Visual Studio Code

UNIVERSIDAD NACIONAL DE LA PLATA

QRGB++: App sustentable, solidaria e innovadora para aumentar la densidad de información acumulada en un Código QR por el método 3-en-1 del sistema de colores RGB, utilizando librerías de código abierto en Python en Visual Studio Code.

Author: Ph.D., Mg., DI Ibar F. Anderson.
Email: ibaranderson@empleados.fba.unlp.edu.ar
Google Scholar:
https://scholar.google.com/citations?user=mXD4RFUAAAAJ&hl=en

OSFPREPRINTS https://osf.io/preprints/osf/g3ame

Abstract:
El proyecto QRGB++ desarrolla una aplicación en Python que utiliza el sistema de colores RGB para generar códigos QR con mayor capacidad de almacenamiento, eficiencia en el uso de recursos y un impacto positivo en el medio ambiente. Este innovador método permite almacenar hasta tres veces más información en el mismo espacio al combinar tres códigos QR (rojo, verde y azul) en uno solo, mejorando también la seguridad de los datos mediante la redundancia. Además de optimizar el uso del papel y tinta, contribuye a la reducción de residuos, deforestación y huella de carbono, alineándose con el ecodiseño y las tendencias sostenibles. Su capacidad para extender la vida útil de productos impresos lo hace ideal para campañas informativas que requieren actualizaciones frecuentes, promoviendo un enfoque minimalista y funcional, especialmente cuando se usa con materiales reciclados. El código QRGB se genera mediante librerías de código abierto como qrcode[pil], Pillow y opencv-python, integrando una interfaz gráfica (GUI) con Kivy o Tkinter para facilitar la creación y decodificación de los códigos. Esta accesibilidad técnica promueve la inclusión tecnológica y la democratización del uso de herramientas avanzadas, al permitir que diseñadores con conocimientos básicos de programación lo utilicen gratuitamente. Visual Studio Code es el entorno elegido para el desarrollo, lo que facilita la programación y depuración del código. A pesar de algunos desafíos técnicos con la mezcla de colores, QRGB++ ofrece un aumento significativo en la densidad de datos, con potenciales aplicaciones en seguridad y almacenamiento de alta densidad. Su uso en proyectos sociales y educativos mejora la accesibilidad para personas con discapacidades, fomenta la creatividad y la solidaridad, y refuerza la sostenibilidad en todas las etapas del ciclo de vida del producto, demostrando cómo la tecnología y el diseño pueden converger para generar un impacto positivo en la sociedad.

Introducción.
describe la aplicación innovadora QRGB++, que utiliza el sistema de colores RGB para generar códigos QR con mayor capacidad de almacenamiento, eficiencia en el uso de recursos y un impacto positivo en el medio ambiente. Esta herramienta tiene múltiples conexiones con la agenda del futuro enfocada en sostenibilidad, innovación y triple impacto. Desde la perspectiva de la sustentabilidad y el reciclaje, QRGB++ reduce la necesidad de imprimir múltiples códigos QR, disminuyendo el consumo de papel y tinta, lo que reduce la generación de residuos y el uso de recursos naturales. Esto se alinea con la prioridad del cuidado del medio ambiente y el reciclaje mencionada en la frase. Además, el enfoque innovador de esta aplicación combina tecnología, sustentabilidad y ecodiseño, lo que demuestra un impacto positivo en la sociedad. Al aplicar el método 3-en-1 mediante el uso de capas de color (rojo, verde y azul), se incrementa la densidad

de información acumulada, permitiendo almacenar tres códigos QR en uno solo. Esta optimización no solo contribuye a la sostenibilidad ecológica al reducir el uso de papel, sino que también está en sintonía con el ecodiseño, una corriente avanzada de los países desarrollados, que busca evitar el consumo excesivo de recursos.

Por otro lado, la innovación tecnológica de QRGB++ radica en su implementación mediante Python, utilizando librerías de código abierto como Kivy, Pillow, qrcode[pil] y opencv-python, lo que permite su acceso gratuito a cualquier diseñador con conocimientos de programación. Este aspecto refuerza el impacto positivo en la sociedad, ya que promueve la inclusión tecnológica y la democratización del acceso a herramientas avanzadas de generación y manipulación de códigos QR. El script corre en Visual Studio Code, lo que facilita su uso mediante una interfaz gráfica intuitiva. En resumen, QRGB++ no solo representa una innovación técnica en la densificación de la información, sino que también refuerza la sostenibilidad al reducir el consumo de recursos, fomenta la creatividad y se alinea con las corrientes de diseño más evolucionadas, integrando tecnología, ecología y responsabilidad social.

El presente trabajo, titulado QRGB++, consiste en el desarrollo de una aplicación en Python para la generación de códigos QR mediante la técnica de generación aditiva de colores (RGB). Este método innovador utiliza tres capas de colores—rojo, verde y azul—cada una representando un conjunto de datos distinto, lo que permite almacenar hasta tres veces más información en el mismo espacio en comparación con los códigos QR tradicionales en blanco y negro. Al combinar tres códigos QR en uno solo mediante colores, se incrementa la capacidad de almacenamiento y se mejora la seguridad de la información, dificultando la falsificación y manipulación, y permitiendo la implementación de redundancia que aumenta la robustez frente a daños o errores de lectura.

El uso de códigos QRGB también tiene un impacto positivo en la sostenibilidad y el ecodiseño, al reducir la necesidad de imprimir múltiples códigos, disminuyendo el uso de papel y tinta. Esto contribuye a la reducción de residuos, el ahorro de recursos naturales, y la disminución de la demanda de papel, lo que a su vez reduce la deforestación y el consumo de agua y energía en la producción de papel, resultando en una menor huella de carbono y un impacto reducido en los ecosistemas. Además, al reducir el volumen de papel transportado, se disminuye la emisión de gases contaminantes durante el transporte. Los códigos QRGB optimizan el uso del espacio en los diseños impresos, permitiendo la creación de materiales más compactos y eficientes, promoviendo un enfoque minimalista y funcional, especialmente cuando se combinan con papeles reciclados o sostenibles. Su capacidad para extender la vida útil de un producto impreso, al no necesitar ser reemplazado con frecuencia, es relevante en campañas informativas que se actualizan constantemente. Integrar QRGB en campañas de sostenibilidad educa sobre la importancia de reducir el consumo de papel y otros recursos, destacando el papel de la tecnología y el diseño en la protección del medio ambiente.

Desde el punto de vista de la programación, el script en Python para la generación de códigos QRGB utiliza varias bibliotecas clave: qrcode[pil] para la generación de códigos QR, Pillow para la manipulación de imágenes, y opencv-python para el procesamiento avanzado de imágenes. La implementación incluye una interfaz gráfica de usuario (GUI) creada con Tkinter, que permite al usuario generar y decodificar códigos QR con logotipos superpuestos, así como combinar imágenes QR de diferentes colores. El segundo script está diseñado para ser más modular y extensible, incorporando bibliotecas clave como Kivy para la interfaz gráfica, Pillow para la manipulación de imágenes, qrcode[pil] para la generación de códigos QR, y opencv-python para el procesamiento avanzado de imágenes. Para instalar las bibliotecas necesarias, se deben utilizar los siguientes comandos: pip install kivy, pip install pillow, pip install qrcode[pil], y pip install opencv-python. El script facilita la selección de archivos mediante FileChooserIconView y la apertura de enlaces en el navegador, lo que aumenta la flexibilidad y la funcionalidad de la aplicación.
Además, el desarrollo y la ejecución del código se realiza utilizando Visual Studio Code, un entorno de desarrollo integrado (IDE) que facilita la escritura, depuración y ejecución del código Python de manera eficiente. Este IDE ofrece herramientas integradas y extensiones que permiten una programación más efectiva y un manejo adecuado de las bibliotecas utilizadas en el proyecto.
A pesar de enfrentar problemas con la mezcla de colores y la recuperación precisa de la información, el método propuesto demuestra un aumento significativo en la densidad de datos dentro de un solo código QR. El trabajo futuro se centrará en optimizar el algoritmo y explorar aplicaciones potenciales en seguridad de datos y almacenamiento de información de alta densidad.
Los códigos QRGB no solo representan una innovación tecnológica, sino que también tienen un impacto significativo en el diseño ecológico y la accesibilidad. Su implementación en proyectos de diseño social puede mejorar la accesibilidad al proporcionar información de manera visualmente atractiva para personas con discapacidades, facilitar la educación en comunidades vulnerables y fomentar la creatividad y solidaridad en proyectos artísticos o campañas de sostenibilidad. Al relacionar QRGB con creatividad solidaria, se demuestra cómo la tecnología y el diseño convergen para crear un impacto social positivo, alineándose con los principios del ecodiseño y maximizando la sostenibilidad a lo largo del ciclo de vida del producto o servicio.

Keywords: QR Code, QRGB++, RGB Color, Python, Script, Software, Visual Studio Code.

Componentes del Código QR.
El código QR, abreviatura de "Quick Response", es un tipo de código de barras bidimensional que fue desarrollado en 1994 por la empresa japonesa Denso Wave. Su propósito inicial era rastrear piezas en la industria automotriz, pero su capacidad para almacenar grandes cantidades de datos y su facilidad de escaneo

han llevado a su adopción en una variedad de sectores, desde la publicidad hasta la gestión de inventarios. Los códigos QR se han convertido en una herramienta poderosa para la codificación y transferencia rápida de información.

Los códigos QR están compuestos por varios elementos clave que permiten su funcionamiento efectivo. Los Finder Patterns son tres grandes cuadrados ubicados en las esquinas del código QR que ayudan a los escáneres a identificar y orientar el código correctamente. Estos patrones proporcionan una referencia para alinear el código durante el escaneo. Los Alignment Patterns son pequeños cuadrados adicionales que se encuentran dentro del código QR y que ayudan a ajustar el escaneo en caso de que el código esté inclinado o distorsionado, asegurando que la información se pueda leer correctamente incluso si el código está en un ángulo. Los Timing Patterns son líneas en zigzag que se extienden entre los Finder Patterns y que determinan la anchura de los módulos, facilitando la interpretación precisa de los datos almacenados.

El Data Area es la sección del código QR donde se almacenan los datos codificados. Esta área puede variar en tamaño dependiendo de la capacidad de almacenamiento requerida. Adicionalmente, el Error Correction es un componente crucial que incluye secciones de información adicional destinadas a recuperar datos si el código está dañado. Este mecanismo permite que el código QR siga siendo legible a pesar de daños parciales, garantizando así una mayor fiabilidad en la transferencia de datos.

En resumen, los códigos QR son una herramienta versátil y eficiente para la transferencia de información, gracias a su capacidad para almacenar una gran cantidad de datos y su facilidad de escaneo. Su diseño estructurado, que incluye patrones de localización, alineación, temporización y corrección de errores, asegura una alta precisión en la lectura y recuperación de datos, lo que los hace útiles en una amplia gama de aplicaciones y sectores.

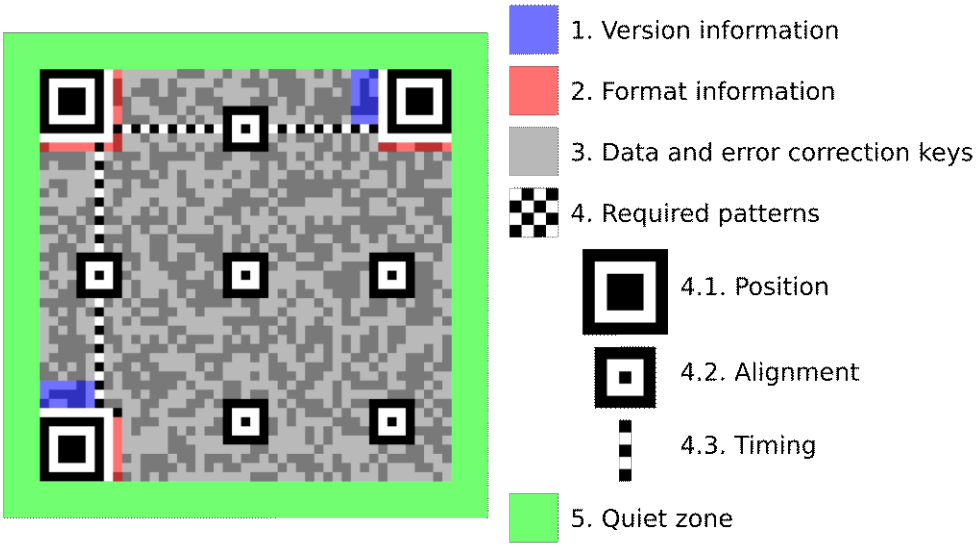

Figura 1. Código QR como convencionalmente se lo conoce. Fuente: https://fr.wikipedia.org/wiki/Code_QR#/media/Fichier:QR_Code_Structure_Example_3.svg/2

Materiales y Metodología

En el desarrollo de este proyecto, se utilizaron diversas bibliotecas de Python para implementar la generación y decodificación de códigos QRGB. A continuación, se enumeran las bibliotecas empleadas y su respectiva utilidad dentro del script:

- *Kivy:* Utilizada para crear la interfaz gráfica de usuario (GUI). Kivy es una biblioteca de código abierto que permite la creación de interfaces interactivas y multiplataforma. En este caso, se utilizó para diseñar y gestionar la ventana principal de la aplicación, permitiendo a los usuarios ingresar los datos que serán codificados en las diferentes capas del código QRGB y para seleccionar los archivos a través de un explorador de archivos visual.
- *Pillow (PIL):* Es una biblioteca de Python para la manipulación de imágenes. En el script, se utiliza para combinar imágenes y superponer logotipos sobre los códigos QR generados. Además, permite el manejo de imágenes durante el proceso de decodificación, extrayendo las capas de color correspondientes (rojo, verde y azul).
- *qrcode[pil]:* Esta biblioteca es clave para la generación de los códigos QR. Permite crear los códigos QR para las capas individuales de color (roja, verde y azul) que luego son combinadas para formar el código QRGB. Incluye también opciones para ajustar los niveles de corrección de errores y otros parámetros del código QR.
- *opencv-python (cv2):* Utilizada para el procesamiento avanzado de imágenes y la decodificación de los códigos QR. Esta biblioteca permite la lectura y decodificación de los códigos QR mediante la detección de los patrones de datos en las imágenes combinadas, facilitando la extracción de la información de cada capa de color.
- *os:* Se utiliza para realizar operaciones con el sistema de archivos, como la creación y almacenamiento de los archivos generados, incluyendo los códigos QR y las imágenes decodificadas. Esta biblioteca permite también comprobar la existencia de archivos, como los logotipos a ser superpuestos, y acceder a las rutas de almacenamiento.
- *webbrowser:* Utilizada para abrir enlaces en el navegador web. Esta función es implementada en la GUI de Kivy para permitir que los usuarios abran enlaces directamente desde la aplicación, si se incluye un enlace en las capas de color del código QRGB.

En el proceso de implementación, para la generación del Código QRGB se utilizaron las bibliotecas *qrcode[pil]* y *Pillow* para generar tres códigos QR individuales, cada uno con los datos correspondientes a una de las capas de color (rojo, verde y azul). Estos códigos se combinaron luego en una sola imagen utilizando la técnica de superposición de colores.

La interfaz Gráfica (GUI) se llevó adelante con la biblioteca *Kivy* que facilitó la creación de la interfaz gráfica que permite a los usuarios interactuar con el sistema, ingresando los datos y seleccionando imágenes para el logotipo.
En la decodificación manual se emplearon las bibliotecas *Pillow y opencv-python* para analizar y decodificar manualmente el código QRGB superpuesto, extrayendo los datos almacenados en cada una de las capas de color.

Desarrollo del programa en lenguaje de programación Python.

```
import kivy
from kivy.app import App
from kivy.uix.boxlayout import BoxLayout
from kivy.uix.label import Label
from kivy.uix.button import Button
from kivy.uix.textinput import TextInput
from kivy.uix.image import Image as KivyImage
from kivy.core.window import Window
from kivy.uix.popup import Popup
from PIL import Image
import qrcode
import os
import cv2

# Ruta para guardar y cargar archivos
FOLDER_PATH = r'C:¥Users¥Usuario¥Documents¥Archivos_Fede¥Tutorial'

# Función para crear un código QR con un logo superpuesto
def create_qr_with_logo(data, color, logo_path, qr_version=10, box_size=10):
    qr = qrcode.QRCode(
        version=qr_version,
        error_correction=qrcode.constants.ERROR_CORRECT_H,
        box_size=box_size,
        border=4
    )
    qr.add_data(data)
    qr.make(fit=True)

    img = qr.make_image(fill_color=color, back_color="white").convert('RGBA')

    if not os.path.exists(logo_path):
        raise FileNotFoundError(f"Logo file not found: {logo_path}")

    logo = Image.open(logo_path).convert("RGBA")
    basewidth = img.size[0] // 4
    wpercent = (basewidth / float(logo.size[0]))
    hsize = int((float(logo.size[1]) * float(wpercent)))
    logo = logo.resize((basewidth, hsize), Image.LANCZOS)
```

```
    pos = ((img.size[0] - logo.size[0]) // 2, (img.size[1] - logo.size[1]) // 2)
    img.paste(logo, pos, logo)

    return img

# Combina las imágenes QR asegurándose de que todas tengan el mismo tamaño
def combine_qr_images(img1, img2, img3, logo_path):
    size = img1.size
    if img2.size != size or img3.size != size:
        raise ValueError("All QR images must be the same size")

    final_image = Image.new("RGBA", size, "black")

    data_red = img1.getdata()
    data_green = img2.getdata()
    data_blue = img3.getdata()

    new_data = []
    for i in range(len(data_red)):
        r1, g1, b1, a1 = data_red[i]
        red_pixel = (r1, g1, b1) != (255, 255, 255)
        r2, g2, b2, a2 = data_green[i]
        green_pixel = (r2, g2, b2) != (255, 255, 255)
        r3, g3, b3, a3 = data_blue[i]
        blue_pixel = (r3, g3, b3) != (255, 255, 255)

        if red_pixel and green_pixel and blue_pixel:
            new_data.append((255, 255, 255, 255))
        elif red_pixel and green_pixel:
            new_data.append((255, 255, 0, 255))
        elif red_pixel and blue_pixel:
            new_data.append((255, 0, 255, 255))
        elif green_pixel and blue_pixel:
            new_data.append((0, 255, 255, 255))
        elif red_pixel:
            new_data.append((255, 0, 0, 255))
        elif green_pixel:
            new_data.append((0, 255, 0, 255))
        elif blue_pixel:
            new_data.append((0, 0, 255, 255))
        else:
            new_data.append((0, 0, 0, 255))

    final_image.putdata(new_data)

    logo = Image.open(logo_path).convert("RGBA")
    basewidth = final_image.size[0] // 4
```

```
    wpercent = (basewidth / float(logo.size[0]))
    hsize = int((float(logo.size[1]) * float(wpercent)))
    logo = logo.resize((basewidth, hsize), Image.LANCZOS)

    pos = ((final_image.size[0] - logo.size[0]) // 2, (final_image.size[1] -
logo.size[1]) // 2)
    final_image.paste(logo, pos, logo)

    return final_image

# Función para generar un código QRGB
def generate_qrgb(red_data, green_data, blue_data, logo_path, mode):
    qr_version = 10 if mode == 'link' else 3
    box_size = 10 if mode == 'link' else 20

    img_red = create_qr_with_logo(red_data, "red", logo_path, qr_version, box_size)
    img_green = create_qr_with_logo(green_data, "green", logo_path, qr_version,
box_size)
    img_blue = create_qr_with_logo(blue_data, "blue", logo_path, qr_version, box_size)

    img_red.save(os.path.join(FOLDER_PATH, "qr_red.png"))
    img_green.save(os.path.join(FOLDER_PATH, "qr_green.png"))
    img_blue.save(os.path.join(FOLDER_PATH, "qr_blue.png"))

    combined_img = combine_qr_images(img_red, img_green, img_blue, logo_path)
    combined_img.save(os.path.join(FOLDER_PATH, "superposed_qr.png"))

    return combined_img

# Función para leer un código QR desde una imagen
def read_qr(filename):
    img = cv2.imread(filename)
    detector = cv2.QRCodeDetector()
    data, vertices_array, _ = detector.detectAndDecode(img)
    if vertices_array is not None:
        return data
    else:
        return None

# Función para decodificar manualmente el QR superpuesto
def manual_decode_superposed_qr(filename):
    superposed_img = Image.open(filename)
    superposed_data = superposed_img.getdata()

    size = superposed_img.size
    red_data = [(255, 255, 255, 255)] * len(superposed_data)
    green_data = [(255, 255, 255, 255)] * len(superposed_data)
    blue_data = [(255, 255, 255, 255)] * len(superposed_data)
```

```
    for i in range(len(superposed_data)):
        r, g, b, a = superposed_data[i]
        if r != 0:  # Red
            red_data[i] = (0, 0, 0, 255)
        if g != 0:  # Green
            green_data[i] = (0, 0, 0, 255)
        if b != 0:  # Blue
            blue_data[i] = (0, 0, 0, 255)

    red_img = Image.new("RGBA", size)
    green_img = Image.new("RGBA", size)
    blue_img = Image.new("RGBA", size)

    red_img.putdata(red_data)
    green_img.putdata(green_data)
    blue_img.putdata(blue_data)

    red_img.save(os.path.join(FOLDER_PATH, "decoded_red.png"))
    green_img.save(os.path.join(FOLDER_PATH, "decoded_green.png"))
    blue_img.save(os.path.join(FOLDER_PATH, "decoded_blue.png"))

    data_red = read_qr(os.path.join(FOLDER_PATH, "decoded_red.png"))
    data_green = read_qr(os.path.join(FOLDER_PATH, "decoded_green.png"))
    data_blue = read_qr(os.path.join(FOLDER_PATH, "decoded_blue.png"))

    return data_red, data_green, data_blue

# Ventana principal de Kivy
class MainMenu(BoxLayout):
    def __init__(self, **kwargs):
        super().__init__(**kwargs)
        self.orientation = 'vertical'
        self.padding = 10
        self.spacing = 10

        self.label = Label(text="Selecciona una opción:")
        self.add_widget(self.label)

        self.btn_encode = Button(text="Codificar QRGB")
        self.btn_encode.bind(on_release=self.open_encode_menu)
        self.add_widget(self.btn_encode)

        self.btn_decode = Button(text="Decodificar QRGB")
        self.btn_decode.bind(on_release=self.decode_qr)
        self.add_widget(self.btn_decode)

    def open_encode_menu(self, instance):
```

```
        self.clear_widgets()
        self.label.text = "Introduce el texto o link para cada capa de color:"

        self.red_input = TextInput(hint_text="Texto para capa roja", multiline=False)
        self.add_widget(self.red_input)

        self.green_input = TextInput(hint_text="Texto para capa verde", multiline=False)
        self.add_widget(self.green_input)

        self.blue_input = TextInput(hint_text="Texto para capa azul", multiline=False)
        self.add_widget(self.blue_input)

        self.btn_generate = Button(text="Generar QRGB")
        self.btn_generate.bind(on_release=self.generate_qrgb)
        self.add_widget(self.btn_generate)

    def generate_qrgb(self, instance):
        red_data = self.red_input.text
        green_data = self.green_input.text
        blue_data = self.blue_input
import kivy
from kivy.app import App
from kivy.uix.boxlayout import BoxLayout
from kivy.uix.label import Label
from kivy.uix.button import Button
from kivy.uix.textinput import TextInput
from kivy.uix.image import Image as KivyImage
from kivy.core.window import Window
from kivy.uix.popup import Popup
from kivy.uix.filechooser import FileChooserIconView
from PIL import Image
import qrcode
import os
import cv2
import webbrowser

# Ruta para guardar y cargar archivos
FOLDER_PATH = r'C:¥Users¥Usuario¥Documents¥Archivos_Fede¥Tutorial'

# Función para crear un código QR con un logo superpuesto
def create_qr_with_logo(data, color, logo_path, qr_version=10, box_size=10):
    qr = qrcode.QRCode(
        version=qr_version,
        error_correction=qrcode.constants.ERROR_CORRECT_H,
        box_size=box_size,
        border=4
    )
    qr.add_data(data)
```

```
    qr.make(fit=True)

    img = qr.make_image(fill_color=color, back_color="white").convert('RGBA')

    if not os.path.exists(logo_path):
        raise FileNotFoundError(f"Logo file not found: {logo_path}")

    logo = Image.open(logo_path).convert("RGBA")
    basewidth = img.size[0] // 4
    wpercent = (basewidth / float(logo.size[0]))
    hsize = int((float(logo.size[1]) * float(wpercent)))
    logo = logo.resize((basewidth, hsize), Image.LANCZOS)

    pos = ((img.size[0] - logo.size[0]) // 2, (img.size[1] - logo.size[1]) // 2)
    img.paste(logo, pos, logo)

    return img

# Combina las imágenes QR asegurándose de que todas tengan el mismo tamaño
def combine_qr_images(img1, img2, img3, logo_path):
    size = img1.size
    if img2.size != size or img3.size != size:
        raise ValueError("All QR images must be the same size")

    final_image = Image.new("RGBA", size, "black")

    data_red = img1.getdata()
    data_green = img2.getdata()
    data_blue = img3.getdata()

    new_data = []
    for i in range(len(data_red)):
        r1, g1, b1, a1 = data_red[i]
        red_pixel = (r1, g1, b1) != (255, 255, 255)
        r2, g2, b2, a2 = data_green[i]
        green_pixel = (r2, g2, b2) != (255, 255, 255)
        r3, g3, b3, a3 = data_blue[i]
        blue_pixel = (r3, g3, b3) != (255, 255, 255)

        if red_pixel and green_pixel and blue_pixel:
            new_data.append((255, 255, 255, 255))
        elif red_pixel and green_pixel:
            new_data.append((255, 255, 0, 255))
        elif red_pixel and blue_pixel:
            new_data.append((255, 0, 255, 255))
        elif green_pixel and blue_pixel:
            new_data.append((0, 255, 255, 255))
        elif red_pixel:
```

```
            new_data.append((255, 0, 0, 255))
        elif green_pixel:
            new_data.append((0, 255, 0, 255))
        elif blue_pixel:
            new_data.append((0, 0, 255, 255))
        else:
            new_data.append((0, 0, 0, 255))

    final_image.putdata(new_data)

    logo = Image.open(logo_path).convert("RGBA")
    basewidth = final_image.size[0] // 4
    wpercent = (basewidth / float(logo.size[0]))
    hsize = int((float(logo.size[1]) * float(wpercent)))
    logo = logo.resize((basewidth, hsize), Image.LANCZOS)

    pos = ((final_image.size[0] - logo.size[0]) // 2, (final_image.size[1] -
logo.size[1]) // 2)
    final_image.paste(logo, pos, logo)

    return final_image

# Función para generar un código QRGB
def generate_qrgb(red_data, green_data, blue_data, logo_path, mode):
    qr_version = 10 if mode == 'link' else 3
    box_size = 10 if mode == 'link' else 20

    img_red = create_qr_with_logo(red_data, "red", logo_path, qr_version, box_size)
    img_green = create_qr_with_logo(green_data, "green", logo_path, qr_version,
box_size)
    img_blue = create_qr_with_logo(blue_data, "blue", logo_path, qr_version, box_size)

    img_red.save(os.path.join(FOLDER_PATH, "qr_red.png"))
    img_green.save(os.path.join(FOLDER_PATH, "qr_green.png"))
    img_blue.save(os.path.join(FOLDER_PATH, "qr_blue.png"))

    combined_img = combine_qr_images(img_red, img_green, img_blue, logo_path)
    combined_img.save(os.path.join(FOLDER_PATH, "superposed_qr.png"))

    return combined_img

# Función para leer un código QR desde una imagen
def read_qr(filename):
    img = cv2.imread(filename)
    detector = cv2.QRCodeDetector()
    data, vertices_array, _ = detector.detectAndDecode(img)
    if vertices_array is not None:
        return data
```

```
    else:
        return None

# Función para decodificar manualmente el QR superpuesto
def manual_decode_superposed_qr(filename):
    superposed_img = Image.open(filename)
    superposed_data = superposed_img.getdata()

    size = superposed_img.size
    red_data = [(255, 255, 255, 255)] * len(superposed_data)
    green_data = [(255, 255, 255, 255)] * len(superposed_data)
    blue_data = [(255, 255, 255, 255)] * len(superposed_data)

    for i in range(len(superposed_data)):
        r, g, b, a = superposed_data[i]
        if r != 0:  # Red
            red_data[i] = (0, 0, 0, 255)
        if g != 0:  # Green
            green_data[i] = (0, 0, 0, 255)
        if b != 0:  # Blue
            blue_data[i] = (0, 0, 0, 255)

    red_img = Image.new("RGBA", size)
    green_img = Image.new("RGBA", size)
    blue_img = Image.new("RGBA", size)

    red_img.putdata(red_data)
    green_img.putdata(green_data)
    blue_img.putdata(blue_data)

    red_img.save(os.path.join(FOLDER_PATH, "decoded_red.png"))
    green_img.save(os.path.join(FOLDER_PATH, "decoded_green.png"))
    blue_img.save(os.path.join(FOLDER_PATH, "decoded_blue.png"))

    data_red = read_qr(os.path.join(FOLDER_PATH, "decoded_red.png"))
    data_green = read_qr(os.path.join(FOLDER_PATH, "decoded_green.png"))
    data_blue = read_qr(os.path.join(FOLDER_PATH, "decoded_blue.png"))

    return data_red, data_green, data_blue

# Ventana principal de Kivy
class MainMenu(BoxLayout):
    def __init__(self, **kwargs):
        super().__init__(**kwargs)
        self.orientation = 'vertical'
        self.padding = 10
        self.spacing = 10
```

```
        self.label = Label(text="Selecciona una opción:")
        self.add_widget(self.label)

        self.btn_encode = Button(text="Codificar QRGB")
        self.btn_encode.bind(on_release=self.open_encode_menu)
        self.add_widget(self.btn_encode)

        self.btn_decode = Button(text="Decodificar QRGB")
        self.btn_decode.bind(on_release=self.decode_qr)
        self.add_widget(self.btn_decode)

    def open_encode_menu(self, instance):
        self.clear_widgets()
        self.label.text = "Introduce el texto o link para cada capa de color:"

        self.red_input = TextInput(hint_text="Texto para capa roja", multiline=False)
        self.add_widget(self.red_input)

        self.green_input = TextInput(hint_text="Texto para capa verde", multiline=False)
        self.add_widget(self.green_input)

        self.blue_input = TextInput(hint_text="Texto para capa azul", multiline=False)
        self.add_widget(self.blue_input)

        self.btn_select_logo = Button(text="Seleccionar Logo")
        self.btn_select_logo.bind(on_release=self.select_logo)
        self.add_widget(self.btn_select_logo)

        self.btn_generate = Button(text="Generar QRGB")
        self.btn_generate.bind(on_release=self.generate_qrgb)
        self.add_widget(self.btn_generate)

    def select_logo(self, instance):
        filechooser = FileChooserIconView()
        filechooser.filters = ['*.png']
        filechooser.bind(on_submit=self.load_logo)
        self.popup = Popup(title="Selecciona el archivo logo.png", content=filechooser,
size_hint=(0.9, 0.9))
        self.popup.open()

    def load_logo(self, filechooser, selection, *args):
        if selection:
            self.logo_path = selection[0]
            self.popup.dismiss()
        else:
            self.show_popup("Error", "No se seleccionó ningún archivo.")

    def generate_qrgb(self, instance):
```

```
        red_data = self.red_input.text
        green_data = self.green_input.text
        blue_data = self.blue_input.text

        if not red_data or not green_data or not blue_data:
            self.show_popup("Error", "Por favor, ingresa texto en todas las capas de color.")
            return

        if not hasattr(self, 'logo_path'):
            self.show_popup("Error", "Por favor, selecciona un archivo de logo.")
            return

        # Determinando si es un modo link o texto para ajustar la versión del QR
        mode = 'link' if 'http' in red_data or 'http' in green_data or 'http' in blue_data else 'text'

        try:
            combined_img = generate_qrgb(red_data, green_data, blue_data, self.logo_path, mode)
            combined_img.show()  # Mostrar la imagen generada
            self.show_popup("Éxito", "El QRGB ha sido generado correctamente y guardado en la carpeta.")
        except FileNotFoundError as e:
            self.show_popup("Error", str(e))
        except Exception as e:
            self.show_popup("Error", f"Ha ocurrido un error: {str(e)}")

    def decode_qr(self, instance):
        filechooser = FileChooserIconView()
        filechooser.filters = ['*.png']
        filechooser.bind(on_submit=self.load_qrgb)
        self.popup = Popup(title="Selecciona el archivo QRGB", content=filechooser, size_hint=(0.9, 0.9))
        self.popup.open()

    def load_qrgb(self, filechooser, selection, *args):
        if selection:
            self.qrgb_path = selection[0]
            self.popup.dismiss()
            self.decode_qrgb_file()
        else:
            self.show_popup("Error", "No se seleccionó ningún archivo.")

    def decode_qrgb_file(self):
        try:
            if not hasattr(self, 'qrgb_path'):
                self.show_popup("Error", "No se ha seleccionado ningún archivo QRGB.")
```

```
                return

            data_red, data_green, data_blue = manual_decode_superposed_qr(self.qrgb_path)

            message = f"Capa Roja: {data_red}¥nCapa Verde: {data_green}¥nCapa Azul: {data_blue}"
            self.show_popup("Resultado de Decodificación", message)
        except Exception as e:
            self.show_popup("Error", f"Ha ocurrido un error al decodificar: {str(e)}")

    def show_popup(self, title, message):
        content = BoxLayout(orientation='vertical', padding=10, spacing=10)
        label = Label(text=message)
        btn_close = Button(text="Cerrar", size_hint=(1, 0.5))
        content.add_widget(label)
        content.add_widget(btn_close)

        if "http" in message:
            btn_open_link = Button(text="Abrir enlace", size_hint=(1, 0.5))
            btn_open_link.bind(on_release=self.open_link)
            content.add_widget(btn_open_link)

        popup = Popup(title=title, content=content, size_hint=(0.75, 0.5))
        btn_close.bind(on_release=popup.dismiss)
        popup.open()

    def open_link(self, instance):
        webbrowser.open(instance.text)

# Clase principal del aplicativo
class QRGBApp(App):
    def build(self):
        Window.clearcolor = (1, 1, 1, 1)  # Fondo blanco
        return MainMenu()

# Ejecutar la aplicación
if __name__ == '__main__':
    QRGBApp().run()
```

Metodología.

La metodología para la generación y decodificación de códigos QRGB se basa en el uso de capas de color RGB para aumentar la densidad de información en los códigos QR. Primero, se generan tres códigos QR independientes, uno para cada capa de color: rojo, verde y azul, donde cada código almacena un conjunto distinto de datos, lo que permite triplicar la cantidad de información almacenada. Luego, estas capas se superponen en una sola imagen utilizando la biblioteca Pillow, que

permite combinar las tres imágenes manteniendo la integridad de la información. Después, se puede superponer un logotipo en el centro del código QRGB utilizando las funciones de Pillow, sin afectar la legibilidad gracias a la corrección de errores que implementa el código QR.

La interfaz gráfica para la generación y decodificación de los códigos QRGB se crea con la biblioteca Kivy, lo que facilita la interacción del usuario mediante una GUI intuitiva que permite ingresar los datos de cada capa de color, seleccionar archivos de imágenes y ejecutar la generación o decodificación del código. Para la decodificación, se utilizan las bibliotecas Pillow y opencv-python, que permiten extraer las capas de color del código QRGB y leer los datos correspondientes en cada una. Finalmente, las imágenes generadas, así como las capas decodificadas, se guardan en directorios específicos del sistema utilizando la biblioteca os, lo que facilita el acceso a los archivos para futuros usos o consultas.

Este enfoque permite una mayor capacidad de almacenamiento de datos en los códigos QR, mejora la seguridad al utilizar capas de color, y su implementación modular con Python y bibliotecas de código abierto asegura la flexibilidad del sistema.

Resultados y discusión.
Se ilustra el proceso de funcionamiento paso a paso, corriendo el software.

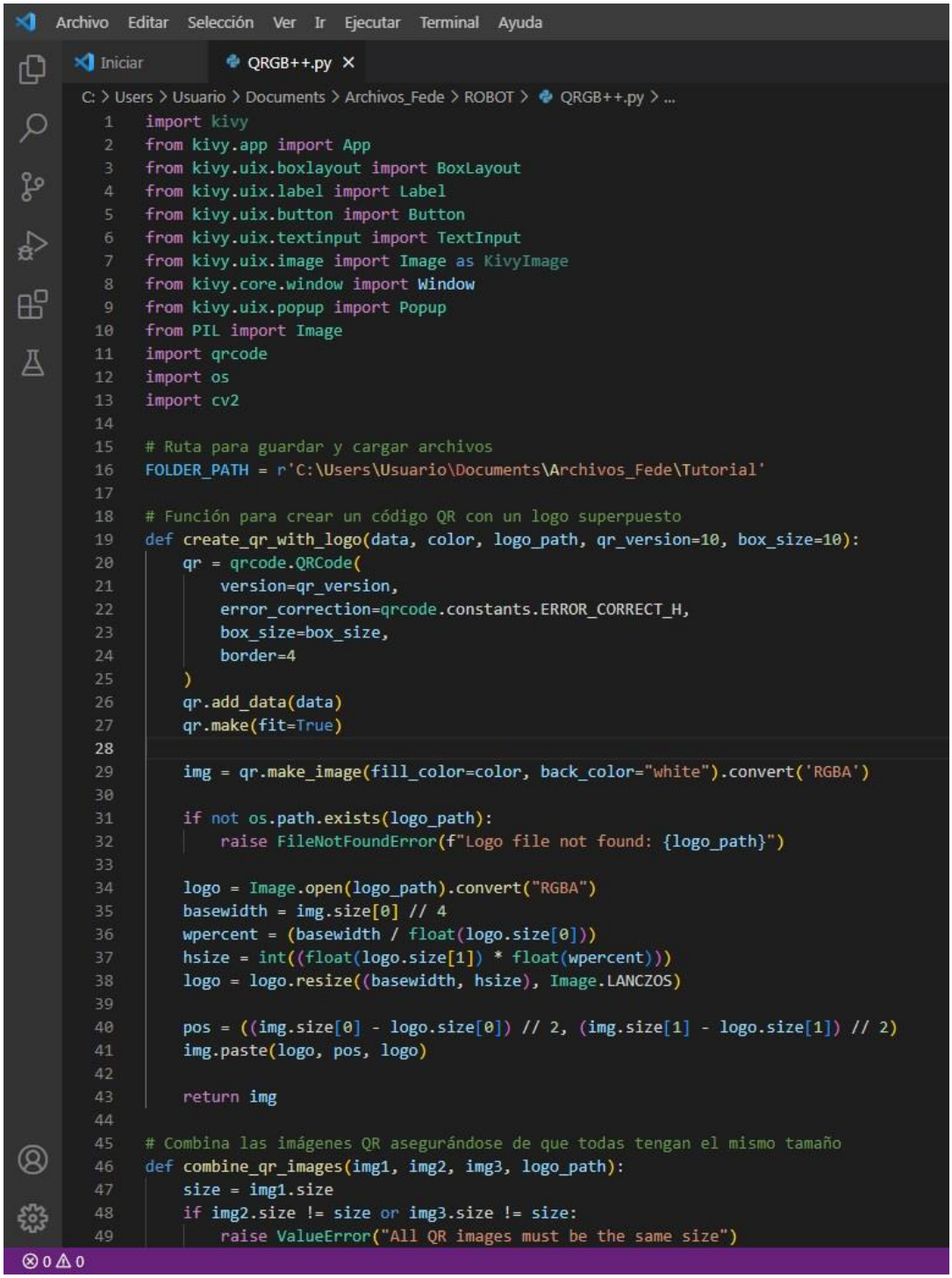

Figura 2. Script Python en Visual Studio Code. Fuente: Elaboración propia.

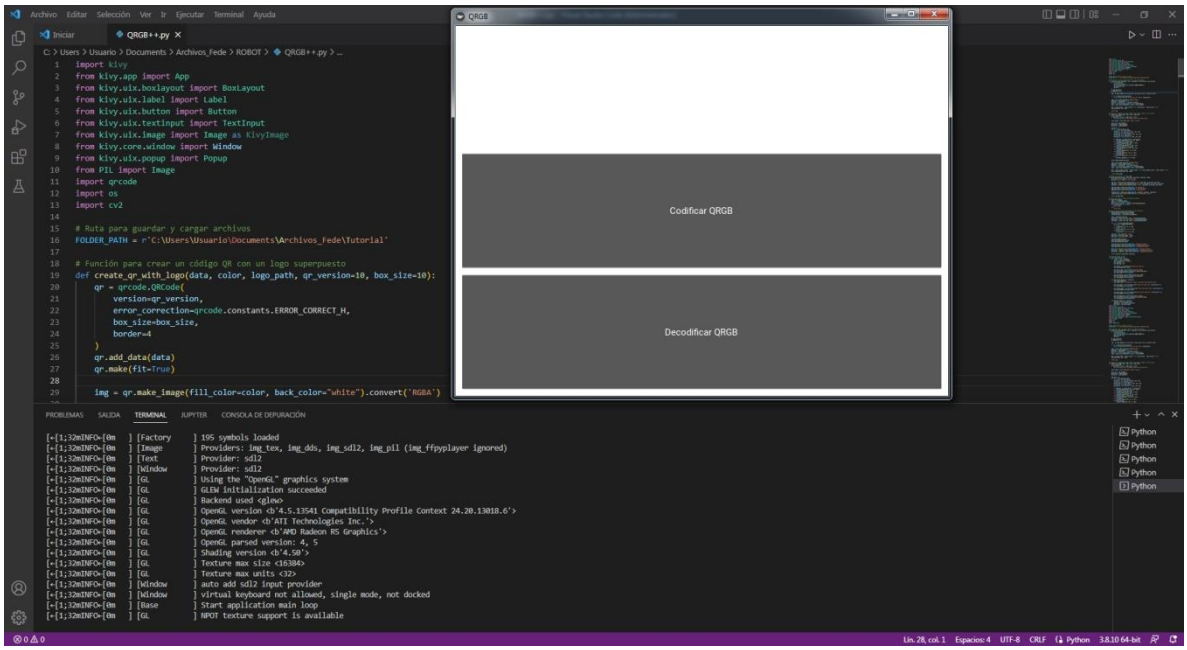

Figura 3. Ventana desarrollada con la biblioteca kivy en Python en Visual Studio Code luego de correr (run) el software. Fuente: Elaboración propia.

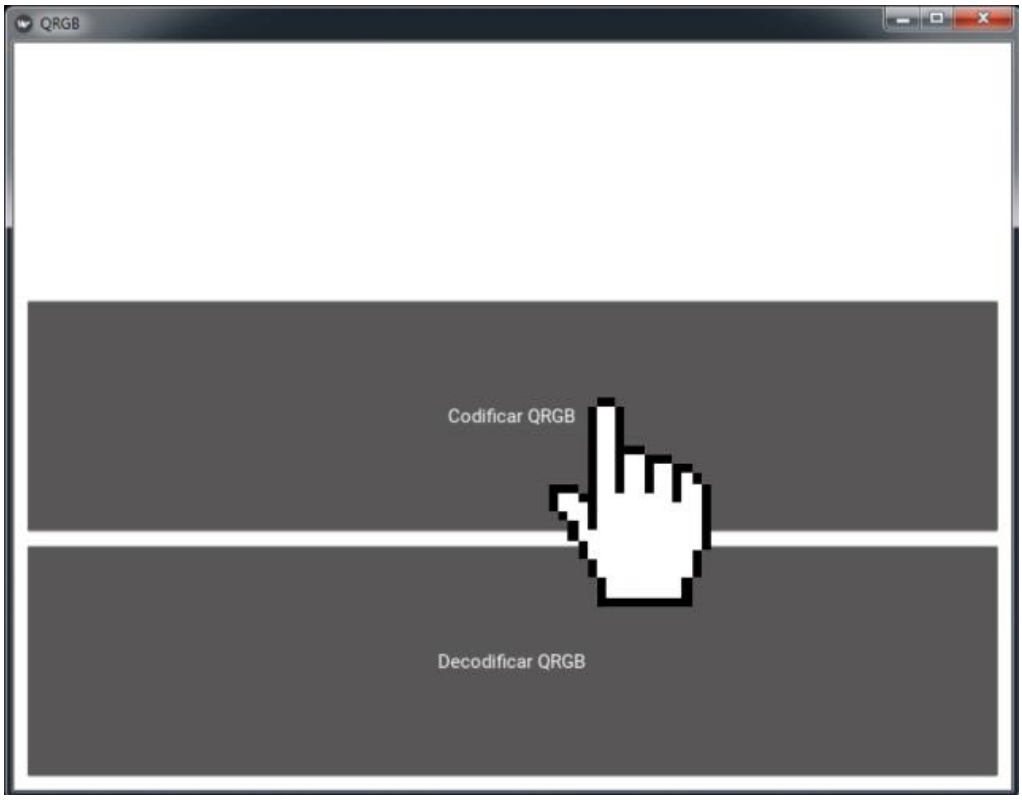

Figura 4. Se debe seleccionar la opción “Codificar QRGB”. Fuente: Elaboración propia.

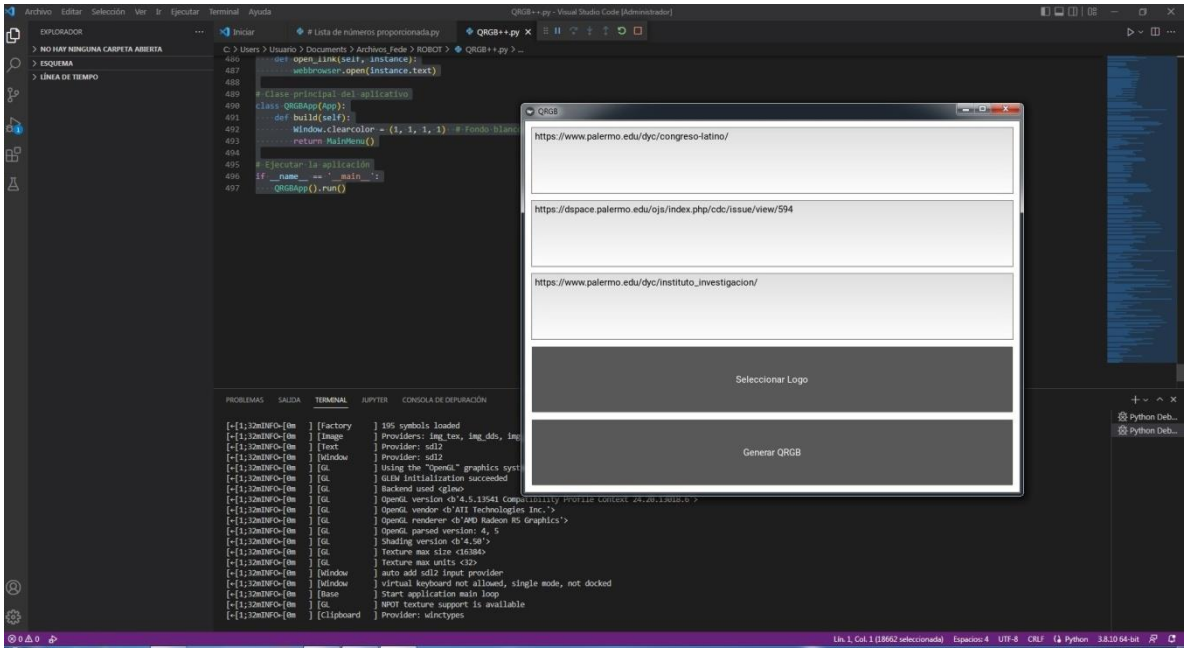

Figura 5. Se debe ingresar o cargar la información de cada canal RGB (rojo, verde y azul por separado en los espacios habilitados en la ventana que aparece para tal efecto). En este ejemplo desarrollado la información ingresada corresponde a los siguientes links respectivamente: https://www.palermo.edu/dyc/congreso-latino/, https://dspace.palermo.edu/ojs/index.php/cdc/issue/view/594, https://www.palermo.edu/dyc/instituto_investigacion/ Fuente: Elaboración propia.

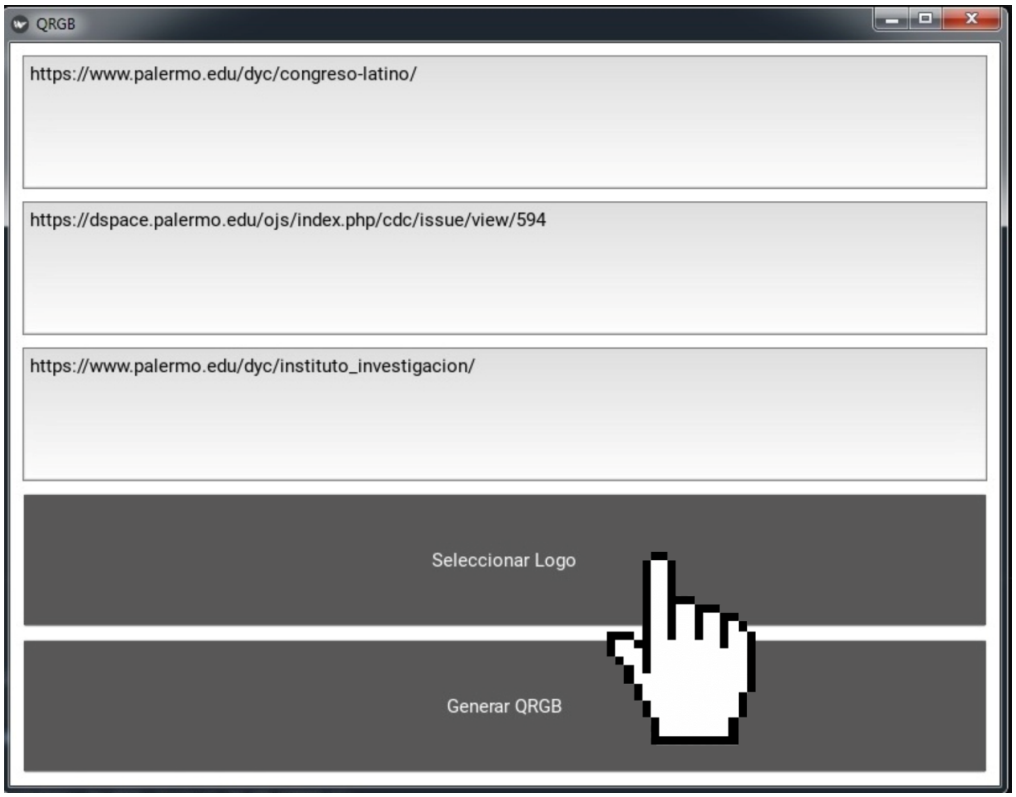

Figura 6. Luego se debe seleccionar la opción “Seleccionar Logo” para poner cargar la imagen o logo (en formato .png) para incorporar al código QRGB. Fuente: Elaboración propia.

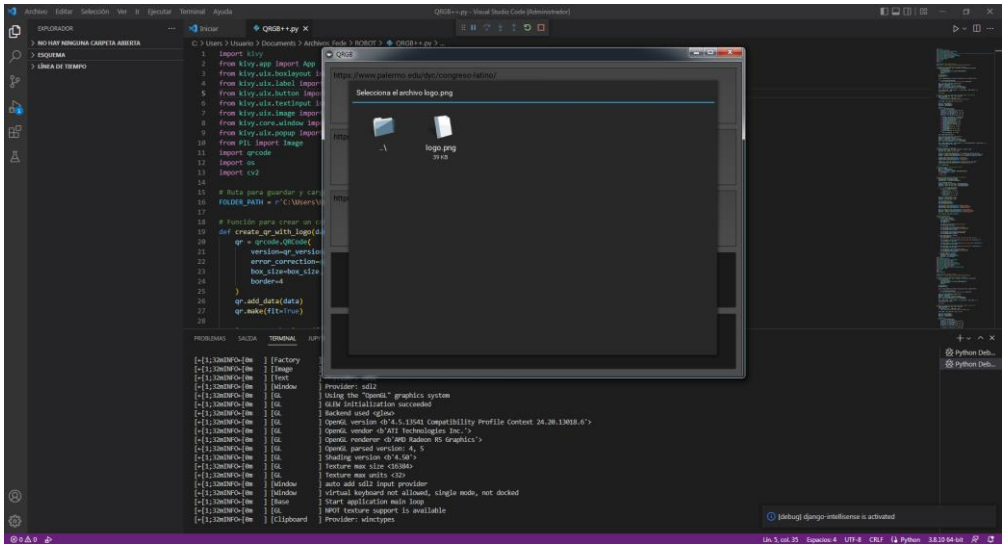

Figura 7. Se podrá ingresar a la carpeta de archivos para seleccionar la imagen Logo.png (que debe estar previamente configurada) para cargar desde una carpeta denominada y configurada como Tutorial. Fuente: Elaboración propia.

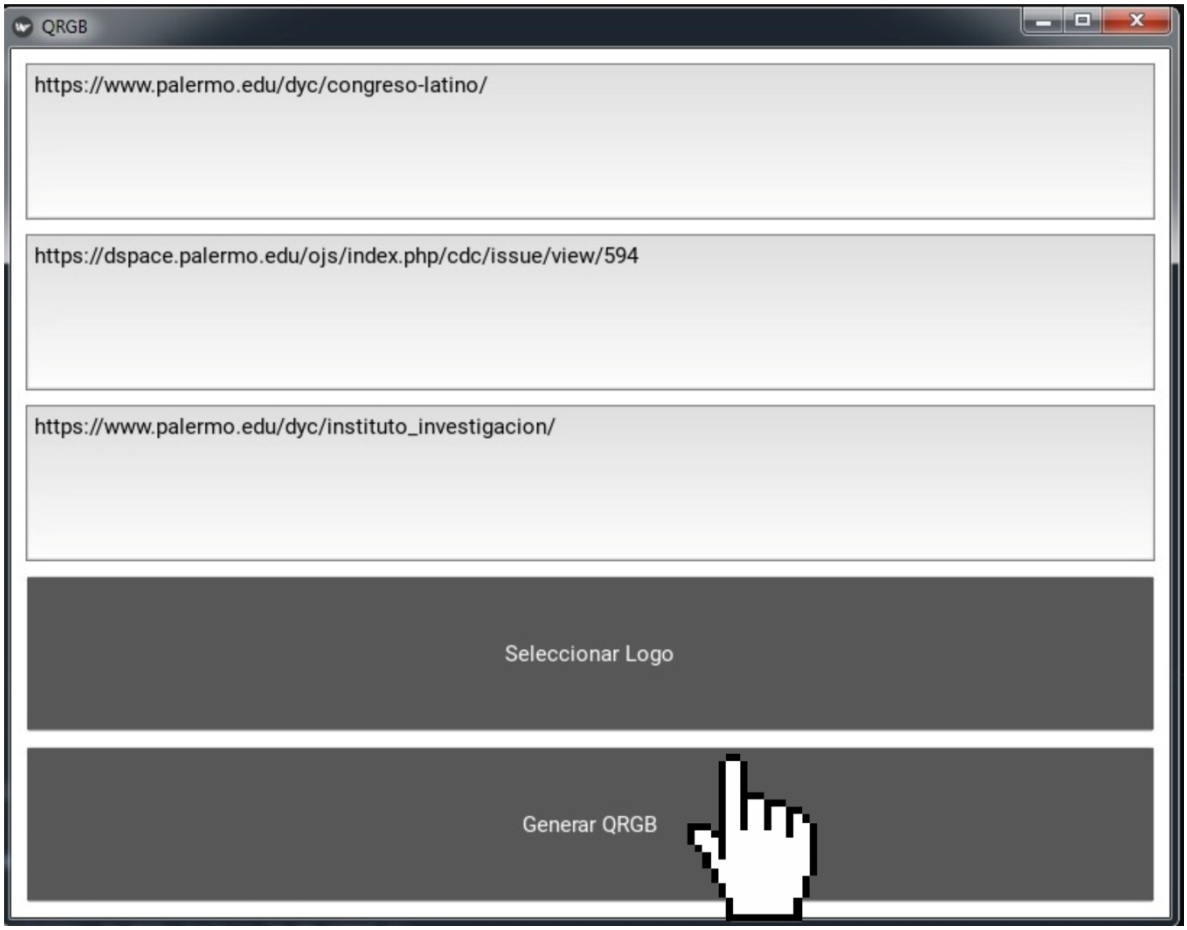

Figura 8. Luego se debe seleccionar la opción “Generar QRGB” para que se genere el código QRGB. Fuente: Elaboración propia.

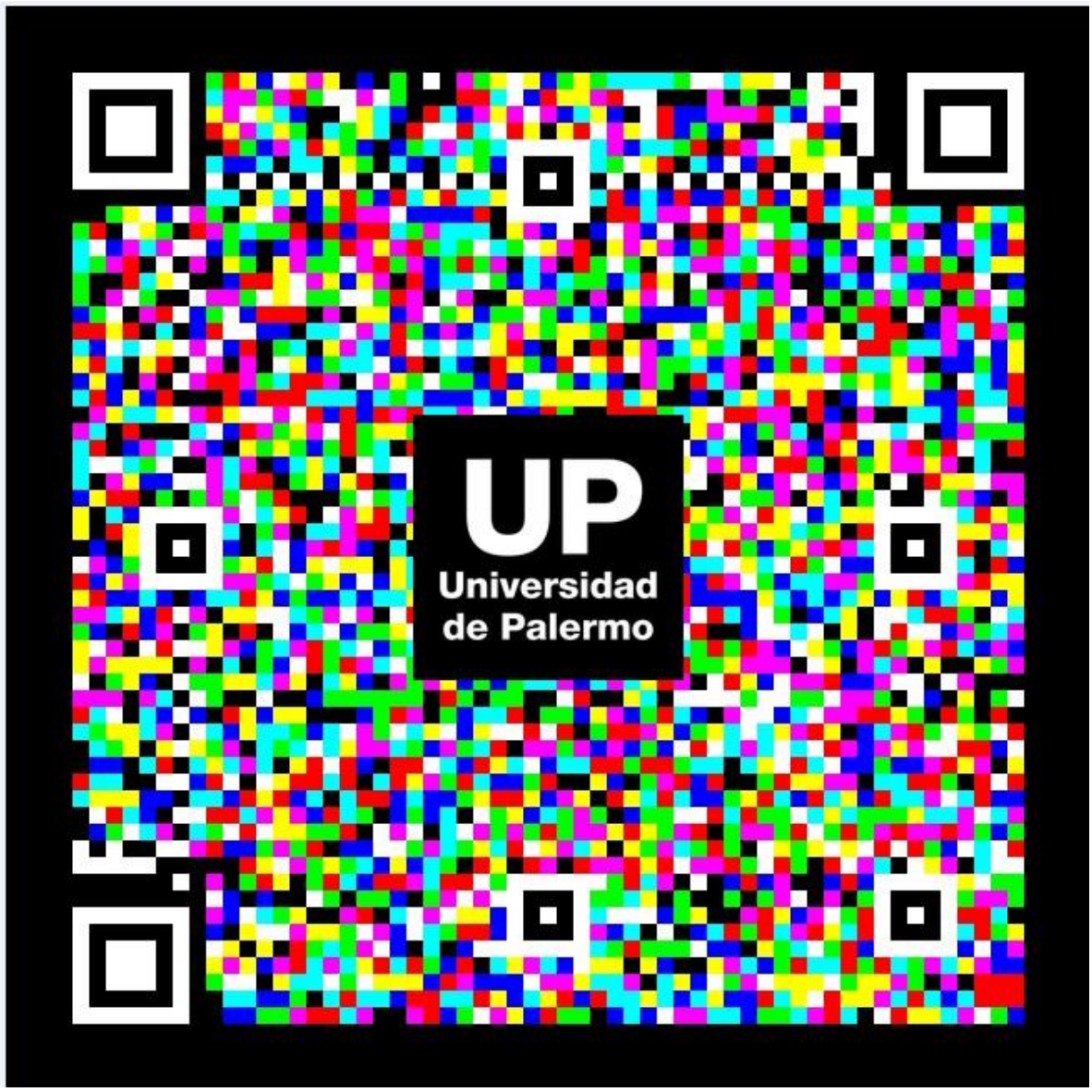

Figura 9. Se generará un código QRGB como el que muestra la imagen. Fuente: Elaboración propia.

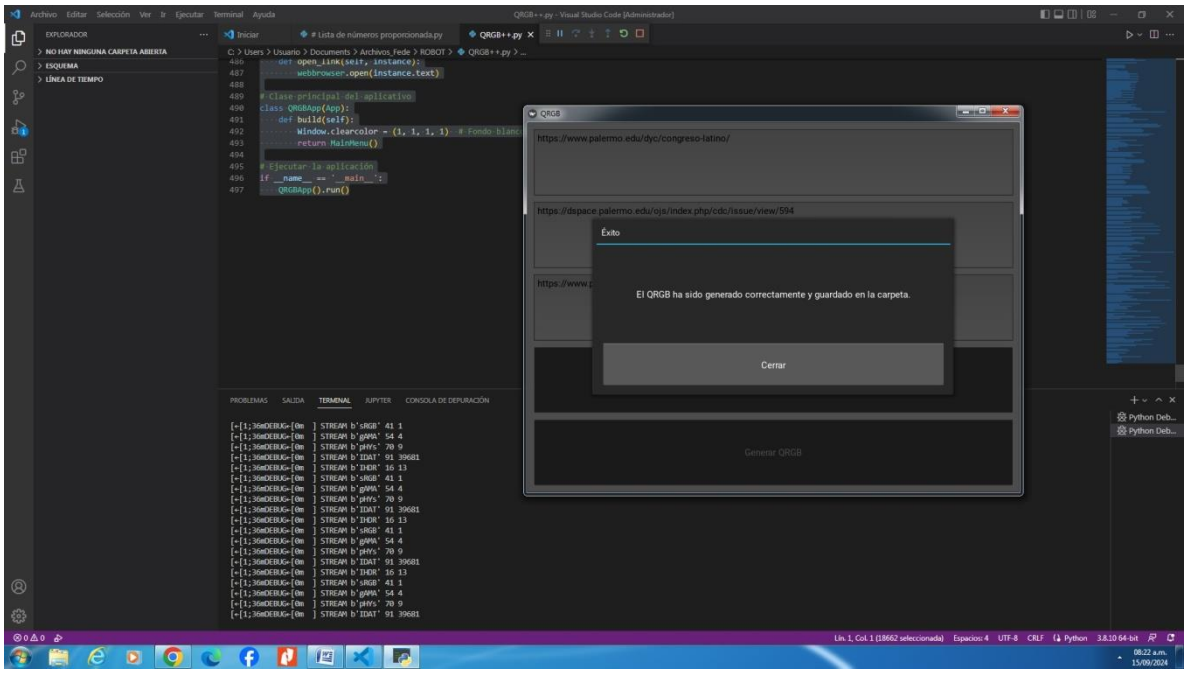

Figura 10. Luego de haberse generado el código QRGB y almacenado como imagen superposed_qr en la carpeta Tutorial, aparece un cartel en una ventana avisándonos que la operación se ha realizado exitosamente. Fuente: Elaboración propia.

Figura 11. Al ingresar a la carpeta Tutorial nos encontraremos con cinco (5) archivos: Logo,png y los tres (3) archivos qr rojo verde y azul correspondientes a los pasos intermedios de la codificación RGB (llamados respectivamente qr_blue, qr_green y qr_red) y junto a ellos el archivo superposed_qr generado. Todos archivos de imágenes de formato .png. Fuente: Elaboración propia.

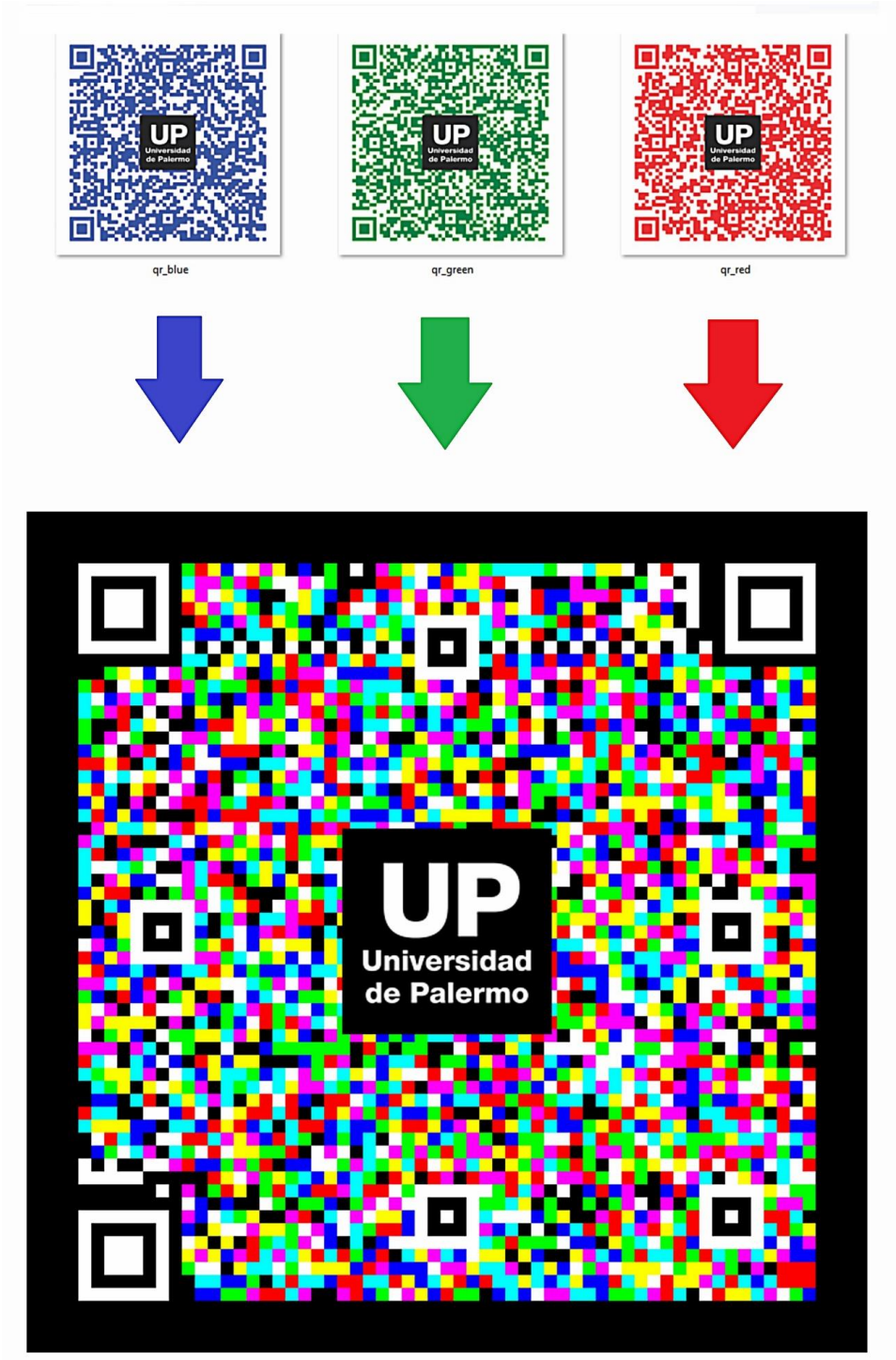

Figura 12. Los tres (3) archivos qr rojo verde y azul correspondientes a los pasos intermedios de la codificación RGB (llamados respectivamente qr_blue, qr_green y qr_red) son las etapas previas para generar la superposición del pixelado en el archivo superposed_qr. Todos archivos de imágenes de formato .png. Fuente: Elaboración propia.

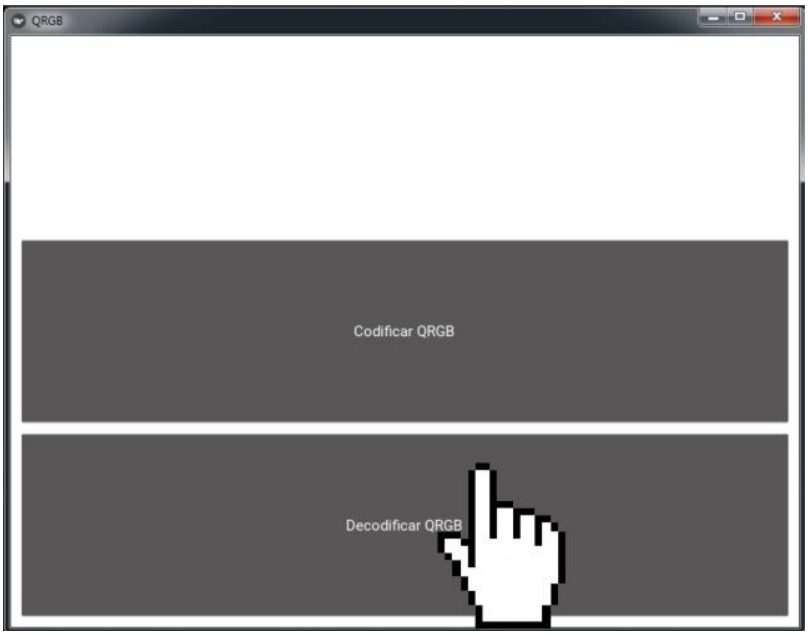

Figura 13. Si ahora seleccionamos la opción “Decodificar QRGB”. Fuente: Elaboración propia.

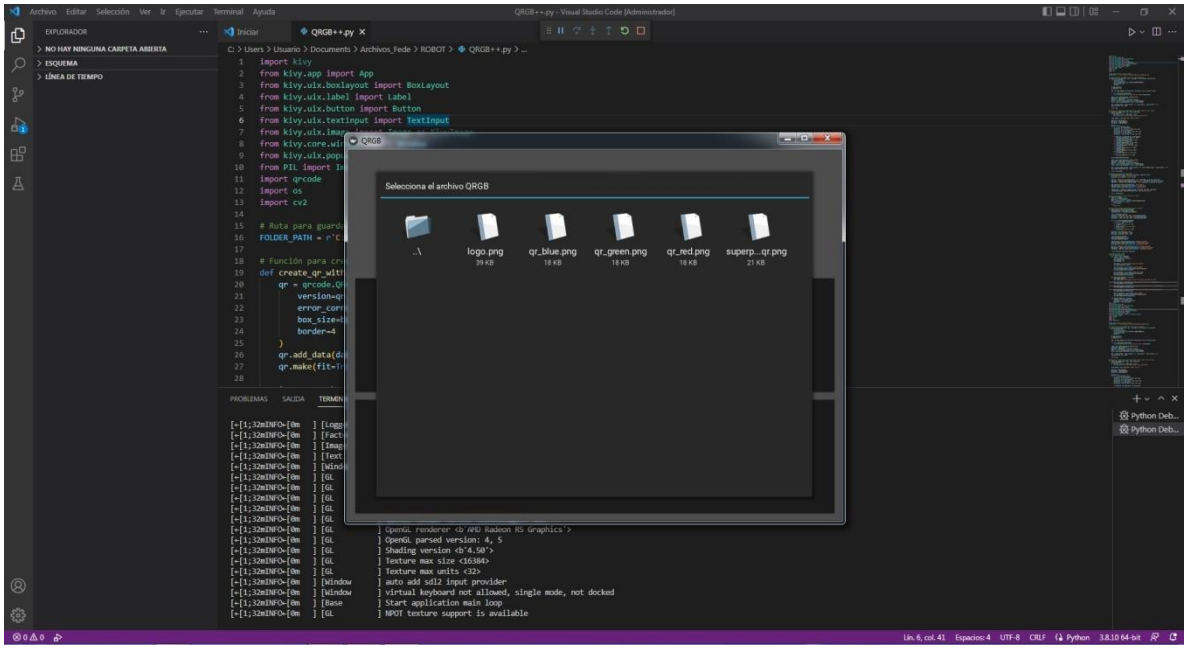

Figura 14. En la carpeta Tutorial deberemos seleccionar el archivo superposed_qr para que se produzca el proceso de decodificación. Fuente: Elaboración propia.

Figura 15. Por lo que el sistema generará tres (3) archivos intermedios (blancos y negros correspondientes al cada uno de los tres colores RGB: rojo, verde y azul) extraídos del archivo superposed_qr para proceder a su decodificación. Fuente: Elaboración propia.

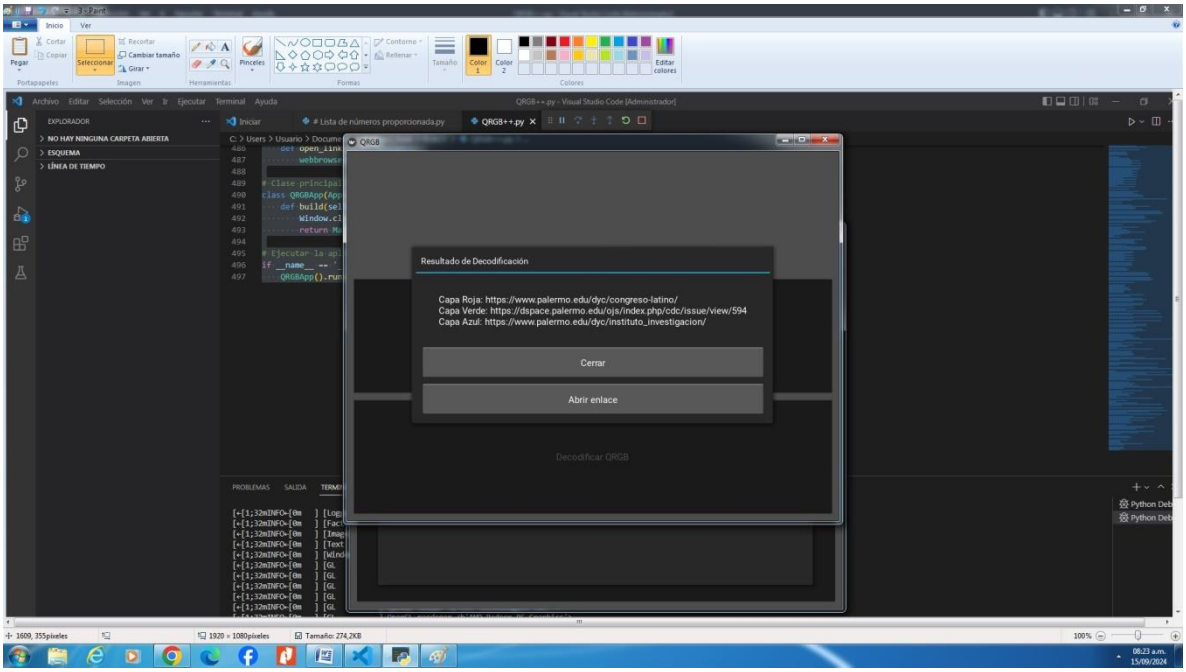

Figura 16. Así es como finalmente se produce la lectura o decodificado de los tres (3) canales rojo, verde y azul con la información respectiva: https://www.palermo.edu/dyc/congreso-latino/, https://dspace.palermo.edu/ojs/index.php/cdc/issue/view/594, https://www.palermo.edu/dyc/instituto_investigacion/ Fuente: Elaboración propia.

Conclusiones.

El concepto de códigos QR en color (QRGB), que combina tres códigos QR en rojo, verde y azul para incrementar la densidad de información, representa una innovación significativa en el ámbito de la codificación y transmisión de datos. A diferencia de los códigos QR tradicionales, los QRGB utilizan la superposición de colores para almacenar hasta tres veces más información en el mismo espacio físico, mejorando así tanto la capacidad de almacenamiento como la seguridad de los datos.

El sistema High Capacity Colored Two-Dimensional Codes (HCC2D) y el Microsoft Tag son desarrollos similares que también emplean colores para aumentar la capacidad de los códigos QR. Sin embargo, estos sistemas no abarcan la superposición de múltiples códigos QR como en el enfoque QRGB. Mientras que el HCC2D se centra en la codificación de información en cada punto de un solo código, el QRGB se destaca por su técnica única de combinar códigos en capas de color.

A pesar de su novedad, el concepto de QRGB no ha sido ampliamente adoptado ni comercializado. La implementación de algoritmos avanzados para la codificación y decodificación de estos códigos es crucial, abordando desafíos como la mezcla de colores y la precisión en la recuperación de datos. La compatibilidad con dispositivos de escaneo actuales y la adaptación de los escáneres para leer códigos QRGB son áreas que requieren atención adicional.

El uso de herramientas de Python, como qrcode[pil], Pillow, y opencv-python, para la creación de QRGB demuestra una adecuada integración de tecnologías de código abierto. No obstante, la creación de una interfaz de usuario intuitiva y la evaluación de compatibilidad con escáneres en diversas condiciones son aspectos a desarrollar para facilitar su adopción.

Los QRGB tienen aplicaciones potenciales en sectores como la seguridad de documentos, tarjetas de visita digitales, y publicidad interactiva. Su capacidad para almacenar información adicional y su mejora en la seguridad pueden revolucionar el uso de códigos QR, brindando soluciones innovadoras para el almacenamiento y la transmisión de datos en la era digital.

En futuros estudios, será esencial profundizar en la precisión de la decodificación, la compatibilidad con dispositivos existentes, y la creación de una interfaz de usuario amigable. El potencial para aplicar QRGB en criptografía visual, marketing, y educación también merece exploración adicional. Este enfoque innovador, aunque aún en desarrollo, ofrece una base sólida para el avance en la tecnología de códigos QR y su implementación práctica.

Bibliografía.

-Anderson, I. F. (2024). “QRGB: App for QR Code Generation (3-in-1 Method), Additive Color Generation Method (RGB), Using Python Programming Code, to Increase Accumulated Information Density”. *Preprints.org,* pp. 1-47. https://doi.org/10.20944/preprints202407.1384.v2

Anderson, I. F. (2024). “QRGB: App para la generación de códigos QR (método: 3 en 1), o método de generación aditiva de colores (RGB), aplicando librerías de Python de código abierto, para aumentar la densidad de información acumulada”. *EdArXiv Preprints*, pp. 1-29. https://doi.org/10.35542/osf.io/hy2em

Anderson, I. F. (2024). “QRGB+: Advanced QR Code Generator with RGB Color Method in Python to Expand Data Capacity”. *Journal of Sensor Networks and Data Communications,* Vol. 4, Nº. 2, pp. 1-20. Handle: http://sedici.unlp.edu.ar/handle/10915/169498

Anderson, I. F. (2024). “QRGB++ in Python running in Visual Studio Code with a graphical in-terface (pip install kivy) + pip install pillow + pip install qrcode[pil] + pip install opencv-python”. *OSF Preprints*, pp. 1-17. https://doi.org/10.31219/osf.io/g3ame