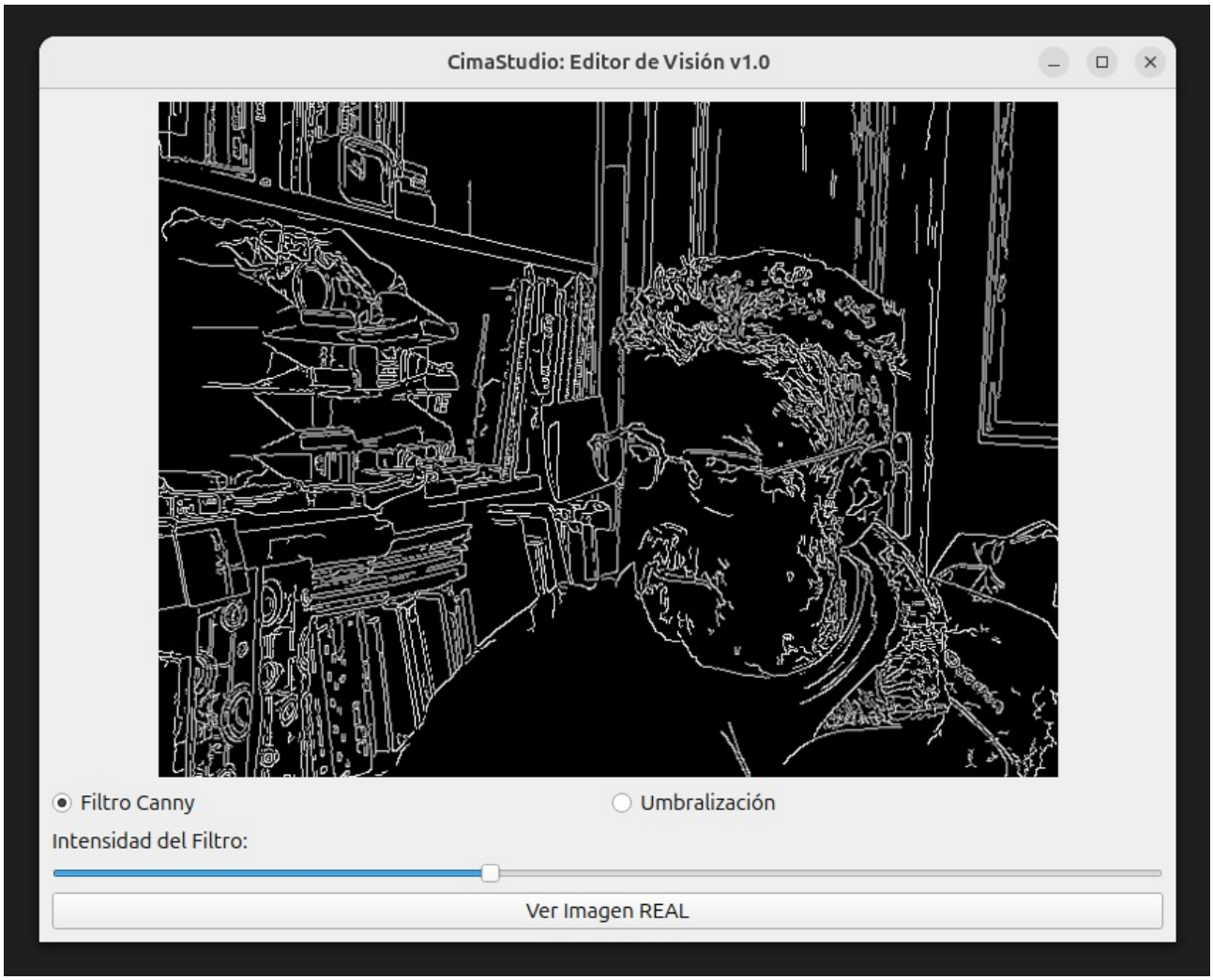
< >

CimaStudio



C++ . OpenCV . Qt

Sumario

1.- Introducción: El arte de enseñar a ver a las máquinas

Bienvenido a CimaStudio. Lo que tienes ante ti no es solo un programa informático; es un laboratorio digital donde aprenderás a capturar la luz y transformarla en información. CimaStudio es una aplicación desarrollada en C++ que utiliza la potencia de la librería OpenCV para el procesamiento de imágenes y el framework Qt6 para la creación de una interfaz de usuario profesional.

A través de una interfaz sencilla, el usuario puede conmutar en tiempo real entre la visualización de la webcam en crudo o la aplicación de filtros avanzados como el algoritmo de bordes Canny

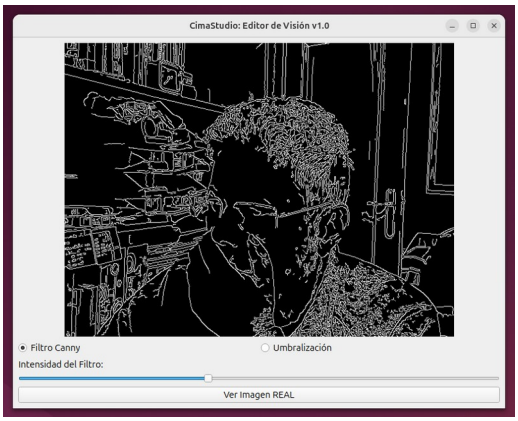
Interfaz de CimaStudio

Para analizar el programa y hacernos con el control total de las riendas utilizaremos ingenieria inversa. Partiremos del programa terminado y arrancaremos un sofisticado proceso de marcha atrás para dejar al descubierto las entrañas de la aplicación. En la analogía que seguiremos en el tutorial acompañaremos la luz en su viaje desde el hardawar (la camara) hasta la imagen implementada en los leds del monitor.

El viaje de la luz: De la física al código

Para tomar el control total de esta herramienta, utilizaremos la ingeniería inversa. No nos limitaremos a leer código; vamos a desarmar la aplicación para entender cómo la luz del mundo físico se convierte en conocimiento visual.

> *En CimaStudio, no solo estamos escribiendo líneas de código; estamos diseñando el cauce de un río de información.*



Interfaz de usuario de CimaStudio

El viaje comienza cuando la luz rebota en un objeto y entra por la lente de tu cámara. Allí, un sensor traduce los fotones en impulsos eléctricos que el hardware digitaliza en una matriz de números. Es entonces cuando nuestro código entra en acción: tomamos esos datos crudos y los alojamos en la memoria RAM bajo la estructura de una **cv::Mat**. En ese instante, la realidad es puro cálculo

matemático. Aplicamos algoritmos, transformamos los datos y cruzamos el puente hacia Qt, convirtiendo la matriz en una **QImage**. Finalmente, el sistema operativo ordena a los LED de tu monitor iluminarse. Lo que tú ves como un "filtro", es en realidad el final de un viaje épico donde hemos gobernado la luz.

1.1.- Tu Puerta a la Visión Artificial: ¿Por qué estas tecnologías?

Para este viaje, hemos unido los tres pilares de la industria actual:

- **OpenCV**: El motor de visión más avanzado del mundo.
- **Qt**: El estándar profesional para interfaces gráficas robustas.
- **C++:** El lenguaje que nos da el control total sobre la velocidad y la memoria.

Aprenderás a manejar la Anatomía de los Bordes (Canny) para identificar formas y la Decisión Binaria (Umbralización) para separar objetos de su fondo. Prepárate para construir tu propio puente entre el mundo físico y el digital.

1.2.- Instalaciones necesarias

Es recomendable hacer todas las instalaciones que vamos a necesitar en este momento. Si en adelante necesitamos algo más ya lo especificaremos. Para Ubuntu/Debian tenemos que instalar Qt6, OpenCV y las herramientas de compilación. Para Windows y macOS daremos también instalaciones equivalentes sin la garantía de haberlas probado porque quien escribe el tutorial trabaja con Ubuntu 24.04, tanto la programación como la edición de textos. En Ubuntu 24.04 hemos probado las recetas de instalación que mencionamos a continuación.

Nota 1: Para Linux, Windows y macOS: Después de realizar las instalaciones que describimos a continuación, es importane pedir ayuda (**Google o IA**) para que nos facilite un método para averiguar si tenemos bien instaladas OpenCV y Qt.

Nota 2: Para Linux y macOS: Si los métodos que describimos abajo no consiguen instalar las librerías, nunca darse por vencidos. Pedir ayuda a la **IA** y os facilitará las instrucciones de instalación actualizadas. Seguro que es fácil.

1.2.1.- Instalaciones en Debian/Ubuntu (probadas en Ubuntu 24.04)

Es el sistema más sencillo de instalar, ya que todo está en los repositorios oficiales.

Qt6:

```
sudo apt update && sudo apt install qt6-base-dev qt6-declarative-dev
```

OpenCV (dede los paquetes oficiale precompiados)**:**

```
sudo apt install libopencv-dev
```

Herramientas de compilación:

```
sudo apt install build-essential cmake
```

Nota importante: la instalación de OpenCV desde los paquetes precompilados oficiales es muy sencilla pero no incluye las utilidades de **opencv_contrib** ni soporte para **CUDA**. Si quieres tener OpenCV completo en tu ordenador tienes que instalar compilando los fuentes. Es por supuesto un proceso más complejo que no requerimos para desarrollar este tutorial.

Todo lo referente a la instalación esta tratado en los tutoriales oficiales de OpenCV a los que puedes acceder desde aquí.

1.2.2.- Instalaciones en Windows

En Windows no hay un comando nativo, por lo que se utiliza un gestor de paquetes llamado vcpkg o instaladores directos.

Opción recomendada (vcpkg):

```
vcpkg install qt6:x64-windows
vcpkg install opencv:x64-windows
```

Opción manual: Descargar el Instalador de Qt y el ejecutable de OpenCV en SourceForge.

1.2.3.- macOS

Se utiliza Homebrew, que es el estándar de facto para desarrolladores en Mac.

Instalar ambos:

```
brew install qt@6
brew install opencv
```

Nota importante: En Mac, después de instalar Qt, hay que añadirlo al PATH (Homebrew te dará el comando exacto al finalizar la instalación).

1.3.- Despliegue Inmediato: Descarga, compilación y ejecución de CimaStudio

Para evitar problemas de compatibilidad entre diferentes versiones de Linux, Windows o macOS, no utilizaremos un ejecutable "precocinado". En su lugar, vamos a generar nuestro propio ejecutable a partir del código fuente. Esto garantiza que el programa se adapte perfectamente a tu ordenador.

Si eres usuario de windows o macOs y no has utilizado nunca un archivo **CmakeLists.txt**, tendrás que conseguir este archivo para tu sistema operativo. El que descargarás a continuación es válido para Linux pero no funcionará con tu S.O. No obstante, los pasos que daremos cuando expliquemos un poco más adelante la compilación y generación del ejecutable en Ubuntu 24.04, son los mismos en cualquier S.O.

1.3.1.- Paso 1: Clonar el repositorio

Abre una terminal y sitúate en la carpeta donde quieras trabajar. Escribe el siguiente comando para descargar todo el material:

```
git clone https://github.com/vsoriano58/CimaStudio.git
cd CimaStudio
```

1.3.2.- Paso 2: Compilación asistida (El poder de CMake)

Ahora dejaremos que el "Director de Orquesta" (nuestro archivo CMakeLists.txt) analice tu sistema y prepare la compilación. Solo necesitas tres comandos:

```
mkdir build && cd build   # Creamos una carpeta para compilar sin ensuciar el código
cmake ..                  # CMake busca Qt6 y OpenCV en tu sistema
make                      # El compilador genera el ejecutable CimaStudio
```

Comprueba primero que tu directorio contiene todos los ficheros del proyecto:

Ficheros del proyecto CimaStudio v1.0

CMakeLists.txt	Archivo de configuración que contiene las instrucciones para compilar el proyecto y gestionar sus dependencias.
cimastudio.h	Cabecera que define la interfaz de la clase CimaStudio, incluyendo sus atributos y prototipos de funciones.
cimastudio.cpp	Archivo de implementación donde se desarrolla la lógica y el funcionamiento de los métodos de la clase CimaStudio.
main.cpp	Punto de entrada del programa que instancia la clase principal y coordina la ejecución de la aplicación.

Mas adelante explicaremos la función de cada uno de ellos y analizaremos el viaje de los datos a través de los mismos.

1.3.2.1.- Automatizar la compilación: los alias

En el proceso de depuración, modificar y compilar es una tarea constante. A veces, los archivos temporales de una compilación anterior en el directorio /build provocan conflictos o errores inesperados. La solución más limpia es borrar el contenido de /build y regenerarlo antes de cada cambio importante.

Para evitar que este proceso sea tedioso, utilizaremos **alias** en la terminal de Ubuntu (en macOS funcionan igual, y en Windows existen alternativas como los doskey o scripts de PowerShell).

Definiendo los alias:

Si conoces el número de núcleos de tu procesador (ej. 4), puedes usar:

```
alias rmbuild='cd .. && rm -rf build && mkdir build && cd build && cmake ..'
alias make4='make -j4'
```

Si prefieres que el sistema lo detecte automáticamente para aprovechar el máximo de potencia disponible:

```
alias rmbuild='cd .. && rm -rf build && mkdir build && cd build && cmake ..'
alias makej='make -j$(nproc)'
```

a) El alias rmbuild

Realiza los siguientes pasos:

1. cd ..: Sale de build hacia la carpeta raíz del proyecto.
2. rm -rf build: Borra la carpeta de compilación anterior y todo su contenido.
3. mkdir build: Crea una carpeta build totalmente limpia.
4. cd build: Entra en la nueva carpeta.
5. cmake ..: Configura el proyecto basándose en el archivo CMakeLists.txt que está en el nivel superior.

b) El alias makej

- **nproc**: Comando que devuelve el número total de hilos de procesamiento disponibles en tu hardware GNU Coreutils
- **$(...)**: Ejecuta el comando interno y entrega su resultado al comando principal
- **-j**: Indica a make que realice la compilación en paralelo, reduciendo drásticamente el tiempo de espera GNU Make Manual

1.3.2.2.- El flujo de trabajo en la terminal

Una vez configurados, el ciclo de desarrollo en CimaStudio se vuelve sumamente ágil. Partiendo de que estás dentro de la carpeta build, el flujo es:

```
# Paso 1: Limpiar y re-configurar el proyecto
cima_project/build> rmbuild

# Paso 2: Compilar a máxima velocidad
cima_project/build> makej

# Paso 3: Ejecutar (puedes crear otro alias para esto si gustas)
cima_project/build> ./CimaStudio
```

Truco de productividad: Si trabajas en la terminal de VS Code o del sistema, usa la tecla flecha arriba. Podrás conmutar entre estos tres comandos rápidamente sin tener que teclear nada, centrando toda tu atención exclusivamente en el código.

Consejo del autor: Si por error ejecutas rmbuild fuera de la carpeta adecuada, el comando fallará al no encontrar el directorio build. Esto es una señal de que debes situarte en la carpeta de compilación antes de continuar.

1.3.3.- Paso 3: ¡Acción!

Si los pasos anteriores han finalizado sin errores, ya tienes tu versión de CimaStudio lista para funcionar. Ejecuta el programa desde la carpeta build en la que te encuentras mediante:

```
./CimaStudio
```

Al compilarlo tú mismo el programa, te aseguras de que utiliza exactamente las librerías que tienes instaladas. Si cmake te da algún error, es que te falta algún componente de Qt o OpenCV (revisa el apartado de Instalaciones).

Por qué este enfoque es mejor que proporcionar un ejecutable suelto:

- **Garantía de funcionamiento**: Si make termina con éxito, el programa funcionará al 100%. Un ejecutable descargado suele fallar por versiones de .so (librerías compartidas) distintas.
- **Aprendizaje real**: Sentirás siente que has "construido" el software. Ese pequeño éxito de ver cómo se compila el código es muy motivador.
- **Transparencia**: No hay "cajas negras". Todo lo que se ejecuta ha sido compilado delante de tus ojos.

2.- Anatomía Visual y Funcional de CimaStudio

En esta sección vamos a conocer no solo qué botones hay en la pantalla, sino cómo están conectados con el "cerebro" del programa.

2.1.- El Panel de Control: Lo que el usuario toca

La interfaz se divide en dos grandes áreas: el Visor de Realidad (donde se proyecta la luz procesada) y la Consola de Parámetros (donde el usuario ejerce su voluntad sobre el algoritmo).

(Aquí mantendrías tus descripciones de 2.1.1 a 2.1.4)

2.1.1.- El Visor Dinámico (viewLabel): Nuestra ventana al mundo procesado...

Es un lienzo inteligente basado en un QLabel.

Su función: Traducir los datos matemáticos de OpenCV a píxeles visibles.

Nota: Aunque es una etiqueta de texto, la usamos como proyector. Su propiedad **scaledContents** (o el escalado manual que hacemos) permite que la imagen de la cámara se adapte al tamaño de la ventana sin deformarse.

2.1.2.- Selector de Algoritmo: Los interruptores de decisión...

Estos controles actúan como un conmutador de caminos.

Filtro Canny: Activa el algoritmo de detección de bordes. Es ideal para mostrar cómo las matemáticas pueden encontrar "la silueta del mundo" basándose en los cambios de intensidad.

Umbralización (Threshold): Es el control más radical. Convierte la realidad en un mundo binario: blanco o negro, sin grises. Sirve para enseñar el concepto de "decisión": o el píxel supera el valor, o se descarta y se pinta en negro.

2.1.3.- El Deslizador de Intensidad (paramSlider): El control de grado de aplicación...

Es el control más táctil de la aplicación.

Rango (0-255): Representa el rango completo de un byte (8 bits).

Doble función:

- En Canny, ajusta la sensibilidad de los bordes.
- En Umbralización, define el "corte" donde la imagen se vuelve negra o blanca.

Conexión en tiempo real: Gracias al sistema de Signals & Slots de Qt, mover este slider actualiza la variable sliderValue instantáneamente, permitiendo al alumno ver el cambio en la imagen sin soltar el ratón.

2.1.4.- Interruptor de Realidad (btnToggleMode): El conmutador entre la cámara y el proceso...

Este botón es fundamental desde el punto de vista pedagógico.

Su función: Alternar entre la imagen procesada y la imagen real (el "feed" directo de la cámara).

Cómo actúa: Permite comparar el "antes" y el "después". Al pulsar este botón, el programa decide si aplica los algoritmos o si simplemente clona el frame original para mostrarlo tal cual.

💡 Un detalle técnico:

Puedes explicar que estos controles son "sensores de eventos". Mientras la cámara corre a 30 FPS, estos botones están "escuchando". En el momento en que el alumno mueve el slider, el programa captura ese nuevo valor y lo inyecta en el siguiente fotograma que la cámara captura.

Es el ejemplo perfecto de Interactividad en Tiempo Real.

2.2.- El Sistema de Signals & Slots: El sistema nervioso de la interfaz

¿Cómo sabe el programa que has movido el deslizador (2.1.3) y que debe cambiar el brillo de la imagen en tiempo real? Aquí es donde entra la magia de Qt.

Imagina que cada control visual es un emisor de radio y la lógica de nuestro código es un receptor:

- **Signals** (Señales): Cuando mueves el paramSlider, este "emite una señal" al aire diciendo: "¡Atención, mi valor ha cambiado a 50!".
- **Slots** (Ranuras/Receptores): En nuestro archivo cimastudio.cpp, tenemos funciones preparadas para escuchar. Un "Slot" es simplemente una función que se activa automáticamente cuando recibe una señal específica.
- **La Conexión:** Nosotros, como programadores, "tiramos un cable" (usando el comando connect) entre la señal del slider y el slot de procesamiento.

¿Por qué es especial?

Permite un acoplamiento débil. Esto significa que puedes cambiar el funcionamiento de un botón (el emisor) o de la lógica de guardado (el receptor) de forma independiente, siempre que el "cable" (la conexión) se mantenga. Es una alternativa mucho más segura y flexible a los callbacks tradicionales de C++.

> En resumen: Sin el sistema de Signals & Slots, los controles que hemos descrito en el punto 2.1 serían solo dibujos inertes en la pantalla. Gracias a este sistema, la interfaz y el algoritmo de visión artificial se dan la mano de forma instantánea.

2.2.1.- En nuestro código

(El código ya está en tu carpeta de proyecto. Esta es una de las líneas maestras que encontrarás en el archivo cimastudio.cpp).

Para actualizar la variable **sliderValue** instantáneamente cada vez que tocas el control, utilizamos esta conexión:

```
// [C2] Conexión entre el slider y la variable interna
connect(slider, &QSlider::valueChanged, this, [this](int value) { sliderValue = value;
});
```

Definir un etiqueta (Ver si ya está [C2] y utilizarla aquí.

Detalles clave:

- **slider**: Es el puntero al objeto QSlider que creaste manualmente en tu código.
- **valueChanged**: Esta señal emite el nuevo valor del slider en el mismo instante en que el usuario lo mueve (a diferencia de sliderReleased, que espera a que sueltes el ratón).
- **[this](int value)**: La función captura el contexto de tu clase para poder acceder y modificar la variable miembro sliderValue.

Referencia al Laboratorio: Como en el punto 2.2.1 hablas de una lambda ([this](int value)...), es el momento ideal para poner un pequeño recordatorio de que eso se explica en el **Anexo**.

Detalles clave para entender esta línea:

- **slider**: Es el puntero al objeto QSlider que creamos manualmente.

- **&QSlider::valueChanged**: Es la señal. Se activa en el mismo instante en que el usuario desliza el control (logrando esa sensación de tiempo real).
- **[this](int value) { ... }**: Esto es una función Lambda. Permite que el valor del slider viaje directamente a nuestra variable sliderValue sin necesidad de crear funciones extra.

3.- La Ingeniería detrás de la Escena

Hasta ahora hemos explorado CimaStudio desde el exterior, como el conductor que se sienta frente al volante de un coche deportivo. Sabemos qué hacen los pedales y el volante, pero para ser verdaderos artesanos de la visión artificial, debemos abrir el capó.

En este bloque vamos a diseccionar la arquitectura del sistema. No se trata solo de acumular líneas de código, sino de entender la estrategia que permite que una imagen capturada por la cámara viaje, se transforme matemáticamente mediante algoritmos de OpenCV y se proyecte en una interfaz de Qt sin perder un solo milisegundo.

Para navegar por este "océano de datos" sin perder el rumbo, utilizaremos tres herramientas fundamentales:

- Un mapa estructural: Para saber dónde reside cada responsabilidad.
- Un sistema de etiquetas: Que actuará como nuestro GPS entre la explicación y el código puro.
- La lógica de compilación: El pegamento que une librerías de distintos mundos en un solo ejecutable.
- Prepárate: aquí es donde el "arte de enseñar a ver a las máquinas" se convierte en ingeniería real.

3.1.- El sistema de etiquetas: Cómo leer el tutorial sin perderse.

Para que este manual sea fluido y no te pierdas en un "baile de archivos", utilizaremos el Sistema de Siglas por Contexto (SSC). Este método permite saltar entre la teoría y el código de forma instantánea mediante el uso de hitos.

3.1.1.- El Código de Siglas

Hemos asignado una letra única a cada pieza clave de CimaStudio según su función. Guarda esta leyenda como recordatorio:

📂 [H] → Header (cimastudio.h): El plano del edificio (la estructura).

📂 [C] → Cpp (cimastudio.cpp): Las máquinas trabajando (la lógica).

📂 [M] → Main (main.cpp): La llave de contacto (el arranque).

📂 [K] → Kit/CMake (CMakeLists.txt): La lista de materiales (la receta).

3.1.2.- Cómo se aplica: Hitos vs. Líneas

En lugar de usar números de línea —que cambian si añadimos un espacio—, usaremos hitos fijos en el código (ej. [C1], [H2]).

Ejemplo de uso en el texto:

> *Para despertar a la cámara, ejecutamos la apertura del flujo físico [C1], habiendo definido previamente el objeto en el archivo de cabecera [H1]. Todo esto es posible gracias a nuestra receta de compilación [K1]."*

3.1.3.- Ventajas de este sistema

- Precisión: Irás directo al punto exacto del archivo sin importar cuántas páginas tenga el código.
- Independencia: Si mañana insertas 10 líneas de código en medio, la etiqueta sigue siendo válida; el manual no queda obsoleto.
- Limpieza: Solo usas 3 o 4 caracteres que no ensucian la lectura del código fuente.

3.1.4.- Ejercicio rápido de localización

Antes de seguir, intenta localizar este par de etiquetas en tus archivos:

- En cimastudio.h: cv::VideoCapture cap; // **[H1]**
- En cimastudio.cpp: cap.open(0); // **[C1]**

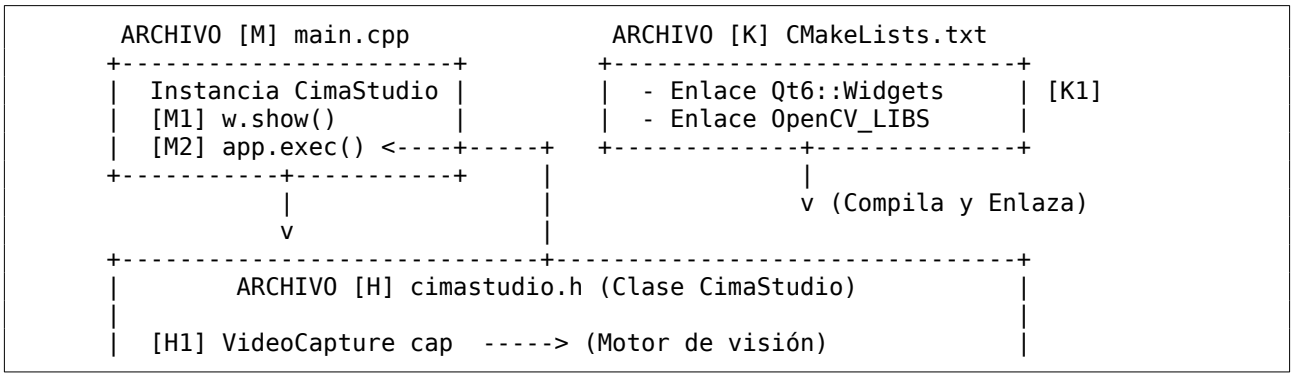
3.2.- Mapa estructural de CimaStudio v1.0: Radiografía de las carpetas y ficheros.

Para capturar la esencia de CimaStudio v1.0, lo mejor es un enfoque mixto: los archivos actúan como "contenedores" y las clases como los "motores" internos.

Aquí tienes el mapa arquitectónico del flujo de información. He diseñado el diagrama para que puedas ver cómo el Timer actúa como el director de orquesta que conecta el mundo de OpenCV con el mundo de Qt.

🗺️ Mapa Estructural de CimaStudio v1.0

text



```
|  [H3] QTimer *timer     -----> (Corazón/Ritmo)              |
|  [H2] QLabel *viewLabel -----> (Lienzo de salida)            |
+-----------------------------+--------------------------------+
            |                 |
            | (Define)        | (Implementa)
            v                 v
+-------------------------------------------------------------+
|       ARCHIVO [C] cimastudio.cpp (Lógica de flujo)          |
|                                                             |
|  1. CONSTRUCTOR [C1]:                                       |
|     cap.open(0)  -----------> Despierta la cámara           |
|     timer->start(30) [C2] --> Inicia el pulso (cada 30ms)   |
|                                                             |
|  2. BUCLE updateFrame():  <--- (Se ejecuta en cada pulso)   |
|     [C4] cap >> frame   ----> Captura luz (Matriz RAM)      |
|     [C7] cvtColor()     ----> Alquimia (Paso a grises)      |
|     [C8/C9] Algoritmos  ----> Canny / Threshold             |
|     [C10] QImage puente ----> Traducción Mat -> QImage      |
|     [C11] setPixmap()   ----> Proyección en Pantalla        |
+-------------------------------------------------------------+
```

3.2.1.- 🔍 Guía de Lectura: Entendiendo el pulso de CimaStudio

Para que no te pierdas en el flujo, observa el mapa como si fuera un organismo vivo:

- Los Cimientos (Eje Superior): El main.cpp [M] es la llave que arranca el motor, y el CMakeLists.txt [K] es el manual de ensamblaje que asegura que Qt y OpenCV hablen el mismo idioma.
- El Plano (Eje Central): El archivo de cabecera [H] es nuestro inventario de órganos. Aquí decidimos que tendremos un "ojo" (cap), un "corazón" (timer) y una "pantalla" (viewLabel).
- El Ciclo de Vida (Eje Inferior): El archivo .cpp [C] es donde ocurre la magia.
    - El Constructor [C1]: Es el recepcionista que enciende las luces y comprueba que la cámara responda.
    - El Latido [C2]: Aquí aplicamos lo aprendido sobre Signals & Slots. El timer emite una señal cada 30 milisegundos que "golpea" el Slot updateFrame.
- El Bucle Infinito: Fíjate en la flecha que sale del Timer y activa el bloque de procesamiento. Mientras el cronómetro dicte el ritmo, el programa repetirá el ciclo Captura ➔ Alquimia ➔ Traducción ➔ Proyección unas 33 veces por segundo.

3.3.- El corazón de la compilación: CMakeLists.txt [K]

Si los archivos .cpp y .h son los ladrillos de nuestra aplicación, el archivo CMakeLists.txt (etiquetado como [K]) es el cemento y el plano de ingeniería. Su función es orquestar la integración entre dos mundos distintos: Qt y OpenCV.

Sin este archivo, el compilador no sabría qué es un QPushButton o cómo gestionar una matriz cv::Mat.

La Receta de Compilación

A continuación, vemos el contenido de nuestra "receta" y la explicación de sus hitos fundamentales:

```
# [K] ARCHIVO DE CONFIGURACIÓN DEL PROYECTO
cmake_minimum_required(VERSION 3.10)
project(CimaStudio)

# [K1] Estándar de C++ moderno y automatización de Qt
set(CMAKE_CXX_STANDARD 17)
set(CMAKE_AUTOMOC ON)

# [K2] Búsqueda de Dependencias Externas
find_package(Qt6 REQUIRED COMPONENTS Widgets)
find_package(OpenCV REQUIRED)

# [K3] Definición del Ejecutable Final
add_executable(CimaStudio main.cpp cimastudio.h cimastudio.cpp)

# [K4] El Enlace Maestro
target_link_libraries(CimaStudio Qt6::Widgets ${OpenCV_LIBS})
```

Análisis de los Hitos [K]:

- **La Base [K1**]: Establecemos el uso de C++17, necesario para las funciones modernas (como las lambdas), y activamos AUTOMOC, que es el proceso interno de Qt que permite que nuestras **Signals & Slots** funcionen "entre bambalinas".
- La Búsqueda [K2]: El comando find_package sale al sistema operativo a buscar dónde están instaladas las librerías. Si no las encuentra, la compilación se detiene aquí.
- El Montaje [K3]: Aquí le decimos a CMake: "*Toma estos tres archivos fuente y prepárate para crear un programa llamado CimaStudio*".
- El Enlace Maestro [K4]: Esta es la línea crítica. target_link_libraries toma nuestro código ya compilado y lo "pega" físicamente a las librerías de Qt y OpenCV. Es el momento en que todas las piezas encajan para crear el ejecutable final.

3.4.- Bajo el capó: Inmersión en el código fuente de CimaStudio

main.ccp

```
/**
 * @file main.cpp
 * @brief Punto de entrada de la aplicación CimaStudio.
 * @author alcón68
 * @date 2024
 */

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

/**
 * @brief Función principal que arranca el bucle de eventos de Qt.
 * @param argc Contador de argumentos de consola.
 * @param argv Vector de argumentos de consola.
 * @return Código de salida del sistema.
 */
```

```cpp
int main(int argc, char *argv[]) {
    QApplication a(argc, argv);

    CimaStudio w;            // [M1] El Arranque: Instancia de la clase principal
    w.setWindowTitle("CimaStudio: Editor de Visión v1.0");
    w.resize(800, 600);
    w.show();

    return a.exec();         // [M2] El Bucle: Inicia la gestión de señales y eventos
}
```

cimastudio.h

```cpp
#ifndef CIMASTUDIO_H
#define CIMASTUDIO_H

#include <QMainWindow>
#include <QTimer>
#include <QLabel>
#include <QRadioButton>
#include <QSlider>
#include <QPushButton>
#include <QVBoxLayout>
#include <QHBoxLayout>
#include <opencv2/opencv.hpp>

/**
 * @class CimaStudio
 * @brief Ventana principal de la aplicación de visión artificial.
 * @details Gestiona la interfaz de usuario y el flujo de captura de vídeo
 *          mediante un temporizador sincronizado.
 */
class CimaStudio : public QMainWindow {
    Q_OBJECT

public:
    /**
     * @brief Constructor de la clase CimaStudio.
     * @param parent Puntero al widget padre.
     */
    CimaStudio(QWidget *parent = nullptr);

    /**
     * @brief Destructor de la clase CimaStudio.
     * @details Libera los recursos de hardware de la cámara.
     */
    ~CimaStudio();

private slots:
    /**
     * @brief Ciclo de actualización de fotogramas.
     * @details Función invocada por el QTimer para procesar la imagen.
     */
    void updateFrame();

private:
    cv::VideoCapture cap;    // [H1] El Ojo: Motor de captura de OpenCV
    QTimer *timer;           // [H3] El Corazón: Temporizador de refresco
    int sliderValue;
    bool showOriginal;
```

```
    QWidget *centralWidget;
    QLabel *viewLabel;       // [H2] El Lienzo: Proyector de imagen en Qt
    QRadioButton *radioCanny;
    QRadioButton *radioThreshold;
    QSlider *paramSlider;
    QPushButton *btnToggleMode;

    /**
     * @brief Configura la disposición de los elementos visuales.
     */
    void setupUI();
};

#endif
```

cimastudio.cpp

```
/**
 * @file cimastudio.cpp
 * @brief Implementación de la lógica de visión y gestión de interfaz.
 */

#include "cimastudio.h"

/**
 * @brief Constructor: Configura la UI e inicia la comunicación con el hardware.
 * @param parent Puntero al widget padre.
 */
CimaStudio::CimaStudio(QWidget *parent) : QMainWindow(parent), sliderValue(100),
showOriginal(false) {
    setupUI();

    cap.open(0);             // [C1] Despertando el Hardware: Abre la cámara por
defecto

    timer = new QTimer(this);
    connect(timer, &QTimer::timeout, this, &CimaStudio::updateFrame);
    timer->start(30);        // [C2] El Pulso: Dispara el procesamiento cada 30ms
}

/**
 * @brief Organiza los widgets y establece las conexiones de los controles.
 */
void CimaStudio::setupUI() {
    centralWidget = new QWidget(this);
    setCentralWidget(centralWidget);
    QVBoxLayout *mainLayout = new QVBoxLayout(centralWidget);

    viewLabel = new QLabel("Iniciando flujo...", this);
    viewLabel->setAlignment(Qt::AlignCenter);
    mainLayout->addWidget(viewLabel, 4);

    QHBoxLayout *controls = new QHBoxLayout();
    radioCanny = new QRadioButton("Filtro Canny", this);
    radioThreshold = new QRadioButton("Umbralización", this);
    radioCanny->setChecked(true);
    controls->addWidget(radioCanny);
    controls->addWidget(radioThreshold);
    mainLayout->addLayout(controls);

    paramSlider = new QSlider(Qt::Horizontal, this);
```

```
    paramSlider->setRange(0, 255);
    paramSlider->setValue(sliderValue);
    connect(paramSlider, &QSlider::valueChanged, [this](int v){ sliderValue = v; });
    mainLayout->addWidget(new QLabel("Intensidad del Filtro:"));
    mainLayout->addWidget(paramSlider);

    btnToggleMode = new QPushButton("Ver Imagen REAL", this);
    connect(btnToggleMode, &QPushButton::clicked, [this](){
        showOriginal = !showOriginal;
        btnToggleMode->setText(showOriginal ? "Ver Imagen PROCESADA" : "Ver Imagen
REAL");
    });
    mainLayout->addWidget(btnToggleMode);
    mainLayout->addStretch();
}

/**
 * @brief Función crítica de procesamiento de imagen y actualización visual.
 */
void CimaStudio::updateFrame() {
    cv::Mat frame, processed; // [C3] Los Contenedores: Memoria en el Heap
    cap >> frame;             // [C4] La Captura: Transferencia sensor -> matriz

    if (frame.empty()) return; // [C5] El Seguro: Control de flujo de datos

    if (showOriginal) {
        processed = frame.clone(); // [C6] El Espejo: Duplicado completo de datos
    } else {
        // [C7] La Alquimia: Reducción de dimensionalidad (Color a Gris)
        cv::cvtColor(frame, processed, cv::COLOR_BGR2GRAY);

        if (radioCanny->isChecked()) {
            // [C8] El Dibujo: Detección de bordes mediante algoritmo de Canny
            cv::Canny(processed, processed, sliderValue/2, sliderValue);
        } else {
            // [C9] La Decisión: Binarización por umbral (Threshold)
            cv::threshold(processed, processed, sliderValue, 255, cv::THRESH_BINARY);
        }
    }

    // [C10] El Puente: Conversión de formato Mat a QImage para Qt
    QImage::Format format = (processed.channels() == 1) ? QImage::Format_Grayscale8 :
QImage::Format_BGR888;
    QImage qimg(processed.data, processed.cols, processed.rows, processed.step,
format);

    // [C11] La Proyección: Actualización del Pixmap en el hilo principal
    viewLabel->setPixmap(QPixmap::fromImage(qimg).scaled(viewLabel->size(),
Qt::KeepAspectRatio, Qt::SmoothTransformation));
}

/**
 * @brief Destructor: Asegura la liberación de los descriptores de la cámara.
 */
CimaStudio::~CimaStudio() {
    cap.release();
}
```

CmakeLists.txt

```
# [K] ARCHIVO DE CONFIGURACIÓN DEL PROYECTO (RECETA)
```

```
cmake_minimum_required(VERSION 3.10)
project(CimaStudio)

# Estándar de C++ moderno
set(CMAKE_CXX_STANDARD 17)
set(CMAKE_AUTOMOC ON) # Genera automáticamente el código para Signals/Slots

# Búsqueda de Dependencias Externas (Fuentes Primarias)
find_package(Qt6 REQUIRED COMPONENTS Widgets)
find_package(OpenCV REQUIRED)

# Definición del Ejecutable Final
add_executable(CimaStudio main.cpp cimastudio.h cimastudio.cpp)

# [K1] El Enlace: Une el código objeto con las librerías dinámicas
target_link_libraries(CimaStudio Qt6::Widgets ${OpenCV_LIBS})
```

Con estos listados, tu alumno puede tener el código en una pantalla y tu texto explicativo en otra, saltando de [C4] a [C10] sin perderse ni un solo bit de información.

¿Te gustaría que repasemos alguna etiqueta específica que creas que necesita un refuerzo visual en el manual?Siguiente paso

3.5.- Anatomía del Código: Desglosando las funciones maestras de CimaStudio

Una vez presentados los listados, vamos a detenernos en los bloques que realmente sostienen la aplicación. No analizaremos cada punto y coma, sino la responsabilidad de cada función principal.

3.5.1.- El punto de ignición: main.cpp [M]

Este archivo es breve pero vital. Su única función es crear la instancia de QApplication y lanzar nuestra ventana principal.

- La clave: El método app.exec() es el que mantiene el programa vivo. Si esta línea no se ejecuta, la ventana se abriría y cerraría en una fracción de segundo. Es el "bucle de eventos" que espera a que el usuario interactúe.

3.5.2.- El Constructor: CimaStudio::CimaStudio() [C1]

Es el encargado de la puesta a punto inicial. Aquí ocurren tres hitos críticos:

- Apertura de la cámara: Usamos cap.open(0) para reclamar el control del hardware.
- Configuración del Timer: El objeto timer se configura para que emita un "latido" cada 30 milisegundos.
- El "cableado" (Connect): Aquí es donde vinculamos la señal del Timer con nuestra función de procesamiento. Es el momento en que el programa pasa de ser una foto estática a un sistema dinámico.

3.5.3.- El Motor de Procesamiento: updateFrame() [C4-C11]

Esta es la función más importante de todo el tutorial. Se ejecuta 33 veces por segundo y sigue un flujo lineal de cuatro etapas:

- Captura: Extrae la luz del sensor y la deposita en una matriz cv::Mat frame.
- Alquimia (Procesamiento): Dependiendo del algoritmo seleccionado (Canny, Threshold, etc.), transforma los valores numéricos de esa matriz. Es donde OpenCV demuestra su potencia matemática.
- Traducción: Como Qt no sabe "dibujar" una matriz de OpenCV directamente, creamos un puente usando QImage. Es una conversión de formatos de memoria.
- Proyección: Finalmente, enviamos esa imagen al viewLabel para que el usuario pueda ver el resultado en pantalla.

3.5.4.- La Gestión de Memoria: El Destructor

Aunque a veces se olvida, el destructor ~CimaStudio() asegura que, al cerrar la aplicación, liberemos la cámara (cap.release()) y detengamos el timer. Es la etiqueta de "buen ciudadano" en programación C++: dejar todo como lo encontramos.

3.6.- El corazón de la compilación: CMakeLists.txt [K]: El director de orquesta que une Qt y OpenCV.

El archivo CMakeLists.txt, referenciado con la etiqueta [K], actúa como el director de producción del proyecto, orquestando la integración entre Qt y OpenCV [1]. Su función principal es localizar las librerías, preparar los archivos del proyecto y gestionar el ensamblaje para convertir el código en un ejecutable funcional [1].

CmakeLists.txt

```
# [K] ARCHIVO DE CONFIGURACIÓN DEL PROYECTO (RECETA)
cmake_minimum_required(VERSION 3.10)
project(CimaStudio)

# Estándar de C++ moderno
set(CMAKE_CXX_STANDARD 17)
set(CMAKE_AUTOMOC ON) # Genera automáticamente el código para Signals/Slots

# Búsqueda de Dependencias Externas (Fuentes Primarias)
find_package(Qt6 REQUIRED COMPONENTS Widgets)
find_package(OpenCV REQUIRED)

# Definición del Ejecutable Final
add_executable(CimaStudio main.cpp cimastudio.h cimastudio.cpp)

# [K1] El Enlace: Une el código objeto con las librerías dinámicas
target_link_libraries(CimaStudio Qt6::Widgets ${OpenCV_LIBS})
```

El archivo CMakeLists.txt actúa como director de orquesta, uniendo las librerías Qt y OpenCV para compilar el proyecto CimaStudio. Su estructura incluye la configuración del estándar C++17 [K1], la localización de dependencias mediante find_package [K2], la definición del ejecutable con

add_executable [K3], y el enlazado final de librerías con target_link_libraries [K4]. Esta configuración es crucial para asegurar que el código fuente localice y utilice correctamente los componentes externos.

4.- Flujo de la Luz: La Arquitectura Base

En este capítulo vamos a estudiar cómo CimaStudio establece el camino que sigue la imagen desde que entra por la lente hasta que se dibuja en tu monitor. Utilizaremos nuestro sistema de etiquetas para rastrear este viaje a través de los archivos.

4.1.- El Despertar del Sistema

Antes de procesar un solo píxel, el programa debe "construirse" y conectarse con el hardware.

- **La Definición del Escenario [H]**: Todo comienza en el archivo de cabecera. En [H1] definimos cv::VideoCapture cap;, el objeto de OpenCV encargado de hablar con el controlador de tu cámara. Sin esta línea, el programa estaría ciego. En [H2] preparamos el QLabel *viewLabel;, que funcionará como nuestra pantalla de cine en Qt.
- El Arranque del Motor [M]: El archivo main.cpp es la llave. En [M1] creamos la instancia de la clase, disparando su constructor, y en [M2] ejecutamos app.exec(), el bucle que mantiene el programa vivo y a la escucha del usuario.
- La Conexión con la Realidad [C]: En el constructor [C1], despertamos físicamente a la cámara con cap.open(0). Es aquí donde el LED de tu webcam suele encenderse. Inmediatamente, en [C2], activamos el QTimer: nuestro "corazón" que latirá cada 30ms para pedir un nuevo cuadro.

4.2.- El Alquimista Digital: updateFrame()

Una vez que el motor está en marcha, entramos en la función que se ejecuta 33 veces por segundo. Aquí es donde la luz se convierte en matemáticas y las matemáticas en visión.

```
void CimaStudio::updateFrame() {
    cv::Mat frame, processed;  // [C3] Los contenedores (Matrices)
    cap >> frame;              // [C4] Captura: El "robo" de luz
    if (frame.empty()) return; // [C5] El seguro de vida
    // ... algoritmos ...
}
```

4.3.- Anatomía de la Transformación:

El Contenedor de Realidad [C3]: Es vital entender que cv::Mat no es una imagen "visual", es una matriz numérica en la memoria RAM. Al declarar frame, estamos reservando el espacio exacto para los millones de números que componen un cuadro de video.

El Acto de Captura [C4]: El operador >> es fascinante. Literalmente "vuelca" el flujo de fotones digitalizados desde la cámara hacia nuestra matriz. Es el preciso instante en que el mundo físico se convierte en datos procesables.

El Seguro de Vida [C5]: En visión artificial, siempre debemos comprobar si la matriz está vacía. Si un cable se suelta o la cámara se bloquea, esta línea evita que el programa intente procesar "la nada" y se cierre por error.

4.4.- Resumen del Flujo de Control:

CMake [K] prepara las herramientas.

Main [M] arranca la aplicación.

Constructor [C] enciende la cámara y el pulso (Timer).

updateFrame [C] ejecuta el ciclo infinito de "captura-procesa-muestra".

5.- Ampliación de Conceptos

En esta sección vamos a profundizar en la ingeniería que sostiene a OpenCV. Entenderemos cómo se organiza la memoria y cuál es la lógica matemática que permite a una máquina "ver" bordes o formas.

5.1.- La Estructura cv::Mat: El Corazón de OpenCV

Una imagen digital en OpenCV no es un archivo .jpg o .png; es un objeto de la clase cv::Mat. En C++, cuando declaras una variable cv::Mat A, el objeto se comporta como una estructura bicefálica:

- La Cabecera (La Sombra): Se aloja en la pila (Stack). Es ligera y contiene los metadatos (dimensiones, tipo de dato) y el puntero A.data que señala la dirección donde empieza el bloque masivo de píxeles.
- La Matriz de Datos (El Cuerpo): Reside en el montón (Heap). Es el bloque "pesado" que contiene los píxeles reales.

Esta arquitectura permite que OpenCV sea increíblemente rápido: al pasar una imagen a una función o hacer cv::Mat B = A, solo mueves la "pequeña" cabecera en el Stack. Ambas cabeceras terminarán apuntando al mismo cuerpo en el Heap, compartiendo la información y ahorrando memoria.

5.1.1.- La Cabecera (El "DNI" de la imagen) - Vive en el Stack

Contiene la información de gestión. Es pequeña, ligera y siempre viaja con la imagen. Sus propiedades más útiles son:

- .cols y .rows: Nos dicen el ancho y el alto en píxeles.
- .size(): Devuelve ambos valores a la vez.
- .type(): Indica qué "idioma" hablan los píxeles (¿Color?, ¿Grises?, ¿Decimales?).
- .step: Medida técnica que indica cuántos bytes ocupa una fila completa (vital para que el ordenador no se pierda al leer la memoria).

5.1.2.- La Matriz de Datos (El "Músculo") - Vive en el Heap

Aquí residen los millones de números de los píxeles. El espacio depende de las Constantes de Tipo:

- CV_8UC1: 8 bits (un byte), sin signo (0-255), 1 canal. Estándar para Escala de Grises.
- CV_8UC3: Igual que el anterior, pero con 3 canales (Azul, Verde y Rojo). Estándar para Color (BGR).
- CV_32F: Para precisión decimal (cálculos científicos o HDR).

5.1.3.- La "Copia Inteligente" (Gestión de Memoria)

Hacer Mat B = A; es como tener dos mandos a distancia para una sola TV. Para una copia real e independiente que no afecte a la original, debes usar .clone() (ej. cv::Mat B = A.clone();).

5.2.- El Arte de la Convolución: ¿Cómo "piensa" el algoritmo?

La mayoría de los filtros de CimaStudio utilizan una operación llamada convolución. Imagina una pequeña rejilla cuadrada (número impar de píxeles como 3x3 o 5x5) llamada Núcleo o Kernel.

El Kernel se desliza por la imagen. Su punto central se sitúa sobre un píxel, "mira" a los vecinos de la imagen que tiene debajo y realiza un cálculo rápido de promedios o diferencias. El resultado es el nuevo valor que recibirá ese píxel en la imagen de salida.

- Suavizado (Gaussiano): El núcleo promedia los píxeles vecinos. Es como pasar un dedo húmedo sobre un dibujo al carboncillo; los detalles se difuminan y el "ruido" desaparece.
- Detección de Bordes (Derivadas): El núcleo busca contrastes. Realiza una especie de "derivada" sobre la distribución de píxeles: si hay un salto brusco de luz a sombra, el valor es alto. Así, algoritmos como Canny encuentran las líneas de los objetos.
- Realce (Sharpen): Exagera las diferencias entre vecinos para una mayor nitidez.

  Nota para el alumno: Cuando en nuestro código ejecutamos cv::cvtColor(frame, processed, COLOR_BGR2GRAY), OpenCV crea una nueva cabecera y una nueva matriz de datos donde, mediante un cálculo píxel a píxel, transforma una estructura CV_8UC3 (color, 3 canales) en una CV_8UC1 (gris, un canal).

5.3.- El contador de referencias de cv::Mat

OpenCV utiliza un mecanismo de conteo de referencias para gestionar la memoria automáticamente. Gracias a esto, no necesitas usar comandos como delete o free; el sistema sabe cuándo borrar los datos por ti.

5.3.1.- El Sistema de "Votos" de Memoria

Imagina que el bloque de píxeles en el Heap es una casa y cada Cabecera en el Stack es un mando a distancia que la abre.

- Creación (Voto = 1): Al capturar un frame, OpenCV reserva memoria y crea la primera cabecera. El contador se pone en 1: hay un "dueño" usando esa memoria.

- Asignación (Voto = 2): Al hacer cv::Mat copia = frame;, no se duplican los píxeles. Se crea otra cabecera que apunta al mismo bloque. El contador sube a 2.
- Destrucción (Voto - 1): Cuando una función termina, sus variables locales desaparecen. Si una cabecera sale de ámbito, el contador baja. La memoria sigue viva si queda al menos un "voto" activo.
- Auto-Limpieza (Voto = 0): Cuando ya no queda ninguna cabecera apuntando a esos datos, el contador llega a cero y OpenCV libera automáticamente el bloque del Heap.

5.3.2.- Eficiencia en CimaStudio

En una App que procesa 30 fps, esto es vital para evitar fugas de memoria (Memory Leaks). Permite mover imágenes entre funciones casi instantáneamente (solo se copian cabeceras) y garantiza que la RAM se libere justo cuando deja de ser útil.

5.3.3.- La excepción: .clone()

Al usar B = A.clone();, obligas a OpenCV a duplicar los píxeles en un nuevo bloque del Heap. Ahora tienes dos "casas" independientes, cada una con su propio conteo de votos.

5.4.- ¿Por que lo hemos llamado Vots? (Counting OpenCV)

Llamarlo "votos" es una licencia pedagógica para visualizar que la supervivencia de los datos depende de un consenso:

- La mayoría simple es 1: Basta con que una sola variable mantenga su "mano levantada" (cabecera activa) para que los datos sobrevivan.
- El desahucio: En el instante en que el último "votante" se retira (la variable sale de ámbito), el sistema ejecuta el desahucio y limpia la RAM.

Conclusión: La memoria en OpenCV funciona por democracia. Los píxeles solo mueren cuando el último interesado desaparece del programa.

6.- 🧪 Anexo: El Laboratorio de CimaStudio

Construyendo la base antes de alcanzar la cima. Los ejemplos que veremos a continuación constituyen la base de CimaStudio y los encontrarás en la carpeta **Ejemplos_v1.0** de tu matarial de descarga. Cada ejemplo está alojado en un directorio que contiene todo lo necesario para compilarlo y obtener un ejecutable.

Si has llegado hasta aquí, es porque tienes la curiosidad necesaria para entender no solo qué hace un programa, sino cómo lo hace.

Aunque el tutorial principal de CimaStudio desglosa una arquitectura completa y funcional, a menudo es útil dar un paso atrás y aislar los componentes. Este Laboratorio es una colección de "píldoras tecnológicas" diseñadas para que puedas experimentar con cada concepto de forma independiente, sin el ruido de una aplicación compleja.

Si en algún momento del tutorial sientes que los punteros, las lambdas o el acceso a los píxeles te resultan confusos, no te detengas: este Anexo: Laboratorio, contiene píldoras de código diseñadas para aclarar estos conceptos.

Cada ejemplo sigue una estructura pensada para tu aprendizaje:

- **El Concepto**: Una breve explicación de la herramienta que vamos a dominar.
- **El Código Maestro:** Un archivo **main.cpp** limpio y comentado, donde cada línea tiene una razón de ser.
- **La Conexión**: Una nota final que te explica **exactamente en qué parte de CimaStudio** se aplica ese conocimiento.

No veas estos ejemplos como simples ejercicios de copia y pega; míralos como los cimientos de tu carrera en C++. Si dominas estos seis peldaños, el código de cualquier sistema de visión artificial dejará de ser un misterio para convertirse en un libro abierto.

Te recomendamos recorrer este Laboratorio si:

- Es tu primera vez trabajando con la gestión de memoria en C++.
- Quieres entender la "magia" que hay detrás de las funciones Lambda.
- Deseas ver cómo se tocan los píxeles de una imagen por primera vez sin miedo a romper nada.

Un consejo de autor:

> *"La ingeniería no se aprende leyendo, se aprende rompiendo. Te invito a que modifiques estos ejemplos, cambies los valores, alteres los bucles y veas qué sucede. Solo cuando el programa falla y logras entender por qué, es cuando realmente te conviertes en programador."*

6.1.- 🏷️ Ejemplos v1.0

En este Laboratorio recorreremos mediante un "Currículum de 6 Pasos" los principales conceptos de CimaStudio. Son los ejemplos que, según lo que hemos trabajado, forman la columna vertebral técnica de CimaStudio.

Ésta es nuestra Propuesta de Selección: "La Escalera del Aprendizaje"

1. **HolaMundo_Qt**: El motor mínimo (QApplication + QpushButton).
2. **Memoria_Dinamica**: (Stack vs Heap)
3. **Lambdas_y_Slots:** (Comunicación Moderna)
4. **HolaMundo_OpenCV**: (La Cámara y la Matriz)
5. **El_Puente_Qimage**: (Uniendo Mundos)
6. **Escaneo_Matrices**: (Acceso a los Píxeles)

Los listados de los programas están suficientemente comentados para su total comprensión y se añade además un párrafo de introducción a cada programa para aclarar la función que realiza y otro párrafo al final a modo de conclusión.

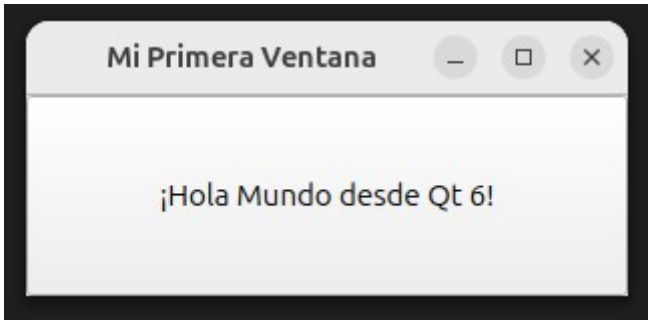
6.2.- 🌱 HolaMundo_Qt (El motor mínimo)

- QApplication + QpushButton

Introducción:

Este ejemplo es la piedra angular de cualquier interfaz gráfica con Qt. El objetivo es entender que una aplicación visual no es una ejecución lineal de arriba hacia abajo, sino un "Bucle de Eventos". Aquí aprenderemos a inicializar el motor de la aplicación (QApplication) y a instanciar nuestro primer objeto visual (QPushButton). Es un programa sencillo, pero contiene el esqueleto jerárquico que permite que ventanas, botones y sliders existan y respondan al usuario. Sin este punto de partida, no habría un "lienzo" donde proyectar nuestras imágenes de OpenCV.

La salida del programa:



El botón forma y ocupa toda la ventana

main.cpp

```
#include <QApplication> // El "cerebro" que gestiona la App
#include <QPushButton>  // Nuestro primer control visual

/**
 * PROGRAMA: 01_HolaMundo_Qt
 * OBJETIVO: Crear una ventana mínima con un botón funcional.
 * FUNCIONAMIENTO: Se instancia la aplicación, se crea un widget y se entra
 * en el bucle de espera (event loop).
 */

int main(int argc, char *argv[]) {
    // 1. Inicializamos el gestor de la aplicación
    // Se encarga de captar clics, movimientos de ratón y el cierre de ventanas.
    QApplication app(argc, argv);

    // 2. Creamos un botón (un "Widget" o control visual)
    // En Qt, casi todo lo que ves hereda de QWidget.
    QPushButton boton("¡Hola Mundo desde Qt 6!");

    // 3. Configuramos su aspecto inicial
    boton.resize(300, 100);
    boton.setWindowTitle("Mi Primera Ventana");

    // 4. Mostramos el botón en pantalla
```

```
    // Por defecto, los widgets en Qt se crean "invisibles" en memoria.
    boton.show();

    // 5. Entramos en el Bucle de Eventos (Event Loop)
    // El programa se queda aquí "escuchando" hasta que cerremos la ventana.
    // return asegura que el S.O. reciba el código de salida correcto.
    return app.exec();
}
```

Comentarios

El método show(). En Qt, los objetos visuales nacen "ocultos" en memoria. Al llamar a .show(), le pedimos al sistema operativo que los pinte. En este caso, el botón actúa como ventana principal. Más adelante, usaremos QMainWindow como contenedor, pero es importante saber que cualquier Widget puede ser, por sí mismo, una ventana.

El motor minimo: QApplication, además de captar clics, movimientos de ratón y el cierre de ventanas, es quien gestiona el ciclo de vida.

Conexión con CimaStudio: El comando app.exec() inicia el Event Loop. Este es exactamente el mismo "motor" que el alumno encontrará en la etiqueta **[M2]** del tutorial principal; sin él, la interfaz se abriría y cerraría en milisegundos.

6.2.1.- Aclaración sobre el fichero CMakeList.txt

El fichero CMakeLists.txt que encotrarás en la carpeta del ejemplo, se requiere en el proceso de compilación tal como se detalla en la parte central del tutorial de CimaStudio. Consulta esa sección para acceder a una explicación del proceso de compilación.

Hemos elegido este CMakeLists.txt por las siguientes razones:

- **Versión 3.16**: Es una elección muy inteligente. Es la mínima requerida por Qt 6, lo que garantiza compatibilidad sin exigir una versión excesivamente moderna del sistema.
- **C++17**: Imprescindible para el uso de Lambdas (que tienes en tu programa de pasos) y para que Qt 6 funcione con toda su potencia.
- **Enlazado directo**: Al usar target_link_libraries directamente con Qt6::Widgets, evitas usar macros automáticas que a veces "esconden" lo que está pasando por debajo.

6.3.- 📦 Memoria_Dinamica (Stack vs Heap)

Este es un paso crítico. En CimaStudio usamos punteros para casi todo (el Timer, las etiquetas, los botones), y si no entiendes por qué los creamos con new, el código te parecerá "ruido".

Introducción:

En C++, la memoria se gestiona entre la Pila (Stack), que es automática y eficiente para datos efímeros, y el Montón (Heap), que es persistente y crucial para objetos que deben sobrevivir fuera del alcance de la función que los crea. En el desarrollo con Qt y CimaStudio, el uso de punteros y el operador new permite que componentes como ventanas (QMainWindow) o etiquetas (QLabel)

persistan a la función que las crea, delegando la limpieza de memoria al sistema de parentesco de Qt.

main.cpp

```
#include <iostream>
#include <string>

/**
 * PROGRAMA: 02_Memoria_Dinamica
 * OBJETIVO: Diferenciar visualmente la persistencia en el Stack vs el Heap.
 * CLAVE: Uso de punteros (*) y el operador 'new'.
 */

int main() {
    // 1. VARIABLE EN EL STACK (La Pila)
    // Es automática. Se crea y se destruye sola al llegar al final de las llaves {}.
    int edadStack = 30;

    // 2. VARIABLE EN EL HEAP (El Montón)
    // Usamos 'new'. No estamos creando un entero, sino "pidiendo sitio" para uno.
    // El puntero 'p_edadHeap' solo guarda la DIRECCIÓN de ese sitio.
    int* p_edadHeap = new int(45);

    std::cout << "Valor en el Stack: " << edadStack << std::endl;
    std::cout << "Direccion de memoria en el Stack: " << &edadStack << std::endl;
    std::cout << "Valor en el Heap (via puntero): " << *p_edadHeap << std::endl;
    std::cout << "Direccion de memoria en el Heap: " << p_edadHeap << std::endl;

    // 3. LA SUTILEZA TÉCNICA
    // Si estuviéramos en una función de CimaStudio, al terminar, 'edadStack'
    // moriría. Pero el entero de 'p_edadHeap' seguiría vivo en la RAM.

    // 4. LIMPIEZA MANUAL
    // En C++ puro, lo que creas con 'new' debes borrarlo con 'delete'.
    // (Nota: En Qt, muchas veces el sistema lo hace por nosotros, pero es
    // vital entender que alguien debe "limpiar la mesa").
    delete p_edadHeap;

    return 0;
}
```

Salida del programa:

Valor en el Stack: 30

Direccion de memoria en el Stack: 0x7ffeec3affcc

Valor en el Heap (via puntero): 45

Direccion de memoria en el Heap: 0x631e6f1342b0

Comentarios:

Hemos comentado en el código anterior que la variable **edadStack** en el Stack muere automáticamente al llegar la ejecución del programa a la llave de cierre del **main**, mientras que la variable **p_edadHeap** en el Head, seguiría viviendo incluso al terminar el programa, lo que nos obliga a borrarla en la última línea **delete p_edadHeap** para que el S.O. no tenga marcada esa memoria como ocupada.

Aquí te resumimos los puntos clave del análisis para que los tengas siempre a mano:

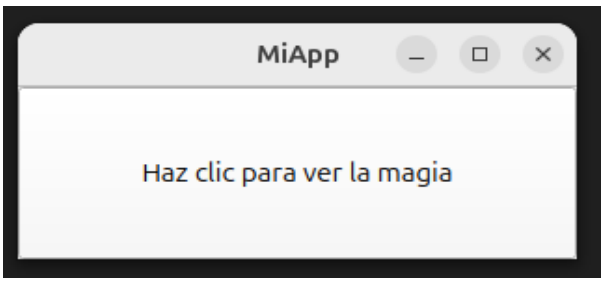
1. **Persistencia:** El Stack es como un post-it que tiras al terminar una tarea; el Heap es el archivador donde guardas lo que CimaStudio necesitará durante toda la ejecución. Que las variables se alogen en uno u otro lugar depende de cómo las creemos, sin new o con new.
2. **La magia de Qt**: Es fundamental ese matiz que mencionamos. En Qt, aunque usemos new, el sistema de parentesco de Qt (QObject) actúa como un "limpiador automático". Si un objeto tiene un padre, cuando el padre muere, el hijo se borra solo. Por eso no verás mil delete en nuestro código.
3. **Eficiencia con OpenCV**: Conectando con el punto 5.1, cv::Mat es el ejemplo perfecto de diseño híbrido: la cabecera es ligera y se mueve rápido por el Stack, mientras que los megabytes de la imagen descansan pesadamente en el Heap.

6.4.- 🔗 Lambdas_y_Slots (La comunicación moderna)

Introducción:

En el C++ clásico, para que un botón hiciera algo, necesitábamos crear una función específica y "conectarla". Con el C++ moderno (C++11 en adelante) y Qt, podemos usar las Lambdas: pequeñas funciones "anónimas" que se escriben directamente en la línea de conexión. Este concepto es vital para CimaStudio, ya que es lo que permite que el Slider de Intensidad actualice nuestra variable de brillo de forma instantánea y elegante. Aprenderemos a usar los corchetes [ ] para "capturar" el contexto y las llaves { } para ejecutar la lógica en un solo paso, eliminando el código innecesario.

Salida del programa



El botón define una ventana

Cada vez que hagas click sobre el botón aparecerń líneas como estas:

El boton se ha pulsado 1 veces.

El boton se ha pulsado 2 veces.

El boton se ha pulsado 3 veces.

main.cpp

```
#include <QApplication>
#include <QPushButton>
#include <QDebug> // Para imprimir en la consola de Qt
```

```
/**
 * PROGRAMA: 03_Lambdas_y_Slots
 * OBJETIVO: Conectar una señal (Signal) a una lógica inmediata (Slot Lambda).
 * CLAVE: Uso de la sintaxis [captura](parámetros) { código }.
 */

int main(int argc, char *argv[]) {
    QApplication app(argc, argv);

    QPushButton boton("Haz clic para ver la magia");
    boton.resize(300, 100);

    // Variable local que queremos modificar desde el botón
    int contador = 0;

    // 1. LA CONEXIÓN MAESTRA (Signal & Slot con Lambda)
    // Conectamos la señal 'clicked' del botón a una función anónima.
    // [ &contador ]: "Capturamos" la variable por referencia para poder sumarle.
    QObject::connect(&boton, &QPushButton::clicked, [&contador]() {
        contador++;
        qDebug() << "El boton se ha pulsado" << contador << "veces.";
    });

    boton.show();

    // 2. LA SUTILEZA TÉCNICA
    // Fíjate que no hemos tenido que crear una clase ni un método especial.
    // La lógica vive justo donde se necesita. En CimaStudio, esto es lo que
    // usamos en la etiqueta [C2] para el Slider.

    return app.exec();
}
```

La función lambda está contenida en las siguientes líneas:

```
QObject::connect(&boton, &QPushButton::clicked, [&contador]() {
    contador++;
    qDebug() << "El boton se ha pulsado" << contador << "veces.";
});
```

- La función lambda es lo que hay entre llaves {}
- Mediante la función **QObject::connect** enlazamos el **click** que se realice en un futuro sobre el **boton,** con la acción que se realizará como respuesta a ese click consistente en ejecutar el código de la lambda. Para ello hemos tenido que pasarle a connect los siguientes parámetros:
  - **&boton**: la dirección del boton. Si hubíeramos definido el botón como **QPushButton *boton**, entonces pasaríamos simplemente boton.
  - **&QpushButton::clicked**: el evento que se dispara al hacer click sobre el botón.
  - **[&contador]**: la dirección (y por tanto el acceso al valor) de la variable **contador** del main atrapada.

- Captura **[this]** vs **[&variable]:** En CimaStudio usamos [this] porque estamos dentro de una clase y queremos acceder a sus variables miembro (como sliderValue), mientras que aquí, al estar en un main, capturamos la variable local a partir de su dirección con **[&contador]**.
- Conexión con el Slider: Este ejemplo es el "entrenamiento" para entender la línea del tutorial: **connect(slider, &QSlider::valueChanged, this, [this](int value) {…}.** Allí slider se definió como puntero y el **value** se recibe automaticamente del slider cuando lo deslizamos.

Observación: ten en cuenta que cuando capturamos por referencia [&contador], debemos asegurarnos de que la variable (contador) siga viva cuando se pulse el botón. En el main, como app.exec() bloquea la salida, no hay problema, pero es una buena práctica recordarlo.

6.5.- 🖼️ HolaMundo_OpenCV (La Cámara y la Matriz)

Introducción:

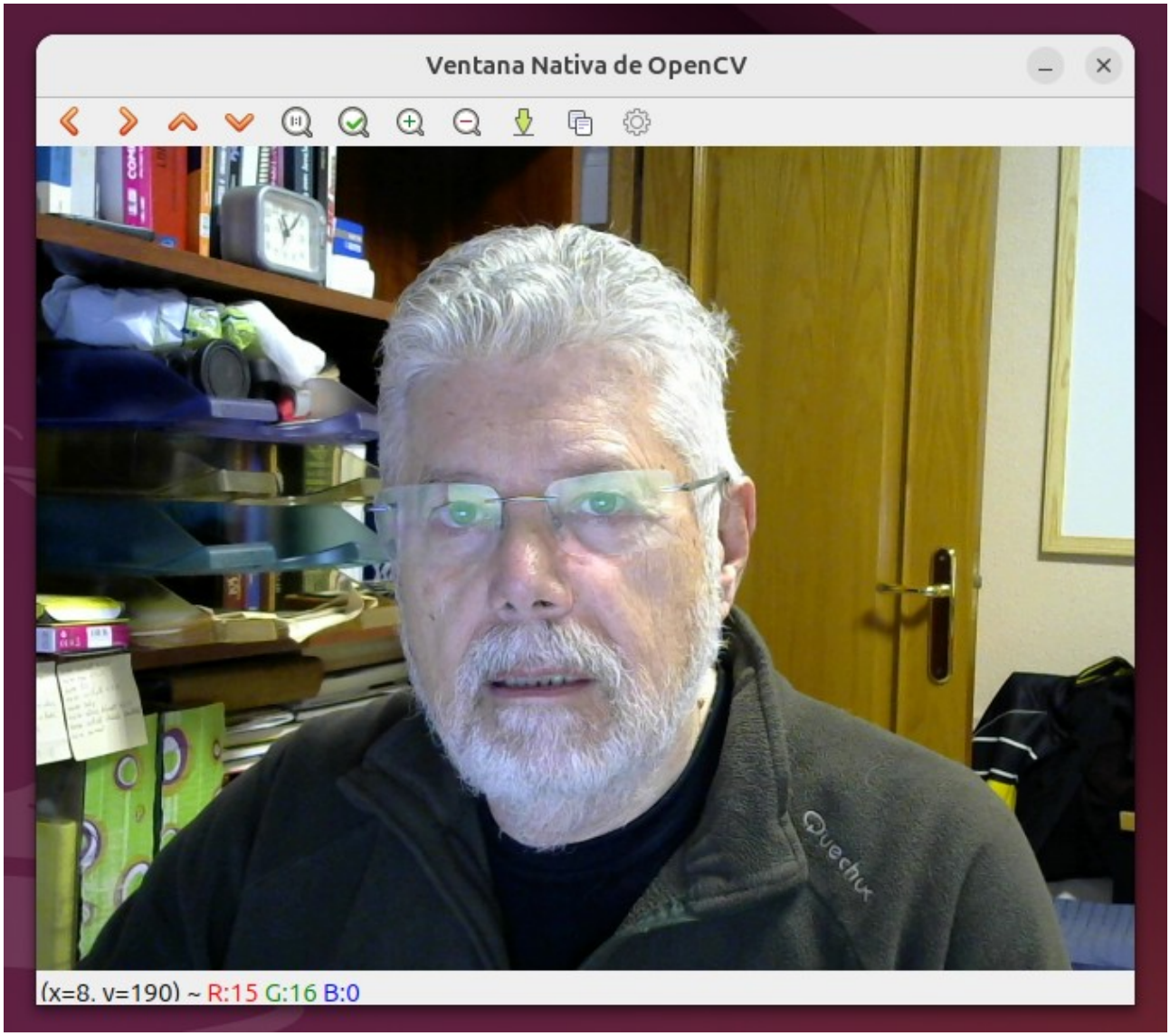
Hasta ahora hemos construido ventanas, pero ¿cómo metemos el mundo real dentro de ellas? En este ejemplo abandonamos momentáneamente Qt para entender el lenguaje de OpenCV. Aprenderemos a usar cv::VideoCapture para "despertar" la cámara y cv::Mat para capturar un instante de realidad en forma de matriz numérica. Este programa es la esencia pura del flujo de datos: capturar un frame, comprobar si contiene información y mostrarlo en una ventana nativa de OpenCV. Es el cimiento sobre el que descansa toda la lógica de procesamiento que veremos en el tutorial principal.

main.cpp

```
#include <opencv2/opencv.hpp> // Incluimos toda la potencia de OpenCV
#include <iostream>

/**
 * PROGRAMA: HolaMundo_OpenCV
 * OBJETIVO: Abrir la cámara y mostrar el flujo de video en una ventana.
 * CLAVE: El bucle while y el objeto cv::Mat (La Matriz).
 */

int main() {
    // 1. EL OJO DEL PROGRAMA
    // Creamos el objeto capturador. El '0' indica la cámara por defecto.
    cv::VideoCapture cap(0);

    // 2. EL SEGURO DE VIDA
    // Siempre debemos comprobar si el hardware ha respondido.
    if (!cap.isOpened()) {
        std::cerr << "Error: No se pudo acceder a la camara." << std::endl;
        return -1;
    }

    // 3. EL CONTENEDOR (La Sombra y el Cuerpo)
    // Reservamos la cabecera en el Stack para la matriz de píxeles.
    cv::Mat frame;

    std::cout << "Presiona cualquier tecla para salir..." << std::endl;

    // 4. EL BUCLE DE CAPTURA
    // Mientras la cámara esté abierta, "robamos" luz del sensor.
```

```
    while (true) {
        cap >> frame; // Volcamos el flujo de la cámara en la matriz 'frame'

        if (frame.empty()) break; // Si el frame viene vacío, salimos.

        // 5. LA VENTANA NATIVA
        // Mostramos la matriz en una ventana creada por OpenCV (HighGUI).
        cv::imshow("Ventana Nativa de OpenCV", frame);

        // Esperamos 30ms. Si el usuario pulsa una tecla, salimos del bucle.
        if (cv::waitKey(30) >= 0) break;
    }

    // Al salir del ámbito, 'cap' se libera automáticamente (Destructor).
    return 0;
}
```

La salida del programa:



En este ejemplo, la ventana (**cv::imshow**) que hemos creado es de OpenCV, no de Qt. Es útil para pruebas rápidas, pero en CimaStudio queremos que el video esté integrado "dentro" de nuestra

interfaz profesional. Para eso utilizamos una QMainWindow y un QLabel en su interior para renderizar el vídeo.

El objeto de este ejemplo, cv::VideoCapture cap y el operador >> son exactamente los mismos que encontrará en las etiquetas **[H1]** y **[C4]** del tutorial de CimaStudio.

6.6.-  El_Puente_QImage (Uniendo Mundos)

Introducción:

Este es, sin duda, el programa más "estratégico" del Laboratorio. Es el punto de unión donde entenderás que programar visión artificial es, en gran medida, gestionar traducciones de memoria.

OpenCV y Qt son como dos genios que hablan idiomas diferentes: el primero piensa en matrices numéricas (cv::Mat) y el segundo en lienzos de dibujo (QImage). Para que CimaStudio funcione, necesitamos un "traductor". En este ejemplo aprenderemos a construir ese puente sin duplicar los datos en la memoria (lo que sería muy lento). Utilizaremos el puntero .**data** de la matriz para decirle a Qt: "No copies la imagen, simplemente léela desde esta dirección de memoria". Es la clave técnica que permite procesar vídeo en tiempo real a 30 cuadros por segundo sin colapsar el procesador.

main.cpp

```
#include <QApplication>
#include <QLabel>
#include <QPixmap>
#include <opencv2/opencv.hpp>

/**
 * PROGRAMA: El_Puente_Qimage
 * OBJETIVO: Mostrar una imagen de OpenCV dentro de un widget de Qt.
 * CLAVE: El constructor de QImage que usa el puntero .data de cv::Mat.
 */

int main(int argc, char *argv[]) {
    QApplication app(argc, argv);

    // 1. CARGAMOS LA IMAGEN CON OPENCV
    // OpenCV carga por defecto en formato BGR (Blue-Green-Red).
    cv::Mat matImg = cv::imread("imagen.jpg");
    if (matImg.empty()) return -1;

    // 2. EL PUENTE DE TRADUCCIÓN (Crucial)
    // Creamos una QImage que "apunta" a los mismos píxeles que la matriz.
    // Argumentos: (puntero a datos, ancho, alto, bytes por fila, formato).
    QImage qimg(matImg.data,
                matImg.cols,
                matImg.rows,
                static_cast<int>(matImg.step),
                QImage::Format_RGB888);

    // 3. LA SUTILEZA: CORRECCIÓN DE COLOR
    // Qt espera RGB, pero OpenCV entrega BGR. Debemos intercambiar los canales.
    // .rgbSwapped() nos devuelve la imagen con los colores correctos.
    QImage qimgCorrecta = qimg.rgbSwapped();

    // 4. PROYECCIÓN EN LA INTERFAZ
    // Usamos un QLabel como "pantalla". Convertimos QImage a QPixmap para pintarla.
```

```
    QLabel lienzo;
    lienzo.setPixmap(QPixmap::fromImage(qimgCorrecta));
    lienzo.setWindowTitle("Puente Qt + OpenCV");
    lienzo.show();

    return app.exec();
}
```

La salida del programa:

Ventana mostrada desde Qt

Notas para tu revisión en papel:

- **La Eficiencia**: QImage qimg(...) no hace una copia de los píxeles; es una "vista" de la memoria de cv::Mat. Esto es lo que permite la velocidad de CimaStudio.
- **El rgbSwapped()**: Es una duda común de los alumnos. Si no lo haces, la gente saldrá en la pantalla con la cara azul (como pitufos) porque los canales Rojo y Azul están invertidos entre librerías.
- **Conexión con CimaStudio**: Este bloque es la base de la etiqueta [C10] que verán en el tutorial principal.

A continuación veremos el último peldaño del Laboratorio, el que separa al usuario de librerías del programador de algoritmos. Aquí es donde el alumno entiende que una imagen no es un objeto mágico, sino una simple cuadrícula de números que puede manipular a su antojo.

6.7.- 🔍 Escaneo_Matrices (Acceso a los Píxeles)

Introducción:

Este es el último peldaño del Laboratorio, el que separa al usuario de librerías del programador de algoritmos. Aquí es donde entenderás que una imagen no es un objeto mágico, sino una simple cuadrícula de números que puedes manipular a tu antojo.

Hasta ahora hemos dejado que OpenCV haga todo el trabajo con funciones prefabricadas. Pero, ¿qué pasa si queremos crear nuestro propio filtro o entender cómo se "barre" una imagen? En este ejemplo bajaremos al nivel más profundo: el píxel. Aprenderemos a recorrer una cv::Mat usando bucles for anidados (filas y columnas). Utilizaremos el método **.at<uchar>(y, x)** para leer y escribir valores directamente en la memoria. Este concepto es la base de cualquier algoritmo de visión artificial: entender que procesar una imagen es, en esencia, realizar miles de operaciones matemáticas sencillas sobre una rejilla de datos.

Salida del programa:

Ventana de OpenCV

main.cpp

```
#include <opencv2/opencv.hpp>
```

```cpp
#include <iostream>

/**
 * PROGRAMA: 06_Escaneo_Matrices
 * OBJETIVO: Recorrer una imagen píxel a píxel y modificar su brillo manualmente.
 * CLAVE: Uso de .at<uchar>(y, x) para lectura y escritura directa.
 */

int main() {
    // 1. CARGAMOS UNA IMAGEN EN GRISES (1 solo canal)
    cv::Mat img = cv::imread("../lena.jpg", cv::IMREAD_GRAYSCALE);
    if (img.empty()) return -1;

    // 2. EL BUCLE DE ESCANEO (Doble For)
    // Recorremos cada fila (y) y cada columna (x)
    for (int y = 0; y < img.rows; y++) {
        for (int x = 0; x < img.cols; x++) {

            // 3. LEEMOS EL VALOR ACTUAL
            // <uchar> indica que el píxel es un byte (0-255)
            uchar pixel = img.at<uchar>(y, x);

            // 4. MODIFICAMOS EL PÍXEL (Aumentamos brillo)
            // Sumamos 50, pero usamos cv::saturate_cast para no pasarnos de 255
            img.at<uchar>(y, x) = cv::saturate_cast<uchar>(pixel + 50);
        }
    }

    // 5. RESULTADO
    cv::imshow("Imagen Procesada Manualmente", img);
    cv::waitKey(0);

    return 0;
}
```

Recorrer la matriz mediante **.at<>** es muy seguro y fácil de entender para empezar, aunque en CimaStudio, OpenCV usa métodos más rápidos (como punteros de fila **.ptr<>**) de forma interna.

El **saturate_cast**: Es un detalle de "Ingeniero". Si a un píxel de valor 250 le sumas 10, en un byte daría la vuelta a 4 (negro). saturate_cast lo clava en 255 (blanco), evitando errores visuales.

Conexión con **CimaStudio**: Este ejemplo **da sentido a la sección 5.2 (Convoluciones),** ya que el alumno visualiza físicamente la cuadrícula sobre la que operan los kernels.

7.- Horizontes de CimaStudio: El siguiente nivel

En esta sección final, dejamos de mirar el código actual para proyectar el potencial de nuestra aplicación. El procesamiento de imagen es exigente; aquí es donde buscamos la eficiencia pura.

7.1.- Multitarea: El arte de no bloquear la mirada

Hasta ahora, CimaStudio funciona de forma secuencial: captura, procesa y muestra. Pero, ¿qué pasa si el proceso es muy pesado?

- **El concepto**: Separar la Interfaz de Usuario (UI) de la lógica de procesamiento.

- **La clave**: Utilizar Hilos (Threads). Mientras un hilo "escucha" si mueves el slider, otro hilo independiente se pelea con los píxeles. Esto evita que la ventana se congele o aparezca el temido "No responde".

7.2.- CUDA: Desatando la potencia del silicio

Si la CPU es un matemático muy inteligente, la GPU (tarjeta gráfica) es un ejército de miles de calculadores trabajando a la vez.

- **El concepto**: Paralelismo masivo. En lugar de procesar los píxeles uno por uno (como un escáner), enviamos la imagen a la tarjeta gráfica para que procese todos los píxeles simultáneamente.
- **La clave**: Usar el módulo **cv::cuda** de OpenCV. La diferencia de velocidad puede ser de hasta 10 o 20 veces más rápido, permitiendo filtros complejos en tiempo real que antes eran imposibles.frase final de cierre para el tutoria

"Recuerda que cada gran programa comenzó con un simple 'Hola Mundo'. Hoy has construido un sistema complejo de Signals, Slots y Píxeles. No te detengas aquí: rompe el código, experimenta y descubre. El arte de enseñar a ver a las máquinas está ahora en tus manos."