Desarrollo de una herramienta de procesamiento de imágenes con C++ y OpenCV

Informática Gráfica, 2022-2023

Ingeniería en Informática

Intensificación en Ingeniería del Software

Universidad de Murcia

Alberto Gálvez Gálvez – alberto.galvezg@um.es

Javier Jiménez Hernández – javier.jimenezh@um.e

Tabla de contenido

1 Introducción

El procesamiento de imágenes es una técnica que permite modificar o analizar imágenes digitales para obtener información útil o para mejorar su apariencia. Una de las librerías más populares para el procesamiento de imágenes es *OpenCV*, que es una biblioteca de código abierto disponible en varios lenguajes, como son *Python* y *C++*. Este trabajo consiste en el desarrollo de una aplicación de procesamiento de imágenes con el lenguaje de programación *C++* y la librería *OpenCV*. Esta aplicación es capaz de poder abrir varias imágenes simultáneamente, pudiendo aplicarse operaciones sobre cualquiera de ellas. Algunas de estas operaciones son ecualizar el histograma, pintar, obtener información, obtener su espectro, realizar una transformación de curva tonal genérica. Las distintas partes en las que se divide nuestro trabajo son:

- Diseño de la aplicación: expone en líneas generales la metodología de desarrollo de las funcionalidades.
- Solución: explica cómo se han implementado las funcionalidades más destacables.
- Listado de código: ilustra mediante imágenes el código de procesamiento de las funcionalidades del apartado anterior.
- Pruebas y ejemplos de uso: muestra distintas pruebas y ejemplos de uso para demostrar el correcto funcionamiento de nuestro programa.
- Conclusiones: en el que se recapitula lo desarrollado a través de los demás apartados y se expone nuestra opinión sobre el trabajo realizado.

2 Diseño de la aplicación

La aplicación, desarrollada con *C++* y *OpenCV*, se divide en módulos (un fichero con extensión *.h* con la definición de las operaciones, y otro con *.cpp* con la implementación de estas operaciones). Algunos de estos módulos son, por ejemplo, *perspectiv*a, *ajustelineal* o *brillocontraste*. Con respecto al desarrollo de cada módulo, los pasos para su creación han sido los siguientes:

1. Crear el módulo si es necesario, es decir, si se necesita una nueva ventana para ello.
2. Incluir la definición de la nueva operación, ya sea en su *.h* correspondiente si se ha creado un nuevo módulo, o en el módulo que corresponda (por ejemplo, *imagenes.h*).
3. Crear una nueva ventana *QTDesignerFormClass* o ajustar la interfaz existente según corresponda.
4. Incluir la implementación de la funcionalidad en el *.cpp* correspondiente, siguiendo el esquema del punto anterior.
5. Añadir el acceso a la nueva funcionalidad en la ventana *mainwindow.ui*.

3 Solución

A continuación, en orden, se puede ver los pasos seguidos para el módulo de procesamiento de cada una de las funcionalidades extras desarrolladas.

3.1 Ver información

Para enseñar la información de la imagen, lo que se ha realizado es una ventana información que recibe como parámetro en su constructor la imagen que tenemos abierta, una vez se ha comprobado que haya una imagen activa se hacen los siguientes pasos:

1. Se extrae el número total de elementos con *total()*.
2. Se extrae la profundidad de una imagen con *depth*().
3. Se extrae el número de canales con channel*s()*.
4. Para sacar la memoria en bytes se obtienen el número de elementos con *total()* y se multiplica con el tamaño de cada elemento *elemSize()*.
5. Para calcular el color medio se ha calculado, con la función *mean()*, una media de la imagen. El resultado de esta media se encuentra en la primera posición del array valor, ya que devuelve un tipo *Scalar*.

Al finalizar estos pasos se muestra una ventana con estas características.

3.2 Ecualización del histograma

Para ecualizar el histograma se ha identificado dos casos, ecualizar cada canal por separado y en conjunto. Para ello se han creados dos funciones *imagenes.cpp* para cada caso.

Ecualiza histograma por canales. Para realizar este proceso:

1. Se ha definido un vector de tres posiciones, uno para cada vector.
2. Se separa la imagen según en los tres canales con *split* y se guardan en el vector.
3. Con la función *equalizeHist* se ecualiza cada canal por separado.
4. Con *merge* devolvemos los canales modificados a la imagen resultado.
5. Almacenamos el resultado de esta operación creando una nueva imagen mediante el método *crear_nueva*.

Ecualizar histograma conjuntamente. Para ecualizar el histograma conjuntamente se ha realizado los siguientes pasos:

1. Primero se pasa la imagen fuente a gris.
2. Se calcula su histograma y se normaliza.
3. Se realiza la integral sobre el histograma para ecualizarla.
4. Se pasa la imagen *lut* a resultado ya con su histograma actualizado.

5. Almacenamos el resultado de esta operación creando una nueva imagen mediante el método *crear_nueva*.

3.3 Ecualización local

Para realizar la ecualización local de una imagen con *OpenCV* y *C++*, hemos realizado los siguientes pasos:

1. Creamos un objeto de la clase Ptr<*CLAHE*> utilizando el método *createCLAHE* para más tarde poder utilizar el método *apply* para aplicar la ecualización local a la imagen.
2. Establecemos el límite de clip para la ecualización local utilizando el método *setClipLimit*. El límite de clip es un valor que indica el máximo número de píxeles en un área local que pueden ser ecualizados.
3. Cargamos la imagen a ecualizar localmente en un objeto *Mat* utilizando el método *imread.*
4. Transformamos la imagen al formato *LAB* mediante la función *cvtColor*
5. Extraemos el canal *L* mediante el método *split* y lo ecualizamos localmente utilizando el método *apply.*
6. Una vez ecualizado el canal *L*, pasamos el nuevo valor de este a la imagen original mediante el método *merge.*
7. Transformamos la imagen ecualizada al formato *BGR* mediante el método *cvtColor*.

Cabe destacar que hemos usado el formato LAB para la ecualización del histograma debido a que permite una representación de colores más amplia y precisa que otros formatos como el RGB, mejor visualización de la distribución de los colores.

3.4 Convertir a color falso

1. Creamos dos imágenes.
2. En una de las imágenes creadas, almacenamos la imagen de entrada transformándola a escala de grises mediante la función *cvtColor* utilizando el flag *COLOR_BGR2GRAY*.
3. Ahora la imagen en escala de grises, la transformamos a escala de color de nuevo mediante la función *cvtColor* utilizando el flag *COLOR_GRAY2BGR.*
4. Aplicamos la escala de color falsa a la imagen mediante la función *applyColorMap* y el flag *COLORMAP_HSV*.
5. Almacenamos el resultado de esta operación en la otra imagen creada (aun no usada) mediante el método *crear_nueva*.

3.5 Modelos de color

1. Suponemos que el modelo que tiene la imagen por defecto tiene el modelo de *OpenCV(BGR*) ya que si no se tendría que identificar numerosos casos.

2. A través de una ventana que hemos definido se selecciona el modelo al que queremos cambiar.
3. Se comprueba mediante un switch que valor del modelo hemos escogido y se aplica la transformación a través de la función *cvtColor*() con la flag correspondiente al modelo seleccionado.
4. Se muestra una previsualización de la imagen para comprobar su resultado y si el booleano de guardar tiene valor true se establece como modificada para guardarla.

3.6 Nueva desde el portapapeles

1. Se crea un *clipboard*.
2. Se comprueba que hemos seleccionado en el portapapeles una imagen.
3. Se comprueba el formato de la *QImage* para crear un *Mat* con el tipo adecuado.
4. A través de *crear_nueva()* se abre una ventana con la imagen seleccionada.

3.7 Copiar al portapapeles

1. Se comprueba si hay una imagen activa.
2. Si hay una imagen activa se copia su *ROI*, si no tiene se escoge la imagen entera.
3. Se comprueba el tipo de imagen es, ya que dependiendo del tipo se necesitará crear una *QImage* con el formato adecuado.
4. Se crea una *QImage*.
5. Se crea un *clipboard*.
6. A través de *setImage()* se le pasa como parámetro la *QImage* creada y se establece en el portapapeles.

3.8 Ajustar RGB

1. Se comprueba si la imagen es de un canal, si es de un canal se pasa a una imagen con tres canales.
2. Si la opción selecciona es sumar (*valor = 0*) se suman los valores de los canales según la constante que el usuario ha seleccionado.
3. Si la opción selecciona es multiplicar (*valor = 1*) se multiplican los valores de los canales según la constante que el usuario ha seleccionado.

3.9 Balanceo de blancos

1. Cambiamos el modelo de color de la imagen *BGR* a *YUV*.
2. Sacamos los canales por separado de *YUV*.
3. Calculamos a través de la función mean la media de los canales *u* e *y*.
4. Comprobamos la diferencia que faltan para que alcance el valor *128* en cada canal.
5. Le sumamos esa diferencia ya sea positiva o negativa, a los canales *u* e *y*.
6. Devolvemos la imagen al modelo *BGR*.

3.10 Espectro

1. Creamos dos imágenes.
2. Guardamos la imagen de entrada en un formato adecuado para su análisis espectral en una de las imágenes creadas. Para ello, usamos la función *cvtColor()* e indicando el flag *COLOR_BGR2GRAY*.
3. Guardamos en la otra imagen creada, la imagen en escala de grises escalándola (para una mejor visualización) y convirtiéndola a un solo canal con profundidad real mediante el método *convertTo* indicando el flag *CV_32FC1* y el factor de escala *1.0/255*.
4. A esta nueva imagen, le aplicamos la transformada de Fourier discreta usando la función *dft*, así obteniendo el espectro sin centrar de la imagen original en una nueva imagen.
5. Obtenemos los dos canales de esta nueva imagen mediante el método *Split*, calculamos su módulo mediante la función *pow* y le damos este nuevo valor a la imagen.
6. Guardamos en una nueva imagen con formato *CV_8UC1*, el espectro de la imagen sin centrar.
7. Centramos la imagen intercambiando el primer por el tercer cuadrante y el segundo por el cuarto cuadrante, para una mejor visualización.

3.11 Rellenar

1. En *imágenes.h* se ha añadido al enumerado de las herramientas la herramienta rellenar.
2. En *imágenes.cpp* en la función del callback se ha añadido al switch un caso para cuando la herramienta rellenar este seleccionada.
3. Al pulsar el botón izquierdo del botón se lama a la función de *cb_rellenar*.
4. Se clona la imagen actual.
5. Se establecen los rangos según el radio del pincel.
6. Se establece el color a rellenar según el color del pincel
7. Se establece la transparencia a través del difuminado del pincel a través de una media ponderada.
8. Con *floodFill* se rellena la imagens egún los parámetros defintivos.
9. Se muestra la imagen resultante.

3.12 Trazo

1. En *imágenes.h* se ha añadido al enumerado de las herramientas la herramienta rellenar.
2. En *imágenes.cpp* en la función del callback se ha añadido al switch un caso para cuando la herramienta rellenar este seleccionada.
3. Al mantener el botón izquierdo abajo del ratón mientras se mueve se va dibujando el trazo, tal como en la herramienta punto.

4. Para saber que pares de puntos dibujan la línea al dibujar un círculo y la línea para unirlos, se han definido dos variables globales (anteriorx, anteriory) que se inicializan cuando se empieza a hacer click izquierdo. Estas variables, mientras se mantiene el click izquierdo, actualizando dice van actualizando con la posición del puntero del ratón, así realizando el trazo.

4 Listado de código

4.1 Ver información

```
Informacion::Informacion(Mat img,QWidget *parent) :
    QDialog(parent),
    ui(new Ui::Informacion)
{
    Scalar valor=mean(img); //Se calcula la media y se obtiene el scalar
    ui->setupUi(this);
    ui->lineEdit->setText(QString::number(img.total()));
    ui->lineEdit_2->setText(QString::number(img.depth()));
    ui->lineEdit_3->setText(QString::number(img.channels()));
    ui->lineEdit_4->setText(QString::number(img.elemSize()*img.total()));
    ui->lineEdit_5->setText(QString::number(valor.val[0]));
}

Informacion::~Informacion()
{
    delete ui;
}
```

Ilustración 1. Módulo de procesamiento de la funcionalidad ver información disponible en el fichero verinformacion.cpp

4.2 Ecualización del histograma

```
void ecualizar_hist_separado(int nfoto, int nres)
{
    Mat res;
    vector<Mat> canales;
    split(foto[nfoto].img,canales);
    equalizeHist(canales[0],canales[0]);
    equalizeHist(canales[1],canales[1]);
    equalizeHist(canales[2],canales[2]);
    merge(canales,res);

    crear_nueva(nres,res);

}
```

Ilustración 2. Método ecualizar_hist_separado de la funcionalidad ecualizar histograma disponible en imagenes.cpp

```
void ecualizar_hist_conjunto(int nfoto, int nres)
{
    Mat gris, hist;
    cvtColor(foto[nfoto].img, gris, COLOR_BGR2GRAY);
    int canales[1]= {0}, bins[1]= {256};
    float rango[2]= {0, 256};
    const float *rangos[]= {rango};
    calcHist(&gris, 1, canales, noArray(), hist, 1, bins, rangos);
    hist*= 255.0/norm(hist, NORM_L1);
    Mat lut(1, 256, CV_8UC1);
    float acum= 0.0;
    for (int i= 0; i<256; i++) {
        lut.at<uchar>(0, i)= acum;
        acum+= hist.at<float>(i);
    }
    Mat res;
    LUT(foto[nfoto].img, lut, res);
    crear_nueva(nres,res);

}
```

Ilustración 3. Método ecualizar_hist_conjunto de la funcionalidad ecualizar histograma disponible en imagenes.cpp

4.3 Ecualización local

```
void ecualizacion_local_hist(int nfoto, int radio,bool guardar)
{
    // Crear un objeto de la clase cv::Ptr<cv::CLAHE>
    cv::Ptr<cv::CLAHE> clahe = cv::createCLAHE();

    // Establecer el límite de clip para la ecualización
    clahe->setClipLimit(radio);

    // Aplicar la ecualización a la imagen original
    Mat lab_image;
    cvtColor(foto[nfoto].img,lab_image,COLOR_BGR2Lab);
    vector<Mat> lab_channels(3);
    split(lab_image, lab_channels);  //Sacamos los canales
    clahe->apply(lab_channels[0], lab_channels[0]); //Aplicamos la ecualización local para el canal l
    merge(lab_channels,lab_image); //Juntamos de nuevo los canales
    Mat res;
    cvtColor(lab_image,res,COLOR_Lab2BGR);
    // Mostrar la imagen ecualizada
    imshow(foto[nfoto].nombre,res);
    if(guardar){
        res.copyTo(foto[nfoto].img);
        foto[nfoto].modificada=true;
    }
}
```

Ilustración 4. Método ecualización_local_hist de la funcionalidad ecualizar histograma localmente disponible en imagenes.cpp

4.4 Convertir a color falso

```
void escala_color_falso(int nfoto,int nres)
{
    Mat gris;
    Mat res;
    cvtColor(foto[nfoto].img,gris,COLOR_BGR2GRAY);
    cvtColor(gris,gris,COLOR_GRAY2BGR);
    applyColorMap(gris,res,COLORMAP_HSV);
    crear_nueva(nres,res);
}
```

Ilustración 5. Método escala_color_falso de la funcionalidad convertir a color falso disponible en imagenes.cpp

4.5 Modelos de color

```
void cambiar_modelo(int nfoto,int modelo,bool guardar)
{
    Mat res;
    switch (modelo)
    {
    case 0:
        cvtColor(foto[nfoto].img,res,COLOR_BGR2RGB);
        break;
    case 1:
        cvtColor(foto[nfoto].img,res,COLOR_BGR2HLS);
        break;
    case 2:
        cvtColor(foto[nfoto].img,res,COLOR_BGR2HSV);
        break;
    case 3:
        cvtColor(foto[nfoto].img,res,COLOR_BGR2XYZ);
        break;
    case 4:
        cvtColor(foto[nfoto].img,res,COLOR_BGR2YUV);
        break;
    }
    imshow(foto[nfoto].nombre,res);
    if(guardar){
        res.copyTo(foto[nfoto].img);
        foto[nfoto].modificada=true;
    }

}
```

Ilustración 6. Método cambiar_modelo de la funcionalidad cambiar modelo disponible en imagenes.cpp

4.6 Abrir nueva desde el portapapeles

```
void MainWindow::on_actionNuevo_desde_portapapeles_triggered()
{
    QClipboard* clipboard = QApplication::clipboard();
    QImage image = clipboard->image();
    if(image.isNull())
        return;
    switch(image.format())
    {
    case QImage::Format_ARGB32:
    case QImage::Format_ARGB32_Premultiplied:
    {
        Mat mat(image.height(), image.width(), CV_8UC4,(uchar*)image.bits(),image.bytesPerLine());
        Mat sinAlpha;
        cvtColor(mat,sinAlpha,COLOR_BGRA2BGR);
        crear_nueva(primera_libre(),sinAlpha);
        break;
    }
    case QImage::Format_Grayscale8:
    case QImage::Format_Indexed8:
    {
        Mat gris(image.height(), image.width(), CV_8UC1,(uchar*)image.bits(),image.bytesPerLine());
        Mat mat;
        cvtColor(gris,mat,COLOR_GRAY2BGR);
        crear_nueva(primera_libre(),mat);
        break;
    }
    case QImage::Format_RGB32:
    {
        Mat mat(image.height(), image.width(), CV_8UC4,(uchar*)image.bits(),image.bytesPerLine());
        Mat sinAlpha;
        cvtColor(mat,sinAlpha,COLOR_BGRA2BGR);
        crear_nueva(primera_libre(),sinAlpha);
        break;
    }
    case QImage::Format_RGB888:
    {
        Mat mat(image.height(), image.width(), CV_8UC3,(uchar*)image.bits(),image.bytesPerLine());
        Mat reves;
        cvtColor(mat,reves,COLOR_RGB2BGR);
        crear_nueva(primera_libre(),reves);
        break;
    }

    }

}
```

Ilustración 7. Funcionalidad de abrir nueva desde el portapapeles disponible en mainwindow.cpp

4.7 Copiar al portapapeles

```
void MainWindow::on_actionCopiar_a_portapapeles_triggered()
{
    if(foto_activa()!=-1){
        int num= foto_activa();
        Mat imgres=foto[num].img(foto[num].roi).clone();

        if(imgres.type()==CV_8UC1)
        {
            //Se crea una Qimagen con escala de grises
            QImage img(imgres.data, imgres.cols, imgres.rows, imgres.step, QImage::Format_Indexed8);
            //Se crea clipboard
            QClipboard* clipboard = QApplication::clipboard();
            //Se añade al portapapeles
            clipboard->setImage(img);
        }

        else if(imgres.type()==CV_8UC3)
        {
            Mat img_clipboard;
            cvtColor(imgres, img_clipboard, COLOR_BGR2RGB);

            // Crea una imagen QImage a partir de la imagen de OpenCV RGB
            QImage img(img_clipboard.data,img_clipboard.cols, img_clipboard.rows,img_clipboard.step, QImage::Format_RGB888);
            //Se crea clipboard
            QClipboard* clipboard = QApplication::clipboard();
            //Se añade al portapapeles
            clipboard->setImage(img);
        }

    }
}
```

Ilustración 8. Funcionalidad de copiar al portapapeles disponible en mainwindow.cpp

4.8 Ajustar RGB

```
void ajustar_color_canales(int nfoto, int constante,int opcion,bool guardar)
{
    if(foto[nfoto].img.type()==CV_8UC1)
        cvtColor(foto[nfoto].img,foto[nfoto].img,COLOR_GRAY2BGR);

    if(opcion==0)
    {
        Mat res=foto[nfoto].img+Scalar(constante,constante,constante);
        imshow(foto[nfoto].nombre,res);
        if(guardar){
            res.copyTo(foto[nfoto].img);
            foto[nfoto].modificada=true;
        }

    }
    else {
        Mat res=foto[nfoto].img.mul(Scalar(constante,constante,constante));
        imshow(foto[nfoto].nombre,res);
        if(guardar){
            res.copyTo(foto[nfoto].img);
            foto[nfoto].modificada=true;
        }
    }

}
```

Ilustración 9. Método ajustar_color_canales de la funcionalidad ajustar RGB disponible en imagenes.cpp

4.9 Balanceo de blancos

```
void balance_blancos(int nfoto, int nres)
{
    Mat res,espacio_YUV;
    cvtColor(foto[nfoto].img,espacio_YUV,COLOR_BGR2YUV);
    vector<Mat> yuv_channels(3);
    split(espacio_YUV, yuv_channels);
    Scalar media_u=mean(yuv_channels[1]);
    Scalar media_y=mean(yuv_channels[2]);
    double modificar_u=128.0-media_u.val[0];
    double modificar_y=128.0-media_y.val[0];
    espacio_YUV+=Scalar(0,modificar_u,modificar_y);
    cvtColor(espacio_YUV,res,COLOR_YUV2BGR);
    crear_nueva(nres,res);
}
```

Ilustración 10. Método balance_blancos de la funcionalidad balance de blancos disponible en imagenes.cpp

4.10 Espectro

```
void generar_espectro(int nfoto, int nres)
{
    Mat gris;
    Mat escala;
    cvtColor(foto[nfoto].img,gris,COLOR_BGR2GRAY); //Pasamos a escala de grises la imagen
    gris.convertTo(escala, CV_32FC1, 1.0/255);
    Mat imagenDFT;
    dft(escala, imagenDFT, DFT_COMPLEX_OUTPUT); //Se aplica DFT
    vector<Mat> canales;
    split(imagenDFT, canales);
    pow(canales[0], 2, canales[0]);
    pow(canales[1], 2, canales[1]);
    pow(canales[0]+canales[1], 0.5, imagenDFT);
    Mat res;
    imagenDFT.convertTo(res, CV_8UC1, -1, 255);
    int cx = res.cols/2;
    int cy = res.rows/2;
    Mat q0(res, Rect(0, 0, cx, cy));   // Esquina superior izquierda
    Mat q1(res, Rect(cx, 0, cx, cy));  // Esquina superior derecha
    Mat q2(res, Rect(0, cy, cx, cy));  // Esquina inferior izquerda
    Mat q3(res, Rect(cx, cy, cx, cy)); // Esquina inferior derecha
    Mat tmp;                           // Cambiar cuadrantes (Esquina superior izquierda con Esquina inferior derecha)
    q0.copyTo(tmp);
    q3.copyTo(q0);
    tmp.copyTo(q3);
    q1.copyTo(tmp);                    // cambiar cuadrantes (Esquina superior derecha con Esquina inferior izquerda)
    q2.copyTo(q1);
    tmp.copyTo(q2);
    crear_nueva(nres,res);
}
```

Ilustración 11. Método generar_espectro de la funcionalidad generar espectro disponible en imagenes.cpp

4.11 Rellenar

```
void cb_rellenar (int factual, int x, int y)
{
    Mat rellenado=foto[factual].img.clone();
    Mat res;
    Scalar lo=Scalar(radio_pincel+1,radio_pincel+1,radio_pincel+1);
    Scalar up=Scalar(radio_pincel+1,radio_pincel+1,radio_pincel+1);
    Scalar newVal=Scalar(color_pincel.val[0],color_pincel.val[1],color_pincel.val[2]);
    floodFill(rellenado,Point(x,y),newVal,NULL,lo,up,FLOODFILL_FIXED_RANGE);
    double ponderacion=(difum_pincel+1)/121.0;
    addWeighted(foto[factual].img,ponderacion,rellenado,1-ponderacion,0,res);
    imshow(foto[factual].nombre, res);
    foto[factual].img=res;
    foto[factual].modificada= true;
}

//-----------------------------------------------------------------------

void cb_ver_rellenar (int factual, int x, int y)
{
    Mat rellenado=foto[factual].img.clone();
    Mat res;
    Scalar lo=Scalar(radio_pincel+1,radio_pincel+1,radio_pincel+1);
    Scalar up=Scalar(radio_pincel+1,radio_pincel+1,radio_pincel+1);
    Scalar newVal=Scalar(color_pincel.val[0],color_pincel.val[1],color_pincel.val[2]);
    floodFill(rellenado,Point(x,y),newVal,NULL,lo,up,FLOODFILL_FIXED_RANGE);
    double ponderacion=(difum_pincel+1)/121.0;
    addWeighted(foto[factual].img,ponderacion,rellenado,1-ponderacion,0,res);
    imshow(foto[factual].nombre, res);
}
```

Ilustración 12. Métodos cb_rellenar y cb_ver_rellenar de la funcionalidad rellenar disponible en imagenes.cpp

4.12 Trazos

```
void cb_trazo(int factual, int x, int y)
{
    Mat im= foto[factual].img;  // Ojo: esto no es una copia, sino a la misma imagen
    if (difum_pincel==0){
        circle(im, Point(x, y), radio_pincel+1, color_pincel, -1, LINE_AA);
        line(im, Point(anteriox, anteriory), Point(x,y), color_pincel, radio_pincel+1);
        anteriox=x;
        anteriory=y;
    }
    else {
        int tam=radio_pincel+difum_pincel;
        int posx=tam,posy=tam;
        Rect roi(x-tam,y-tam,2*tam+1,2*tam+1);
        if(roi.x<0){
            roi.width+=roi.x;
            posx+=roi.x;
            roi.x=0;
        }
        if(roi.y<0){
            roi.height+=roi.y;
            posy+=roi.y;
            roi.y=0;
        }
        if(roi.x+roi.width > im.cols){
            roi.width=im.cols-roi.x;
        }
        if(roi.y+roi.height > im.rows){
            roi.height=im.rows-roi.y;
        }
        Mat frag=im(roi);
        Mat res(frag.size(), frag.type(), color_pincel);
        Mat cop(frag.size(), frag.type(), CV_RGB(0,0,0));
        circle(cop, Point(posx, posy), radio_pincel+1, CV_RGB(255,255,255), -1, LINE_AA);
        line(cop, Point(anteriox, anteriory), Point(x,y), CV_RGB(255,255,255), radio_pincel+1);
        anteriox=x;
        anteriory=y;
        blur(cop, cop, Size(difum_pincel*2+1, difum_pincel*2+1));
        multiply(res, cop, res, 1.0/255.0);
        bitwise_not(cop, cop);
        multiply(frag, cop, frag, 1.0/255.0);
        frag= res + frag;
    }
    imshow(foto[factual].nombre, im);
    foto[factual].modificada= true;
}
```

Ilustración 13. Método cb_trazo de la funcionalidad trazo disponible en imagenes.cpp

5 Pruebas y ejemplos de uso

5.1 Ver información

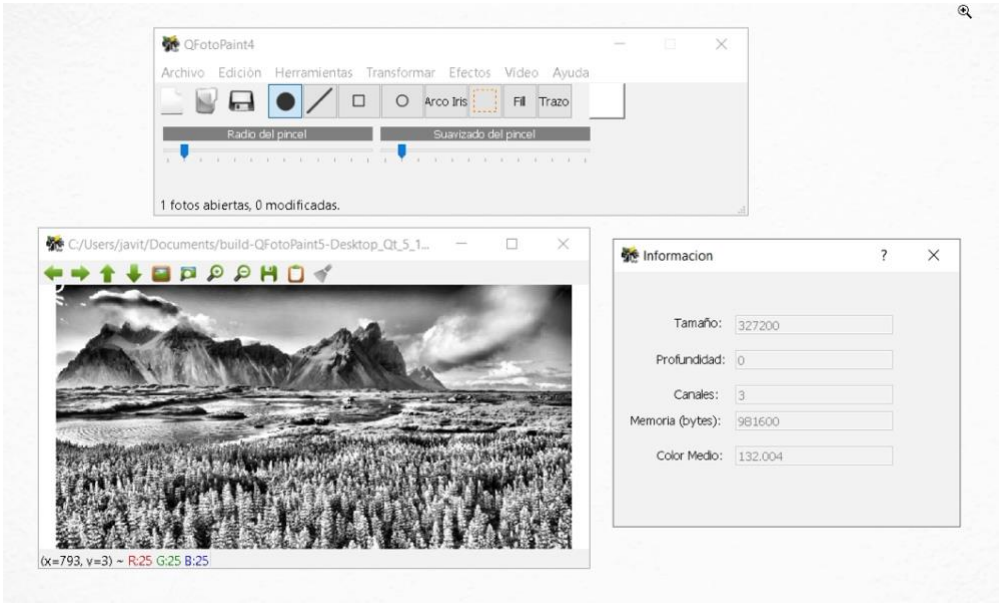

Ilustración 14. Ejemplo de uso de la funcionalidad ver información

5.2 Ecualización local

Ilustración 15. Imagen de entrada para la funcionalidad ecualización local

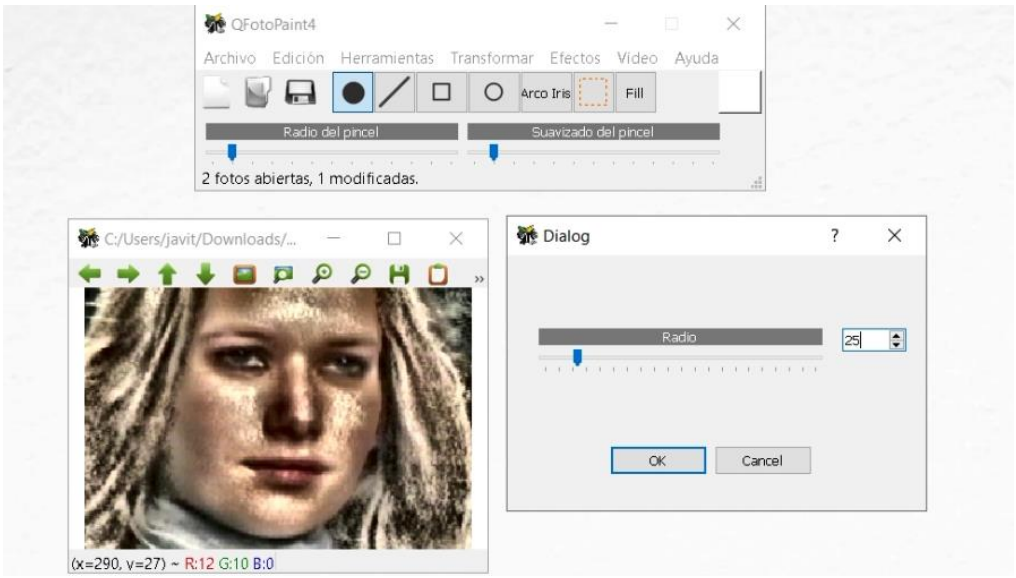

Ilustración 16. Ejemplo de uso de la funcionalidad ecualización local

5.3 Ajustar RGB

Ilustración 17. Imagen de entrada para la funcionalidad ajustar RGB

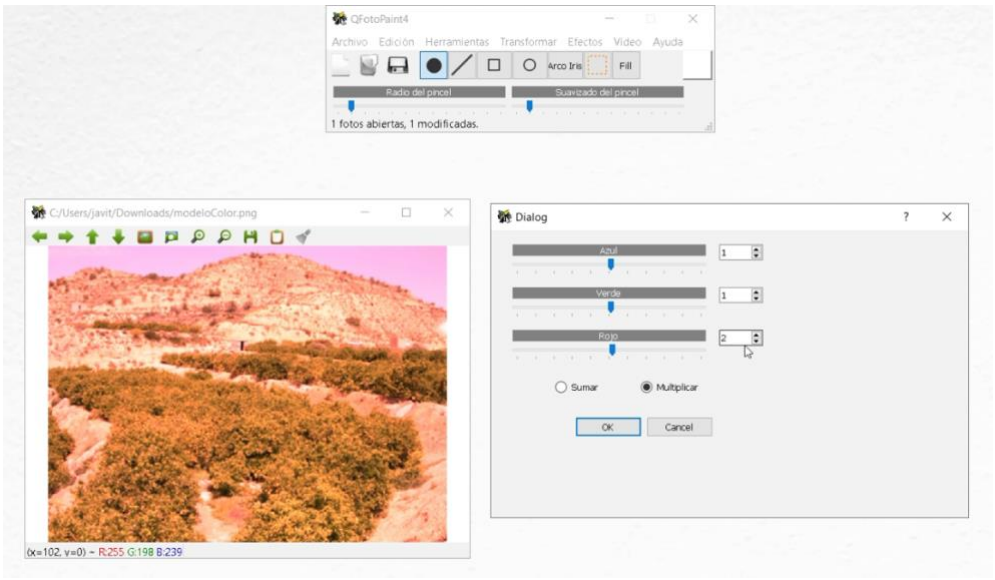

Ilustración 18. Ejemplo de uso de la funcionalidad ajustar RGB

5.4 Espectro

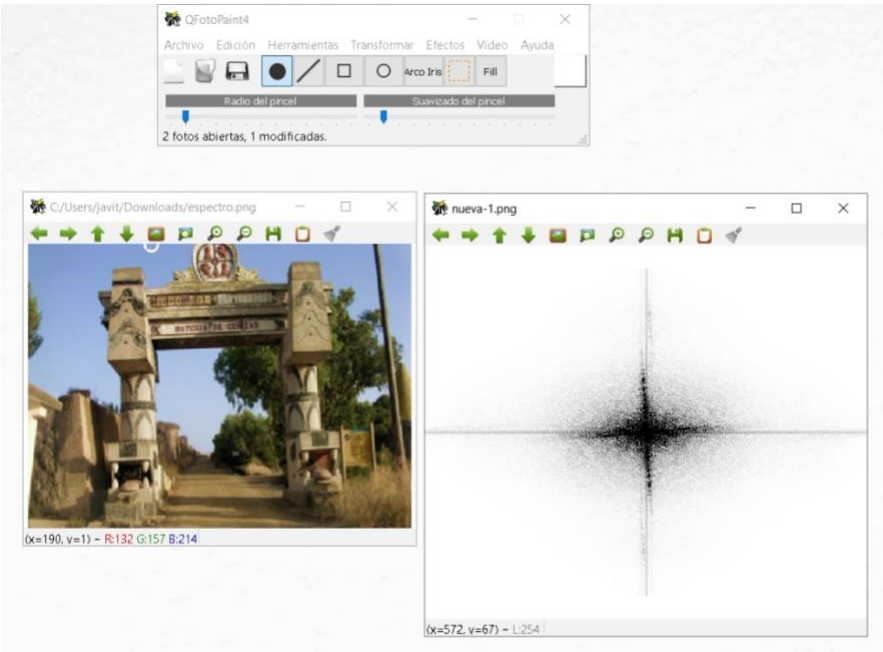

Ilustración 19. Ejemplo de uso de la funcionalidad generar espectro

5.5 Rellenar

Ilustración 20. Imagen de entrada para la funcionalidad rellenar

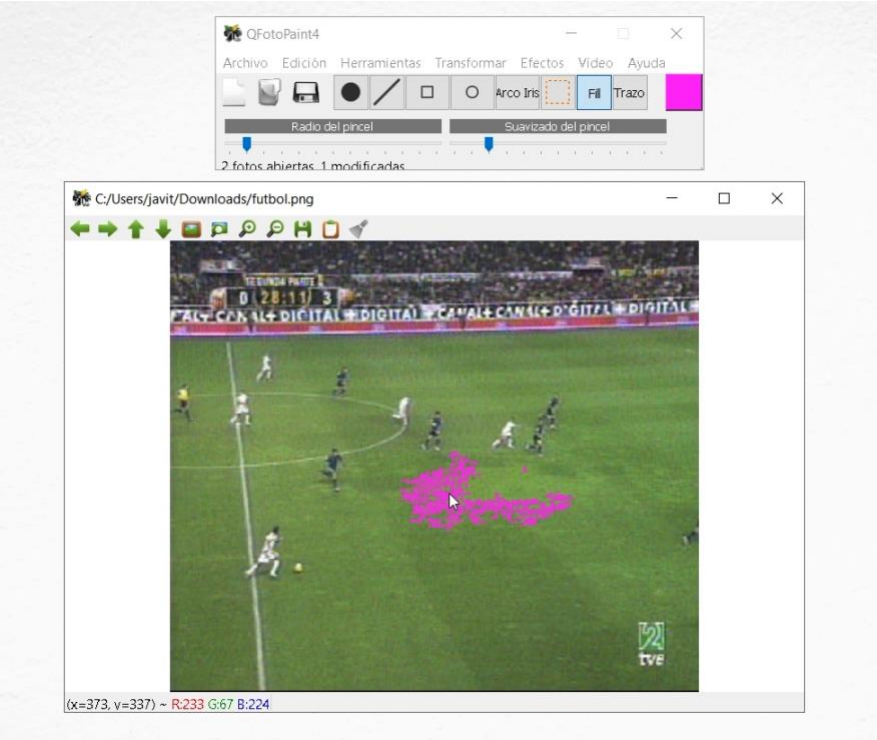

Ilustración 21. Ejemplo de uso de la funcionalidad rellenar

5.6 Trazos

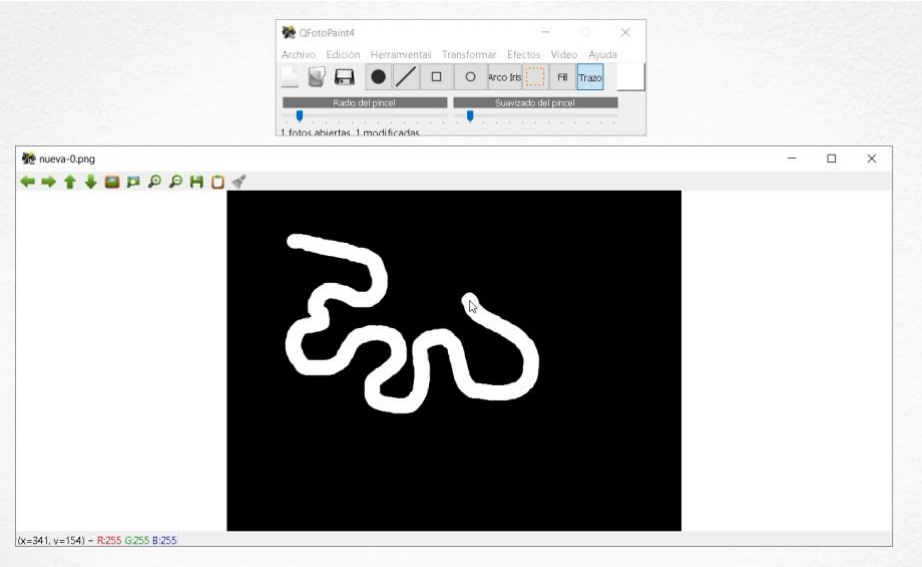

Ilustración 22. Ejemplo de uso de la funcionalidad trazo

6 Conclusiones

La combinación de *OpenCV* y *C++* ha demostrado ser muy potente, ya que, como se ha podido ver a lo largo del trabajo desarrollado, su uso ha dado lugar a funcionalidades muy interesantes. En cuanto a la satisfacción del trabajo, ha sido muy alta, ya que hemos visto como conceptos teóricos se aplican a casos reales, algo que es de una gran importancia. Sin embargo, cabe destacar que el poder desarrollar funcionalidades de forma autónoma al principio nos abrumó, pero tras desarrollar un par, entramos en velocidad de crucero. Esto último debido a que interiorizamos y mejoramos la metodología para desarrollar funcionalidades, que básicamente era, primero buscar información en teoría sobre la funcionalidad a desarrollar, para así comprender a fondo todo lo relacionado con ella; a continuación buscar en la documentación oficial de *OpenCV* si hay alguna clase o función que nos cubra o apoye el desarrollo la funcionalidad; entonces ver ejemplos de uso para comprender mejor aún cómo funciona lo que necesitamos; después implementar la funcionalidad en sí; por último realizar pruebas con imágenes de las diapositivas de teoría. Cabe mencionar que el ámbito de la práctica y de la asignatura ha sido de gran interés, debido a que ahora tenemos una mejor comprensión y acercamiento al procesamiento de imágenes, algo que nos parece de vital importancia en el mundo actual y para nuestro futuro laboral. En resumen, la realización de la práctica ha sido muy agradable y fue más sencilla de lo que pensamos inicialmente.