PROGRAMACIÓN APLICADA A LA AUTOMATIZACIÓN

M.Sc. Alexander Prieto León

Conferencia XI

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

3.3 Elementos de Gráficos 2D.

Objetivos:

- Usar los elementos básicos que se utilizan en Qt para el manejo de gráficos 2D.
  - Explicar los conceptos básicos relativos a imágenes o gráficos en la pantalla.
  - Enunciar básicamente los objetos para el manejo de imágenes en Qt y sus principales métodos.
  - Enunciar básicamente el objeto principal de Qt para dibujar y sus principales métodos.
- Usar ventanas de diálogo personalizadas.

➢ 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:

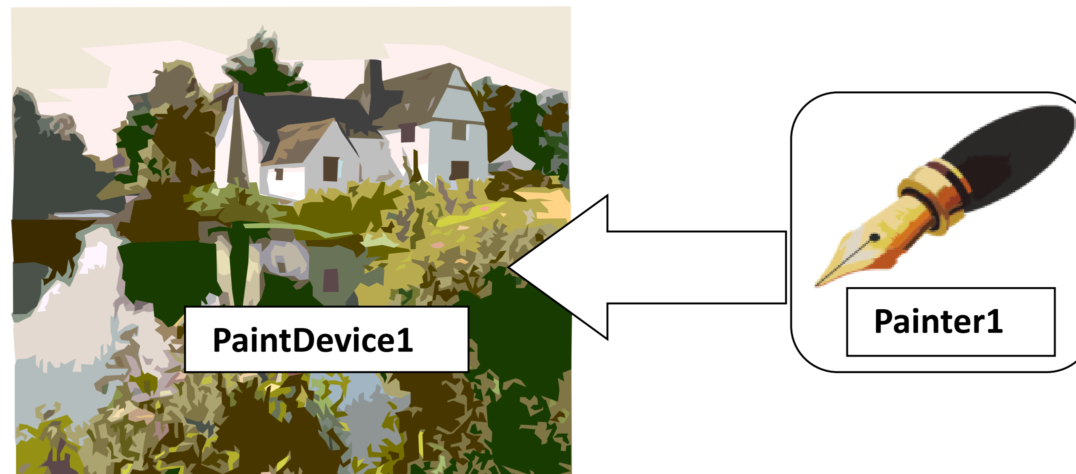
- ¿Para qué se utiliza el mecanismo Signal & Slot? ¿Cuáles son sus principales elementos?

FIEC
FACULTAD DE INGENIERÍA EN ELECTRICIDAD Y COMPUTACIÓN

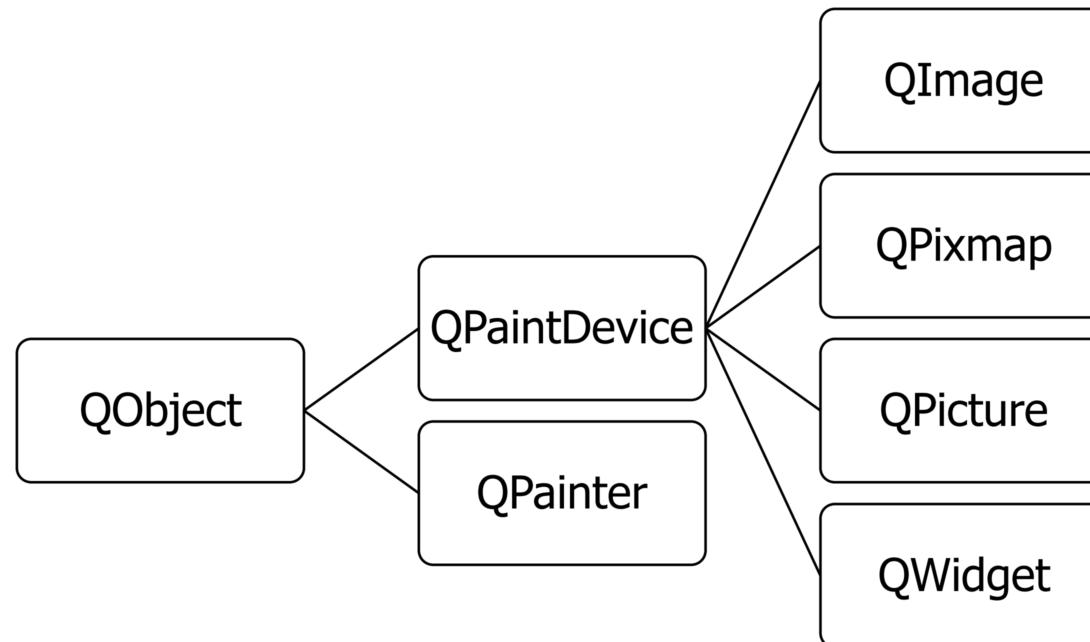
- ¿Qué software han utilizado para dibujar o editar imágenes?

- ¿Creen que pueden hacer un software similar en Qt?

- El manejo de imágenes y dibujos en Qt se basa en el uso de dos objetos básicos: el dispositivo de dibujo (Paint Device) que sería el que almacenaría la imagen o dibujo con que se esté trabajando, y un objeto que se encargaría de dibujar o modificar el dibujo o imagen (Painter).



- Jerarquía de las clases fundamentales para manejar imágenes y dibujar en Qt:



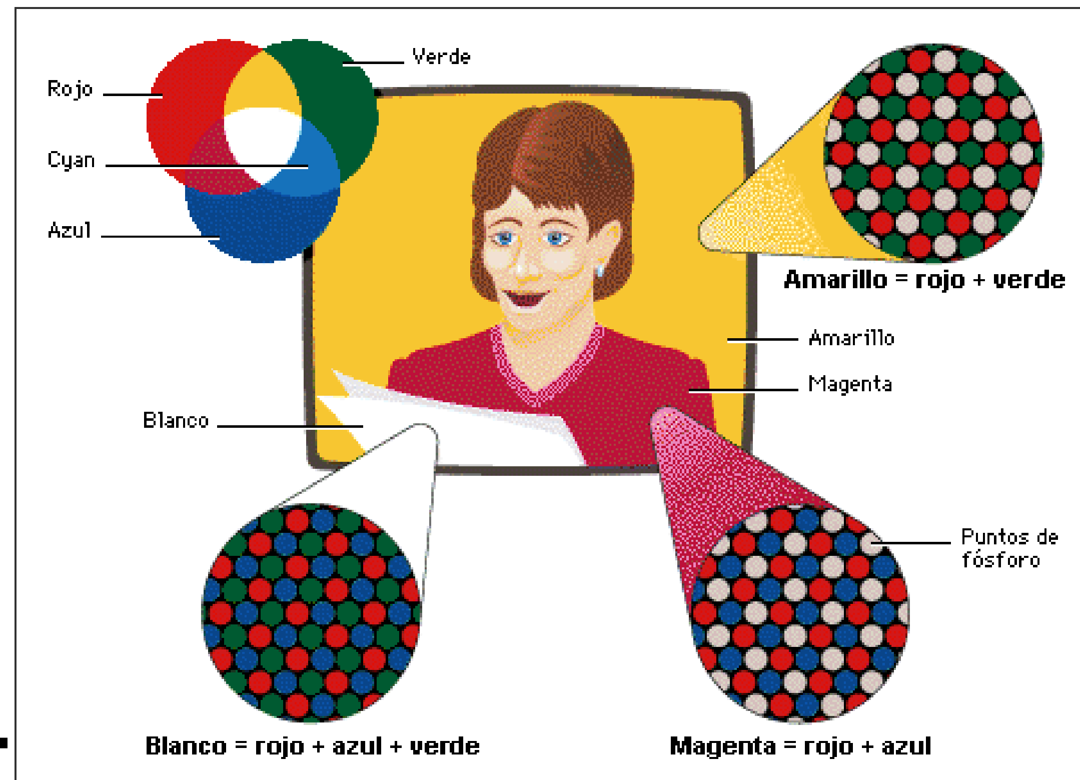
- **QPaintDevice:** es la abstracción de un espacio bidimensional donde se puede dibujar mediante un objeto de **QPainter**. Es la clase base de los objetos que pueden ser pintados en Qt.
- **QPainter:** se usa para llamar a las primitivas de dibujo: puntos, líneas, rectángulos, elipses, arcos, polígonos, curvas de Bézier, pixmaps, imágenes, texto, etc. Puede usarse sobre todos los objetos derivados de **QPaintDevice**.
- **QImage**: está diseñada y optimizada para operaciones de entrada/salida, y para acceso y manipulación directa a nivel de pixel.
- **QPixmap**: está diseñada y optimizada para mostrar imágenes en pantalla.
- **QPicture**: es un dispositivo de dibujo que permite almacenar y reproducir comandos de **QPainter**.
- **QWidget**: es la clase base de todos los objetos de la interfaz de usuario de Qt.

➢ El Sistema Coordenado en Qt.

- El sistema coordenado por defecto que tendrán los objetos derivados de **QPaintDevice** tiene su origen en la esquina superior izquierda. El valor horizontal ó **x** se incrementa hacia la derecha, y el vertical o **y** se incrementa hacia abajo. La unidad que se utiliza por defecto en este sistema coordenado es el **pixel** que es el elemento más pequeño que el monitor puede manipular modificando su color.

Síntesis aditiva de color de un píxel.

- Se llama síntesis aditiva del color a la reproducción de un color cualquiera mezclando cantidades adecuadas de sólo otros tres llamados aditivos primarios: rojo, azul y verde (se conoce más en inglés como Red-Green-Blue o **RGB,** y en Qt sería **QRgb** la clase que maneja este tipo de información de color). De esta forma se producen los colores que forman las imágenes de un monitor.



Clase QImage.

- Por defecto Qt soporta los siguientes formatos:

Format	Description	Qt's support
BMP	Windows Bitmap	Read/write
GIF	Graphic Interchange Format (optional)	Read
JPG	Joint Photographic Experts Group	Read/write
JPEG	Joint Photographic Experts Group	Read/write
PNG	Portable Network Graphics	Read/write
PBM	Portable Bitmap	Read
PGM	Portable Graymap	Read
PPM	Portable Pixmap	Read/write
TIFF	Tagged Image File Format	Read/write
XBM	X11 Bitmap	Read/write
XPM	X11 Pixmap	Read/write

Clase QImage.

- Una imagen puede ser cargada de un fichero con el constructor o con **QImage::load()**.
  - El fichero podrá ser un fichero normal (cargado en tiempo de ejecución) o uno leído con el sistema de recursos (en tiempo de compilación).
- Una imagen puede ser grabada a disco con **QImage::save()**.
- **QImage::height()**: retorna la altura de la imagen en pixels.
- **QImage::width()**: retorna el ancho de la imagen en pixels.
- Las funciones para manipular una imagen dependen del formato (**QImage::Format**) usado al crear el objeto **QImage**.

Clase QImage.

- Las imágenes de 32 bit usan valores ARGB.

	0xff7aa327	
0xff7aa327	0xffbd9527	0xffedba31

```
QImage image(3, 3, QImage::Format_RGB32);
QRgb value;

value = qRgb(189, 149, 39); // 0xffbd9527
image.setPixel(1, 1, value);

value = qRgb(122, 163, 39); // 0xff7aa327
image.setPixel(0, 1, value);
image.setPixel(1, 0, value);

value = qRgb(237, 187, 51); // 0xffedba31
image.setPixel(2, 1, value);
```

32-bit

Clase QImage.

- Cada valor de pixel (32 bits) se descompone en 8 bits para la intensidad de rojo, 8 para verde y 8 para azul, y 8 para nivel de trasparencia (el componente alpha u opacidad).
- Por ejemplo: El color rojo puro se representará con:

  QRgb red = qRgba(255, 0, 0, 255);
- o bien con:

  QRgb red = qRgb(255, 0, 0);
- o bien con:

  QRgb red = 0xFFFF0000;
- (la función contraria a **QImage::setPixel()** es **QImage::pixel(int x, int y)** que retorna el color del pixel dado por la posición x,y)

➢ Clase QImage.

- Las imágenes monocromo y 8 bits usan valores basados en índices de la paleta color.

	0	
0	2	1

0	0xff7aa327
1	0xffedba31
2	0xffbd9527

```
QImage image(3, 3, QImage::Format_Indexed8);
QRgb value;

value = qRgb(122, 163, 39); // 0xff7aa327
image.setColor(0, value);

value = qRgb(237, 187, 51); // 0xffedba31
image.setColor(1, value);

value = qRgb(189, 149, 39); // 0xffbd9527
image.setColor(2, value);

image.setPixel(0, 1, 0);
image.setPixel(1, 0, 0);
image.setPixel(1, 1, 2);
image.setPixel(2, 1, 1);
```

8-bit

En este caso el valor del pixel es un índice en la tabla (paleta) de color de la imagen.

Clase QImage.

- **QImage::transformed**(**const QMatrix**& matrix): retorna una copia de la imagen transformada según la transformación almacenada en matrix.
  - **QMatrix**: es una clase que permite la creación de objetos que modelan utilizando una matriz de dimensión 3x3 las diferentes transformaciones que se le apliquen a un sistema coordenado. Por ejemplo:

```
QImage Imagen1;
Imagen1.load("c://photo.jpg");
QMatrix matrix;
matrix.translate(50,50);
matrix.rotate(45.0);
matrix.scale(0.5,1.0);
Imagen1.transformed(matrix);
```

➢ Clase QPixmap.

- Una imagen puede ser cargada de un fichero con el constructor o bien con **QPixmap::load()**.
- Un pixmap puede crearse con uno de los constructores o con las funciones estáticas:
  - **grabWidget()**: Crea un pixmap con el contenido capturado de un widget.
  - **grabWindow()**: Crea un pixmap con el contenido capturado de una ventana.
- Un **QPixmap** puede mostrarse en pantalla con un **QLabel** o alguna de las subclases de **QAbstractButton** (como **QPushButton** y **QToolButton**):
  - **QLabel** tiene la propiedad **pixmap** y las funciones de acceso **pixmap()** y **setPixmap().**
  - **QAbstractButton** tiene la propiedad **icon** (**QIcon**) y las funciones de acceso **icon()** y **setIcon().**

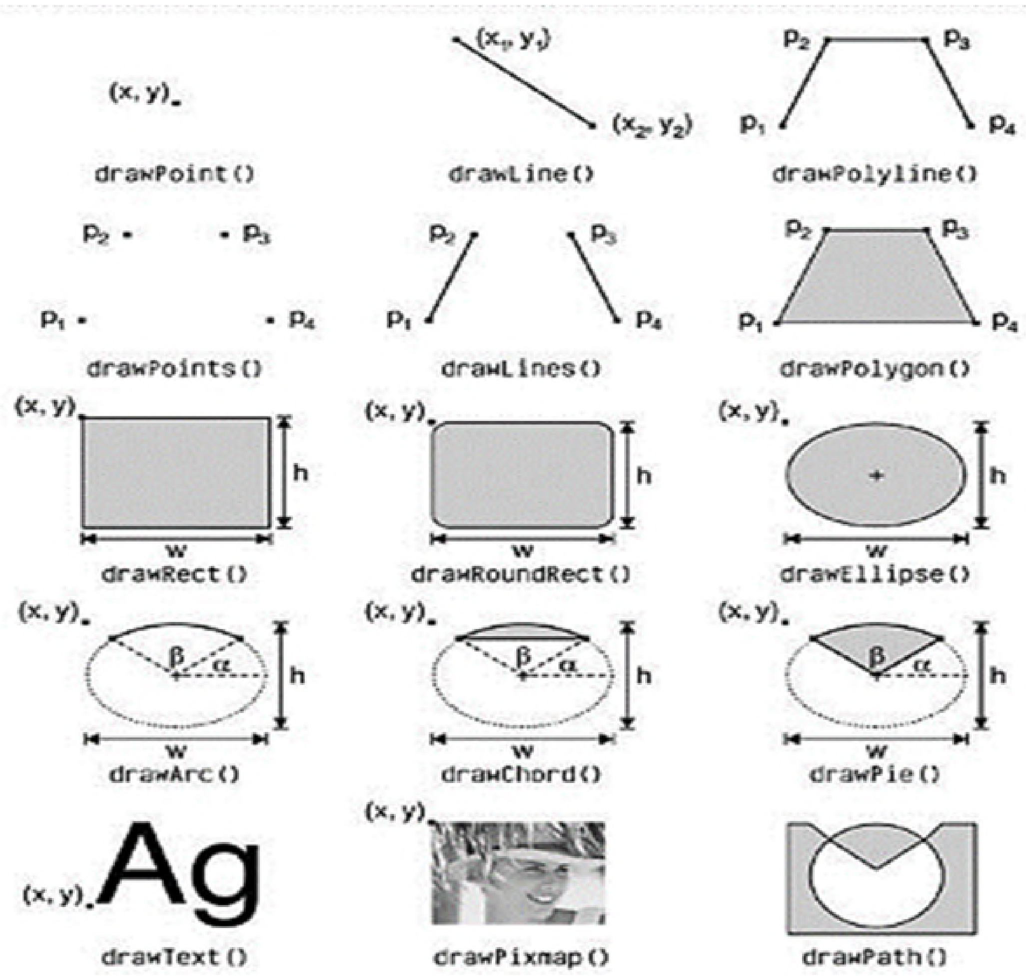
➢ Clase QPixmap.

- Los datos de cada pixel sólo pueden ser accedidos a través de funciones de la clase **QPainter** o convirtiendo el **QPixmap** en un **QImage**. En un pixmap los datos de cada pixel son datos internos manejados por el correspondiente manejador de ventanas (o servidor X).
    - Un **QPixmap** puede convertirse en **QImage** con **QPixmap::toImage().**
    - Un **QImage** puede convertirse en **QPixmap** con **QPixmap::fromImage().**

➢ **Nota**: la clase **QPicture** se deja de estudio individual.

➢ Dibujar con QPainter.

- Con el objeto **QPainter** podemos dibujar varias formas o primitivas tal y como se ve en la figura.

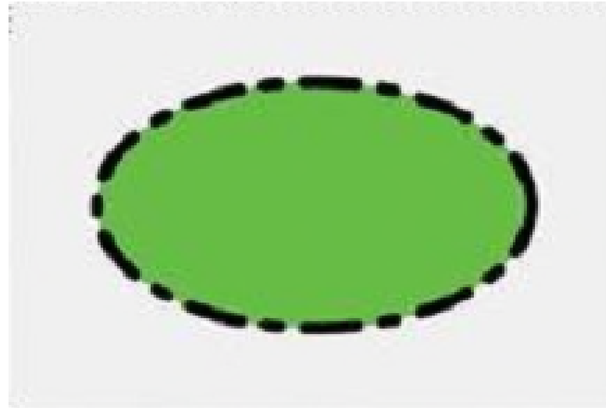


➢ Dibujar con QPainter.

- El resultado de cada primitiva gráfica depende de los atributos del **QPainter**. Los más importantes son:
  - **Pincel (pen)**: Se usa para dibujar líneas y bordes de formas (rectángulos, elipses, etc). Consiste de un color (**QColor**), una anchura, estilo de línea, **cap style** y **join style**.
  - **Brocha (brush)**: Patrón usado para rellenar formas geométricas. Consiste normalmente de un color y un estilo, pero puede ser también una textura (pixmap que se repite infinitamente) o un gradiente.
  - **Font**: Se usa al dibujar texto. Contiene muchos atributos tal como la familia y el tamaño de punto.
- Tales atributos pueden modificarse en cualquier momento con **setPen()**, **setBrush()** y **setFont()** con objetos de **QPen**, **QBrush** o **QFont**.

Dibujar con QPainter.

- Ejemplo 1: Dibujar:



```
QPainter painter(this);
painter.setRenderHint(QPainter::Antialiasing, true);
painter.setPen(QPen(Qt::black, 12, Qt::DashDotLine, Qt::RoundCap));
painter.setBrush(QBrush(Qt::green, Qt::SolidPattern));
painter.drawEllipse(80, 80, 400, 240);
```

- La llamada a **setRenderHint()** activa el antialiasing, que hace que se usen diferentes intensidades de color en las fronteras para reducir la distorsión visual al convertir las fronteras de una figura en pixels.

Dibujar con QPainter.

- Ejemplo 2: Dibujar:

```
QPainter painter(this);
painter.setRenderHint(QPainter::Antialiasing, true);
painter.setPen(QPen(Qt::black, 15, Qt::SolidLine, Qt::RoundCap,
                    Qt::MiterJoin));
painter.setBrush(QBrush(Qt::blue, Qt::DiagCrossPattern));
painter.drawPie(80, 80, 400, 240, 60 * 16, 270 * 16);
```

Dibujar con QPainter.

- Ejemplo 3: Dibujar:

```
QPainter painter(this);
painter.setRenderHint(QPainter::Antialiasing, true);
QPainterPath path;
path.moveTo(80, 320);
path.cubicTo(200, 80, 320, 80, 480, 320);
painter.setPen(QPen(Qt::black, 8));
painter.drawPath(path);
```

- La clase **QPainterPath** permite especificar un dibujo vectorial mediante la unión de varios elementos gráficos básicos: líneas rectas, elipses, polígonos, arcos, curvas cuadráticas o de Bezier, y otros objetos **QPainterPath.**El objeto **QPainterPath** especifica el borde de una figura, que puede rellenarse si usamos una brocha en el **QPainter**.

Dibujar con QPainter.

- **Brocha del background**: Es la brocha usada al dibujar texto opaco, líneas tipo stippled y bitmaps. Esta brocha no tiene efecto en modo de background transparente (el de por defecto).
  - **setBackgroundBrush**(**QBrush**)
  - **setBackgroundMode**(**BGMode**): Establece el modo de background (**Qt::TransparentMode**, **Qt::OpaqueMode**)
- **Origen de la brocha**: Punto inicial para patrones de rellenado con la brocha (esquina superior izquierda normalmente).
- **Máscara de recorte** (**clip region**): Área del dispositivo que se afectará por las primitivas gráficas.
- **Viewport**, **window** y **world matrix**: Determinan cómo transformar las coordenadas del **QPainter** en coordenadas físicas del dispositivo.

➢ Dibujar con QPainter.

- **Modo de composición**: Especifica cómo combinar los píxeles existentes con los que se van a dibujar.
  - **setCompositionMode(CompositionMode)**

Source | Destination | SourceOver | DestinationOver | SourceIn | DestinationIn

SourceOut | DestinationOut | SourceAtop | DestinationAtop | Clear | Xor

Dibujar con QPainter.

Transformaciones.

- Las transformaciones sobre un dibujo no son más que desplazamientos o rotaciones que puede sufrir el mismo. Para realizar estas es necesario primero conocer el sistema de coordenadas en que se realizan los dibujos.
  - El sistema de coordenadas por defecto de QPainter tiene su origen (0; 0) en la esquina superior izquierda. Las coordenadas x se incrementan hacia la derecha y las y hacia abajo.
  - Cada pixel ocupa un área de tamaño 1x1.

➢ Dibujar con QPainter.

- Es posible modificar el sistema de coordenadas por defecto usando el los mecanismos **viewport, window** y **world matrix**.

  - ❖ El mecanismo **viewport** es un rectángulo que especifica las coordenadas físicas, mientras que el mecanismo window especifica el mismo rectángulo pero en coordenadas lógicas. Cuando dibujamos con una primitiva gráfica, se usan coordenadas lógicas, que se trasladan en coordenadas físicas usando el **viewport** y **window** actuales.

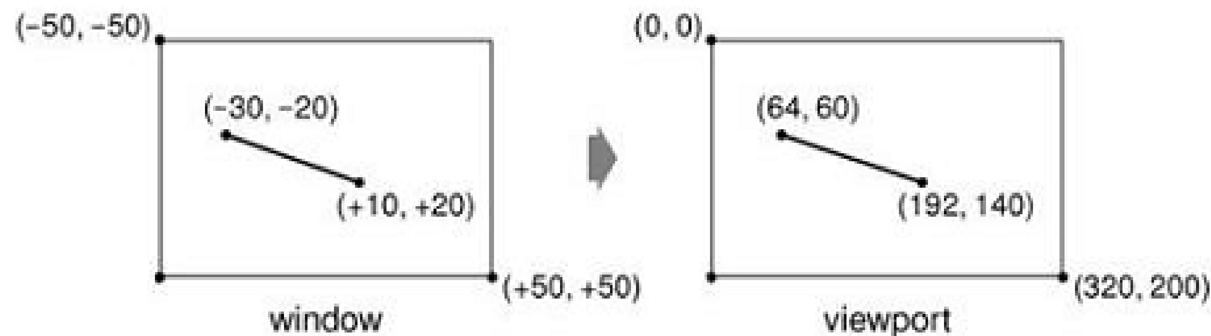
  - ❖ Por defecto, ambos rectángulos coinciden con el rectángulo del dispositivo de dibujo. Ejemplo: Si el dispositivo de dibujo es un widget de 320x200, el **viewport** y **window** son también un rectángulo de 320x200 con su esquina superior izquierda en la posición (0; 0). En este caso, el sistema de coordenadas físico y lógico es el mismo.

Dibujar con QPainter.

- Este sistema hace que el código para dibujar pueda hacerse independiente del tamaño o resolución del dispositivo de dibujo. Por ejemplo, en el caso de antes, podríamos definir el sistema de coordenadas lógicas en el rango (-50; 50) a (50; 50) siendo (0; 0) el centro. Esto se hace mediante:

```
painter.setWindow(-50, -50, 100, 100);
```

- El (-50; -50) especifica el origen y el par (100, 100) especifica el ancho y alto. Esto significa que la coordenada lógica (-50; -50) ahora corresponde con la coordenada física (0; 0) y la coordenada lógica (50; 50) corresponde con la física (320; 200). En este caso no se ha cambiado el sistema de coordenadas físico o **viewport**.



Dibujar con QPainter.

- Por otro lado el mecanismo **world matrix** es una matriz de trasformación que es aplicada adicionalmente además de los mecanismos **viewport** y **window.** Esta matriz permite hacer las siguientes trasformaciones a los items que dibujamos: trasladar, escalar, rotar y girar.

- **Por ejemplo** si se quiere dibujar un texto en un ángulo de 45º se puede usar el siguiente código:

```
QMatrix matrix;
matrix.rotate(45.0);
painter.setMatrix(matrix);
painter.drawText(rect, Qt::AlignCenter, tr("Revenue"));
```

Dibujar con QPainter.

- Por otro lado el mecanismo **world matrix** es una matriz de trasformación que es aplicada adicionalmente además de los mecanismos **viewport** y **window.** Esta matriz permite hacer las siguientes trasformaciones a los items que dibujamos: trasladar, escalar, rotar y girar. **Por ejemplo** si se quiere dibujar un texto en un ángulo de 45º se puede usar el siguiente código:

```
QMatrix matrix;

matrix.rotate(45.0);

painter.setMatrix(matrix);

painter.drawText(rect, Qt::AlignCenter, tr("Revenue"));
```

Las coordenadas lógicas que son pasadas a través de drawText() son transformadas por el world matrix y entonces se mapea al sistema de coordenadas físicas usando la configuración de window-viewport.

Dibujar con QPainter.

- Si especificamos varias trasformaciones, se aplicarán en el orden que las demos. Por ejemplo si queremos usar el punto (10; 20) como punto de rotación, podemos trasladar primero el sistema de referencia **window** (inicialmente está en (0; 0)), hacer la rotación y trasladar el sist. Ref. window de nuevo a su posición original:

```
QMatrix matrix;
matrix.translate(-10.0, -20.0);
matrix.rotate(45.0);
matrix.translate(+10.0, +20.0);
painter.setMatrix(matrix);
painter.drawText(rect, Qt::AlignCenter, tr("Revenue"));
```

➢ Dibujar con QPainter.

➢ Una forma más simple de especificar trasformaciones es usando los métodos translate(), scale(), rotate(), y shear() de QPainter.

```
painter.translate(-10.0, -20.0);
painter.rotate(45.0);
painter.translate(+10.0, +20.0);
painter.drawText(rect, Qt::AlignCenter, tr("Revenue"));
```

➢ Pero si queremos usar varias veces las mismas trasformaciones, es más eficiente almacenarlas en un objeto **QMatrix** y definir la **window matrix** del objeto **QPainter** cuando se necesiten tales trasformaciones.

- **Ventanas de diálogo propias**

- **Ejemplo:** Realice una aplicación que llame a ventanas de diálogo personalizadas modales y no modales.

CONCLUSIONES

- Para manejar gráficos 2D en Qt se utilizan dos objetos fundamentales el Painter y el Paint Device (QImage, QPixmap, QPicture, QWidget).
- El método QImage::setPixel() permite definir el color que tendrá un pixel específico del Paint Device.
- Los métodos de los objetos de QPainter permiten dibujar de múltiples formas y realizar transformaciones a esos dibujos.