …

BIBLIOGRAFÍA

Bibliografía:

- Zhi Eng, Lee; Rischpater, Ray. Application Development with Qt Creator: Build cross-platform applications and GUIs using Qt 5 and C++, 3rd Edition. (2020). Packt Publishing. ISBN-10: 1789951755, ISBN-13: 978-1789951752.
- http://www.qtrac.eu/C++-GUI-Programming-with-Qt-4-1st-ed.zip
- Qt 5.14.2 Reference Documentation. Qt Creator Help.
- https://doc.qt.io/qtcreator/index.html
- https://doc.qt.io/qt-5/qtdesigner-manual.html

En la clase anterior:

- ¿Para manipular imágenes en Qt qué clases fundamentales se utilizan? ¿Qué características tienen estas clases?
- ¿Qué clase derivada de QPaintDevice permite manipular los pixels de las imágenes?
- ¿Cuáles son las formas de hacerle transformaciones a un dibujo?

INTRODUCCIÓN

- **¿Qué widgets ha utilizado hasta ahora en Qt?**
- **¿Con los widgets que hay en la paleta de widgets se puede hacer todo lo que usted desee para cualquier aplicación?**

INTRODUCCIÓN

- Normalmente, programando algunas funcionalidades de un widget existente en Qt usted logra satisfacer algunos objetivos de su aplicación pero, en ocasiones, no son suficientes para lograr los objetivos de su programa.

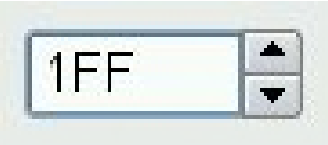
- En esta situación puede buscar un widget que se acerque al que necesita y a partir de él hacer uno nuevo con las funcionalidades extras que necesita, o si no encuentra ninguno que se acerque, a partir de QWidget crear uno nuevo con todas las funcionalidades requeridas.

Personalizando widgets:

- En algunos casos puede que un widget determinado necesite más personalización de lo que es posible editando sus propiedades o llamando a sus funciones. Una simple y directa solución es crear una subclase de dicho widget y adaptarla a nuestras necesidades.

Problema a resolver:

- Se desea desarrollar un spin box que trabaje con valores hexadecimales.



- Solución: Un QSpinBox solo soporta valores enteros, pero una subclase de esta puede utilizarse fácilmente para que acepte y muestre valores hexadecimales.

Solución HexSpinBox:

```
#include <QSpinBox>
class HexSpinBox : public QSpinBox
{
    Q_OBJECT
public:
    HexSpinBox(QWidget *parent = 0);
    protected:
    virtual QValidator::State validate(QString &text,int &pos)
    const;
        virtual int valueFromText(const QString &text) const;
        virtual QString textFromValue(int value) const;
    private:
        QRegExpValidator *validator;
};
```

- La clase HexSpinBox hereda la mayoría de sus funcionalidades del SpinBox, añade un constructor típico y reimplementa tres funciones virtuales de **QSpinBox**. Además, añade una variable de tipo **QRegExpValidator** para regular la entrada de datos como se verá posteriormente. (Buscar en Help)

Solución HexSpinBox:

- Implementación:

```
#include <QtGui>
#include "hexspinbox.h“

HexSpinBox::HexSpinBox(QWidget *parent) : QSpinBox(parent)
{
    setRange(0, 255);
    validator = new QRegExpValidator(QRegExp("[0-9A-Fa-f]{1,8}"), this);
}
```

- Se pone como rango por defecto el intervalo 0-255 (0x00 – 0xFF) en vez del que trae **QSpinBox** que es de 0-99.
- La variable privada **validator** permitirá validar si las entradas en el editor de líneas del spin box son válidas. Para ello se usa la clase **QRegExpValidator** que acepta entre 1 y 8 caracteres, cada uno en el conjunto 0-9, A-F, a-f.

Solución HexSpinBox:

- Implementación:

```
QValidator::State HexSpinBox::validate(QString &text, int &pos) const
{
        return validator->validate(text, pos);
}
```

- Esta función es llamada por el **QSpinBox** para verificar si el texto introducido es válido. Ella puede devolver tres posibles valores: **Invalid** (el texto no coincide con la expresión definida), **Intermediate** (el texto tiene una parte válida) o **Acceptable** (el texto es válido). La función **QRegExpValidator::validate()** nos permite verificar si la entrada cumple con la regla puesta y por tanto devolvemos el resultado de la llamada a esta.

Solución HexSpinBox:

- Implementación:

```
QString HexSpinBox::textFromValue(int value) const
{
        return QString::number(value, 16).toUpper();
}
```

- La función **QSpinBox::textFromValue()** convierte un valor entero en un string. El **QSpinBox** la llama para actualizar el texto del editor de líneas, cuando el usuario pulsa la flecha ascendente o descendente.

Solución HexSpinBox:

- Implementación:

```
int HexSpinBox::valueFromText(const QString &text) const
{
        bool ok;
        return text.toInt(&ok, 16);
}
```

- La función **valueFromText()** realiza la conversión inversa, de string a valor entero. El **QSpinBox** la llama cuando el usuario introduce un valor en el editor de líneas y pulsa Enter. Aquí se usa la función **QString::toInt()** para convertir un texto a un valor entero usando base 16. Si el texto no es un valor hexadecimal válido la variable ok toma valor falso y la función **toInt()** devuelve valor 0. No obstante, dicha posibilidad no se ha tenido en cuenta pues el validador solo permite valores hexadecimales válidos.

Solución HexSpinBox:

- Integración del widget creado con Qt.

- Existen 2 formas integrar los widgets personalizados con Qt:

1. Usando la Promoción.
2. Usando Plugins. (Esto no es objetivo de esta asignatura aunque si lo desea lo puede encontrar en el segundo libro de la bibliografía capítulos 5 y 19)

Solución HexSpinBox:

- Integración del widget creado con Qt.

- El método de la Promoción es el más rápido y fácil. Se basa en el uso de un widget visual de la paleta de widgets que coincida con el widget del cual hemos heredado nuestro nuevo widget personalizado.

- Pasos que se deben dar para que **HexSpinBox** pueda ser utilizado:

1. Arrastrar un **QSpinBox** a la ventana usando el IDE.
2. Dar click derecho encima de dicho widget y seleccionar **Promote to…** en el menú contextual.
3. Escribir en la ventana de diálogo que se abre: **HexSpinBox** como nombre de la nueva clase y **hexspinbox.h** como fichero de encabezamiento.

Solución HexSpinBox:

- Integración del widget creado con Qt.

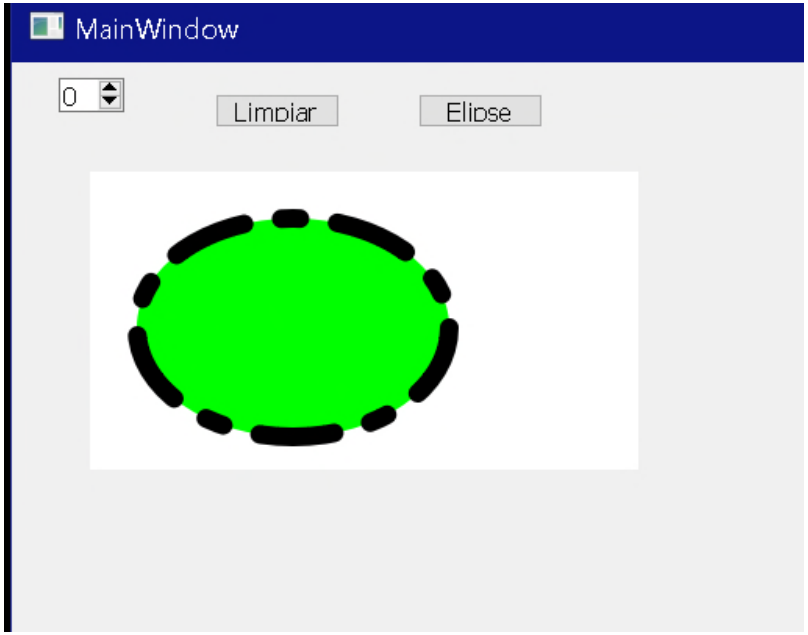
- De esta forma el nuevo **HexSpinBox** está representado en el IDE mediante su antecesor **QSpinBox**, pero tiene todas las nuevas funcionalidades.

Creando widgets como subclases directas de QWidget.:

- Cuando se quiere crear un nuevo widget y ninguno de Qt nos satisface para las funcionalidades que queremos lograr y cuando no hay manera de combinarlos o adaptar los widgets existentes para lograr el resultado deseado, aún existe una forma para hacerlo.
- Esto se puede lograr creando el widget como una subclase de QWidget y reimplementando algunos manejadores de eventos para pintar el widget y responder a los eventos del mouse.
- Esta forma de crearlo nos da completa libertad para definir y controlar la apariencia de nuestros widgets (a diferencia de la metodología explicada anteriormente).

Problema a resolver 2:

- Crear un widget Para dibujar primitivas fijas con botones.



Problema a resolver 2:

```
#include <QWidget>
#include <QImage>
#include <QPainter>

class Papel : public QWidget
{
    Q_OBJECT
public:
    explicit Papel(QWidget *parent = nullptr);
    void Limpiar();
    void elipse();
protected:
    void paintEvent(QPaintEvent *event);
    void resizeEvent(QResizeEvent *event);
signals:
private:
    QImage Img;
};
```

Problema a resolver 2:

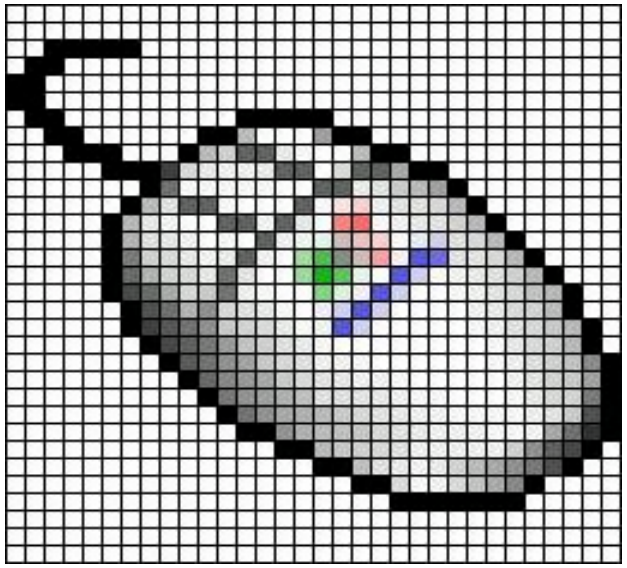
```
Papel::Papel(QWidget *parent) : QWidget(parent)
{
    setAttribute(Qt::WA_StaticContents);
}
void Papel::Limpiar()
{
    Img.fill(qRgb(255,   255, 255));
    update();
}
void Papel::elipse()
{
    QPainter painter(&Img);
    painter.setRenderHint(QPainter::Antialiasing, true);
    painter.setPen(QPen(Qt::black, 12, Qt::DashDotLine, Qt::RoundCap));
    painter.setBrush(QBrush(Qt::green, Qt::SolidPattern));
    painter.drawEllipse(30, 30, 200, 140);
    update();
}
```

Problema a resolver 2:

```
void Papel::paintEvent(QPaintEvent *event)
{
    QPainter painter(this);
    painter.drawImage(QPoint(0, 0), Img);
}
void Papel::resizeEvent(QResizeEvent *event)
{
  if  (width()   > Img.width()   ||  height()   > Img.height()) {
  int newWidth = qMax(width(), Img.width());
  int newHeight = qMax(height(),   Img .height());
  if   (Img.size()   != QSize(newWidth, newHeight)) {
    QImage newImage(QSize(newWidth, newHeight), QImage::Format_RGB32);
    newImage.fill(qRgb(255,  255, 255));
    QPainter painter(&newImage);
    painter.drawImage(QPoint(0,   0),Img);
    Img = newImage;
  }
update();
}
QWidget::resizeEvent(event);   }
```

Problema a resolver 3:

- Crear un widget editor de iconos al cual llamaremos IconEditor.



Solución Problema 2:

- Comencemos por la definición de la clase correspondiente a nuestro widget.

```
#include <QColor>
#include <QImage>
#include <QWidget>
class IconEditor : public QWidget{
    Q_OBJECT
    Q_PROPERTY(QColor penColor READ penColor WRITE setPenColor)
    Q_PROPERTY(QImage iconImage READ iconImage WRITE setIconImage)
    Q_PROPERTY(int zoomFactor READ zoomFactor WRITE setZoomFactor)
public:
    IconEditor(QWidget *parent = 0);
    void setPenColor(const QColor &newColor);
    QColor penColor() const { return curColor; }
    void setZoomFactor(int newZoom);
    int zoomFactor() const { return zoom; }
    void setIconImage(const QImage &newImage);
    QImage iconImage() const { return image; }
    QSize sizeHint() const;
```

Solución Problema 2:

- La clase **IconEditor** usa la macro **Q_PROPERTY** para declarar tres propiedades: penColor, iconImage y zoomFactor. Cada propiedad tiene una función de lectura y una función que puede ser opcional de escritura. Una propiedad es un campo especial de la clase al cual se puede acceder desde **Qt Designer**. La macro **Q_OBJECT** debe incluirse en la clase cuando usamos propiedades.

Solución Problema 2(continuación):

- Continuación de la definición de la clase correspondiente a nuestro widget.

```
protected:
    void mousePressEvent(QMouseEvent *event);
    void mouseMoveEvent(QMouseEvent *event);
    void paintEvent(QPaintEvent *event);
private:
    void setImagePixel(const QPoint &pos, bool opaque);
    QRect pixelRect(int i, int j) const;
    QColor curColor;
    QImage image;
    int zoom;
};
```

- Esta clase reimplementa tres funciones protegidas de **QWidget** y tiene algunas funciones y variables privadas. Las tres variables privadas almacenan los valores de las tres propiedades.

Solución Problema 2(continuación):

- La implementación de la clase comienza por el constructor:

```
#include <QtGui>
#include "iconeditor.h“
IconEditor::IconEditor(QWidget *parent):QWidget(parent)
{
    setAttribute(Qt::WA_StaticContents);
    setSizePolicy(QSizePolicy::Minimum, QSizePolicy::Minimum);
    curColor = Qt::black;
    zoom = 8;
    image = QImage(16, 16, QImage::Format_ARGB32);
    image.fill(qRgba(0, 0, 0, 0));
}
```

Solución Problema 2(continuación):

- Las variables que se inicializan en el constructor son:
  - curColor (color del pincel): asignada con color negro.
  - zoom (factor de zoom): inicializada a valor 8, lo que significa que cada pixel del icono será representado por un cuadrado de 8x8 pixels.
  - image (imagen del editor de iconos): inicializada con un **QImage** de tamaño 16x16 y formato ARGB (formato que soporta semi-trasparencia) de 32 bis de profundidad.

Solución Problema 2(continuación):

```
QSize IconEditor::sizeHint() const
{
    QSize size = zoom * image.size();
    if (zoom >= 3)
        size += QSize(1, 1);
    return size;
}
```

La función sizeHint () se vuelve a implementar desde QWidget y devuelve el tamaño ideal de un widget. Aquí, tomamos el tamaño de la imagen multiplicado por el factor de zoom, con un píxel adicional en cada dirección para acomodar una cuadrícula si el factor de zoom es 3 o más. (No mostramos una cuadrícula si el factor de zoom es 2 o 1, porque la cuadrícula apenas dejaría espacio para los píxeles del icono).

Solución Problema 2(continuación):

```
void IconEditor::setPenColor(const QColor &newColor)
{    curColor = newColor;  }
void IconEditor::setIconImage(const QImage &newImage)
{
   if (newImage != image) {
      image = newImage.convertToFormat(QImage::Format_ARGB32);
      update();
      updateGeometry();
   }
}
```

Después de configurar la variable de imagen, llamamos a QWidget :: update () para forzar un repintado del widget usando la nueva imagen. A continuación, llamamos a QWidget :: updateGeometry () para indicarle a cualquier diseño que contenga el widget que la sugerencia de tamaño del widget ha cambiado.

Solución Problema 2(continuación):

El resto de las funciones se pueden encontrar en el segundo libro de la bibliografía, capítulo 5.

Para ver las propiedades en el editor gráfico debe ser importado como plugin no promocionado.

CONCLUSIONES