Documentación Técnica - Image Decomposer

Índice

Visión General

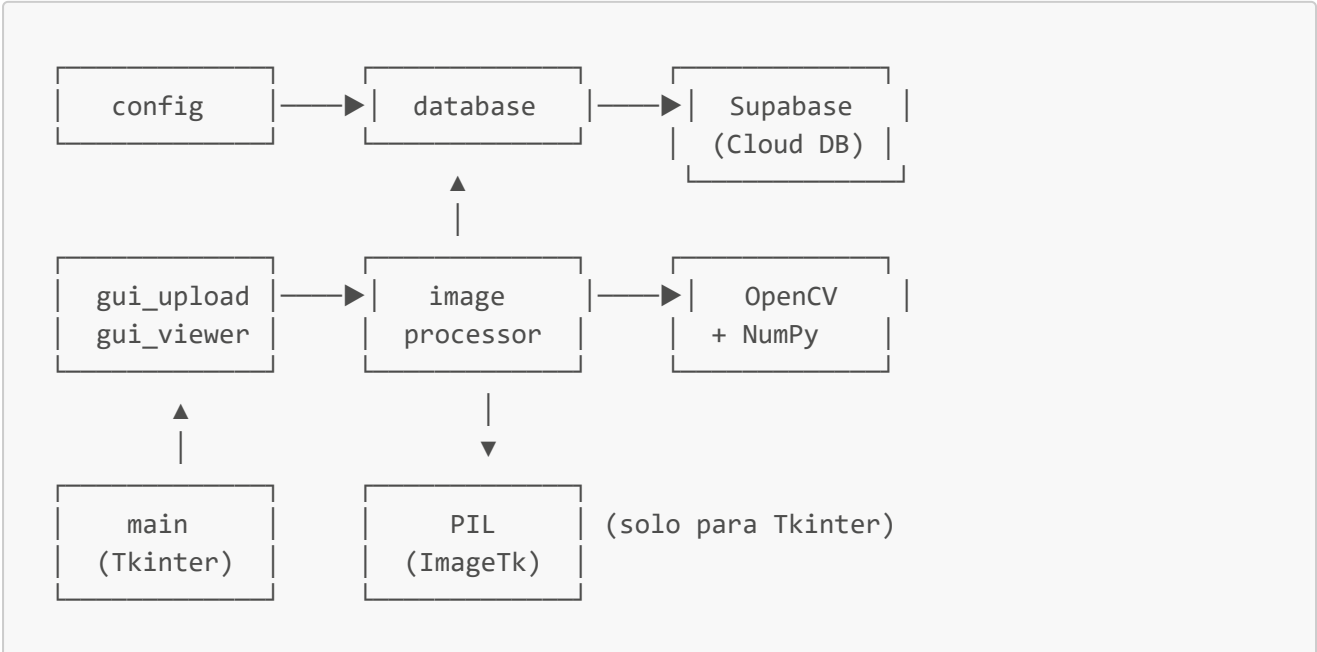
Image Decomposer es una aplicación de escritorio que permite:

- Capturar imágenes desde la cámara web con OpenCV
- Descomponer imágenes en sus valores RGB individuales
- Almacenar estos valores en una base de datos en la nube (Supabase)
- Reconstruir imágenes desde sus valores RGB almacenados

El proyecto utiliza **OpenCV (cv2)** como biblioteca principal de procesamiento de imágenes, siguiendo las técnicas enseñadas en el curso de procesamiento digital de imágenes.

Separación de Responsabilidades

El proyecto sigue el patrón de **separación de responsabilidades** y arquitectura modular:



Flujo de Datos

Captura de Imagen:

```
Cámara → cv2.VideoCapture(0) → Frame BGR → cvtColor(BGR→RGB) → Array NumPy RGB
→ Flatten → String → Supabase
```

Reconstrucción:

```
Supabase → String → Array NumPy → Reshape → Imagen RGB → PIL → Tkinter
```

Tecnologías y Arquitectura

Stack Tecnológico

Tecnología	Propósito	Dónde se Usa
OpenCV (cv2)	Captura de cámara, procesamiento y resize de imágenes	`image_processor.py`, `gui_upload.py`
NumPy	Manipulación de matrices y arrays	`image_processor.py` - flatten, reshape, zeros_like
PIL/Pillow	Conversión de NumPy a formato Tkinter **únicamente**	`image_processor.py` - Solo en `mostrar_en_canvas()`
Tkinter	Interfaz gráfica de usuario	`main.py`, `gui_upload.py`, `gui_viewer.py`
Supabase	Base de datos PostgreSQL en la nube	`database.py`
python-dotenv	Manejo de variables de entorno	`config.py`

¿Por qué OpenCV y no PIL?

OpenCV es superior para procesamiento digital de imágenes porque:

1. **Formato de arrays NumPy nativo**: OpenCV trabaja directamente con arrays NumPy, mientras que PIL usa su propio formato Image
2. **BGR vs RGB**: OpenCV lee en BGR (estándar de visión por computadora), permitiendo control total
3. **Rendimiento**: OpenCV está optimizado en C/C++ para operaciones matriciales
4. **Funcionalidades avanzadas**: Filtros, transformaciones, detección, etc.

PIL solo se usa para una cosa:

```
# OpenCV procesa la imagen
imagen_cv = cv2.imread("foto.jpg")  # Array NumPy
procesar_con_opencv(imagen_cv)

# PIL solo para mostrar en Tkinter (incompatibilidad)
img_pil = Image.fromarray(imagen_cv)
photo = ImageTk.PhotoImage(img_pil)  # Requerido por Tkinter
canvas.create_image(x, y, image=photo)
```

Conceptos de OpenCV Utilizados

Concepto	Descripción	Código
Lectura BGR	OpenCV lee imágenes en formato BGR por defecto	`cv2.imread(path)`
Conversión BGR→RGB	Convertir a RGB para visualización correcta	`cv2.cvtColor(img, cv2.COLOR_BGR2RGB)`
Shape	Dimensiones del array: (alto, ancho, canales)	`imagen.shape` → `(480, 640, 3)`
Indexing	Acceso a píxeles y canales	`imagen[y, x, canal]`
Slicing de canales	Extraer canal individual	`r = imagen[:, :, 0]`
dtype uint8	Tipo de dato: enteros sin signo 0-255	`imagen.dtype` → `uint8`

config.py - Configuración

Propósito

Centraliza la configuración del proyecto, cargando las credenciales de Supabase desde variables de entorno.

Imports

```
import os
from dotenv import load_dotenv
```

Import	Uso
`os`	Acceder a variables de entorno del sistema
`load_dotenv`	Cargar variables desde archivo `.env`

Sección 1: Carga de Variables

```
load_dotenv()

SUPABASE_URL = os.getenv("SUPABASE_URL")
SUPABASE_KEY = os.getenv("SUPABASE_KEY")
```

Explicación:

- `load_dotenv()` lee el archivo `.env` y carga sus valores como variables de entorno
- `os.getenv("NOMBRE")` obtiene el valor de una variable de entorno
- Si la variable no existe, retorna `None`

Sección 2: Validación

```
def validar_configuracion():
    ### Verifica que las credenciales de Supabase esten configuradas
    if not SUPABASE_URL:
        raise ValueError("SUPABASE_URL no esta configurado en el archivo .env")
    if not SUPABASE_KEY:
        raise ValueError("SUPABASE_KEY no esta configurado en el archivo .env")
```

Explicación:

- Verifica que las variables no estén vacías (`not SUPABASE_URL`)
- `raise ValueError` lanza un error descriptivo si algo falla
- Se usa antes de conectar a Supabase para dar errores claros

---

database.py - Capa de Datos

Propósito

Maneja toda la comunicación con la base de datos Supabase. Aísla la lógica de persistencia del resto de la aplicación.

Imports

```
from supabase import create_client
from config import SUPABASE_URL, SUPABASE_KEY, validar_configuracion
```

Import	Uso
`create_client`	Función para crear conexión a Supabase
`config.*`	Credenciales y validación

Sección 1: Cliente Singleton

```
cliente = None

def iniciar_cliente():
    ### Inicializa y retorna el cliente de Supabase
    global cliente
    if cliente is None:
        validar_configuracion()
        cliente = create_client(SUPABASE_URL, SUPABASE_KEY)
    return cliente
```

Explicación:

- `cliente` es una variable global
- **Patrón Singleton**: Solo se crea UNA conexión, sin importar cuántas veces se llame
- `global cliente` permite modificar la variable global dentro de la función
- Si ya existe conexión (`cliente is not None`), la reutiliza

¿Por qué Singleton?

```
# Sin singleton: Crea conexión cada vez (ineficiente)
cliente1 = create_client(url, key)  # Nueva conexión
cliente2 = create_client(url, key)  # Otra conexión

# Con singleton: Reutiliza la misma conexión
cliente1 = iniciar_cliente()  # Crea conexión
cliente2 = iniciar_cliente()  # Retorna la misma conexión
```

Sección 2: Guardar Imagen

```
def guardar_imagen(ancho, alto, datos_rgb):
    ### Guarda los datos de una imagen en Supabase, retorna el ID
    cliente_local = iniciar_cliente()

    datos = {
        "width": ancho,
        "height": alto,
        "rgb_data": datos_rgb
    }

    respuesta = cliente_local.table("images").insert(datos).execute()

    if respuesta.data:
        return respuesta.data[0]["id"]
    else:
        raise Exception("Error al guardar la imagen en la base de datos")
```

Explicación línea por línea:

Línea	Qué hace
`cliente_local = iniciar_cliente()`	Obtiene la conexión a Supabase
`datos = {...}`	Crea diccionario con los campos a insertar
`cliente_local.table("images")`	Selecciona la tabla "images"
`.insert(datos)`	Prepara la operación INSERT
`.execute()`	Ejecuta la query
`respuesta.data[0]["id"]`	Extrae el ID del registro creado

Equivalente SQL:

```
INSERT INTO images (width, height, rgb_data)
VALUES (100, 100, "255,0,0,...")
RETURNING id;
```

Sección 3: Obtener Imagen

```
def obtener_imagen(id_imagen):
    ### Recupera los datos de una imagen por su ID
    cliente_local = iniciar_cliente()

    respuesta = cliente_local.table("images").select("*").eq("id",
id_imagen).execute()

    if respuesta.data:
        return respuesta.data[0]
    else:
        raise Exception(f"No se encontro imagen con ID: {id_imagen}")
```

Explicación:

Método	Qué hace
`.select("*")`	Selecciona todas las columnas
`.eq("id", id_imagen)`	Filtro WHERE id = id_imagen
`.execute()`	Ejecuta la query

Equivalente SQL:

```
SELECT * FROM images WHERE id = 1;
```

Retorna un diccionario:

```
{
    "id": 1,
    "width": 100,
    "height": 100,
    "rgb_data": "255,0,0,255,0,0,...",
    "created_at": "2024-01-15T10:30:00Z"
}
```

---

image_processor.py - Procesamiento de Imágenes

Propósito

Contiene las funciones de conversión de imágenes a string RGB y viceversa, separación de canales, cálculo de tamaño en memoria, y la función compartida `mostrar_en_canvas` para mostrar imágenes en el canvas de Tkinter.

Imports

```
import numpy as np
import cv2
from PIL import Image, ImageTk
import tkinter as tk
```

Import	Uso	Detalles
`numpy`	Operaciones matemáticas con matrices	flatten, reshape, array, zeros_like
`cv2`	Redimensionar imágenes para preview	`cv2.resize()`
`PIL`	Conversión NumPy → ImageTk	`Image.fromarray()`, `ImageTk.PhotoImage()`
`tkinter`	Constante `tk.CENTER` para posicionar en canvas	Solo para `mostrar_en_canvas`

Sección 1: Descomposición - Imagen a String RGB

```
def imagen_a_string_rgb(imagen):
    ### Descompone imagen en valores RGB y convierte a string
    ### Proceso: obtener dimensiones -> aplanar 3D a 1D -> string con comas
```

```
    alto, ancho, canales = imagen.shape
    aplanado = imagen.flatten()
    cadena_rgb = ",".join(map(str, aplanado))
    return cadena_rgb, ancho, alto
```

Explicación paso a paso:

1. Extraer dimensiones del shape

```
alto, ancho, canales = imagen.shape
# Ejemplo: alto=480, ancho=640, canales=3
```

Shape en OpenCV vs PIL:

```
# OpenCV (NumPy array)
imagen.shape = (alto, ancho, canales)  # (480, 640, 3)
               (filas, columnas, depth)

# PIL/Pillow
imagen.size = (ancho, alto)  # (640, 480)
              (width, height)

# ⚠ Orden diferente! Por eso usamos OpenCV
```

2. Aplanar (flatten) - De 3D a 1D

```
aplanado = imagen.flatten()
```

¿Qué hace flatten()?

Convierte una matriz multidimensional en un vector 1D, leyendo los datos en orden **row-major** (por filas).

Ejemplo visual con imagen pequeña (3x2):

```
# Imagen original 3D: shape (2, 3, 3)
imagen = [
    [[255,0,0], [0,255,0], [0,0,255]],    # Fila 0: Rojo, Verde, Azul
    [[255,255,0], [255,0,255], [0,255,255]]  # Fila 1: Amarillo, Magenta, Cian
]

# Después de flatten(): shape (18,)
aplanado = [255,0,0,  0,255,0,  0,0,255,  255,255,0,  255,0,255,  0,255,255]
```

```
└─┬─┘   └─┬─┘   └─┬─┘   └─┬─┘   └─┬─┘   └─┬─┘
Píxel 0   Píxel 1   Píxel 2   Píxel 3   Píxel 4   Píxel 5
```

Orden de lectura:

```
  Columna 0    Columna 1    Columna 2
    ┌─────┐      ┌─────┐      ┌─────┐
F0  │ RGB │ →    │ RGB │ →    │ RGB │ ┐
    └─────┘      └─────┘      └─────┘ │
                                      │
    ┌─────┐      ┌─────┐      ┌─────┐ │ Flatten
F1  │ RGB │ →    │ RGB │ →    │ RGB │ │ lee en
    └─────┘      └─────┘      └─────┘ ┘ este orden

Resultado: [R0,G0,B0, R1,G1,B1, R2,G2,B2, R3,G3,B3, R4,G4,B4, R5,G5,B5]
```

Tamaño del vector aplanado:

```
len(aplanado) = alto × ancho × canales
              = 480 × 640 × 3
              = 921,600 valores
```

3. Convertir a string separado por comas

```
cadena_rgb = ",".join(map(str, aplanado))
```

Desglose de esta línea:

```
# 1. map(str, aplanado): Convierte cada número a string
aplanado = [255, 0, 0, 128, 255, 0]
map(str, aplanado)  →  ["255", "0", "0", "128", "255", "0"]

# 2. ",".join(): Une con comas
",".join(["255", "0", "0", "128", "255", "0"])
→  "255,0,0,128,255,0"
```

Resultado final:

```
cadena_rgb = "255,0,0,0,255,0,0,0,255,255,255,0,255,0,255,0,255,255"
             └──────────────────────── 921,600 valores ────────────────────────┘

# Este string se guardará en la base de datos
```

Sección 2: Reconstrucción - String RGB a Imagen

```
def string_rgb_a_imagen(cadena_rgb, ancho, alto):
    ### Reconstruye imagen desde string de valores RGB
    ### Proceso: parsear string -> array numpy uint8 -> reshape (alto, ancho,
3)
    valores = list(map(int, cadena_rgb.split(",")))
    arr = np.array(valores, dtype=np.uint8)
    imagen = arr.reshape((alto, ancho, 3))
    return imagen
```

Explicación del proceso inverso:

1. Parsear string a lista de enteros

```
valores = list(map(int, cadena_rgb.split(",")))
```

```
# String original
"255,0,0,128,255,0"

# .split(",") → separa por comas
["255", "0", "0", "128", "255", "0"]

# map(int, ...) → convierte cada string a int
[255, 0, 0, 128, 255, 0]
```

2. Crear array NumPy con tipo correcto

```
arr = np.array(valores, dtype=np.uint8)
```

¿Por qué `dtype=np.uint8` es crucial?

```
# Sin especificar dtype (por defecto usa int64)
arr_default = np.array([255, 128, 0])
arr_default.dtype  # int64 (8 bytes por valor)
arr_default.nbytes  # 24 bytes

# Con dtype=np.uint8
arr_uint8 = np.array([255, 128, 0], dtype=np.uint8)
arr_uint8.dtype  # uint8 (1 byte por valor)
arr_uint8.nbytes  # 3 bytes
```

```
# ¡8 veces menos memoria! Y es el formato que espera OpenCV
```

Rango de valores:

```
uint8: 0 a 255 (valores válidos para RGB)
int64: -9,223,372,036,854,775,808 a 9,223,372,036,854,775,807 (desperdicio)
```

3. Reshape - De 1D a 3D

```
imagen = arr.reshape((alto, ancho, 3))
```

¿Qué hace reshape()?

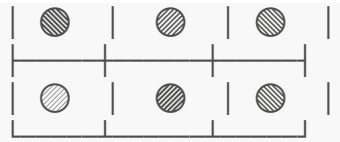
Reorganiza el vector 1D en una matriz 3D sin cambiar los datos, solo su "forma".

Ejemplo con imagen 3x2:

```
# Vector 1D (shape: 18,)
arr = [255,0,0, 0,255,0, 0,0,255, 255,255,0, 255,0,255, 0,255,255]

# Reshape a (alto=2, ancho=3, canales=3)
imagen = arr.reshape((2, 3, 3))

# Resultado:
[
  [ [255,0,0],   [0,255,0],   [0,0,255]   ],  # Fila 0
  [ [255,255,0], [255,0,255], [0,255,255] ]   # Fila 1
]
```

Visualización:

```
Vector 1D:
[255,0,0,0,255,0,0,0,255,255,255,0,255,0,255,0,255,255]
  └─────┘ └─────┘ └─────┘ └─────┘ └─────┘ └─────┘
    P0      P1      P2      P3      P4      P5

Reshape (2, 3, 3):
        Columna 0     Columna 1     Columna 2
Fila 0  [255,0,0]    [0,255,0]    [0,0,255]
Fila 1  [255,255,0]  [255,0,255]  [0,255,255]

Imagen resultante:
┌─────┬─────┬─────┐
```



Condición para reshape:

```
producto_de_dimensiones = alto × ancho × canales

len(arr) debe ser igual a producto_de_dimensiones

Ejemplo:
len(arr) = 18
alto × ancho × canales = 2 × 3 × 3 = 18 ☑

Si fueran diferentes:
reshape((3, 3, 3)) → 3 × 3 × 3 = 27 ❌
# ValueError: cannot reshape array of size 18 into shape (3,3,3)
```

Sección 3: Obtener Canales Separados

```python
def obtener_canales(imagen):
    ### Separa los canales RGB de una imagen
    ### Similar a canales_naturales.py del profesor
    r = imagen[:, :, 0]
    g = imagen[:, :, 1]
    b = imagen[:, :, 2]

    ### Crear imagenes de cada canal
    R = np.zeros_like(imagen)
    R[:, :, 0] = r

    G = np.zeros_like(imagen)
    G[:, :, 1] = g

    B = np.zeros_like(imagen)
    B[:, :, 2] = b

    return R, G, B
```

¿Qué hace esto?

Crea tres imágenes RGB donde cada una muestra solo un canal:

```
Imagen original:          Canal R:            Canal G:            Canal B:
┌──────────┐              ┌──────────┐        ┌──────────┐
```

```
┌──────────┐
│ [255,128,64] │          │ [255, 0, 0] │       │ [ 0,128, 0] │       │ [ 0,
0,64] │
│ [ 0, 255, 0] │    →     │ [  0, 0, 0] │       │ [ 0,255, 0] │       │ [ 0, 0,
0] │
│ [ 0,  0,255] │          │ [  0, 0, 0] │       │ [ 0,  0, 0] │       │ [ 0,
0,255] │
└──────────────┘          └─────────────┘       └─────────────┘
└──────────┘
   Color normal              Solo rojo            Solo verde            Solo
azul
```

Sección 4: Calcular Tamaño en Memoria

```python
def calcular_tamano_imagen(imagen):
    ### Calcula el tamano de una imagen en memoria
    alto, ancho, canales = imagen.shape
    tamano_bytes = alto * ancho * canales
    tamano_kb = tamano_bytes / 1024
    tamano_mb = tamano_kb / 1024

    print(f"Dimensiones: {ancho}x{alto}")
    print(f"Canales: {canales}")
    print(f"Total pixeles: {alto * ancho}")
    print(f"Tamano en memoria: {tamano_kb:.2f} KB ({tamano_mb:.4f} MB)")
```

Cálculo de memoria:

```
Imagen de 1920x1080 (Full HD):

Píxeles = 1920 × 1080 = 2,073,600 píxeles
Valores RGB = 2,073,600 × 3 canales = 6,220,800 valores
Bytes = 6,220,800 × 1 byte (uint8) = 6,220,800 bytes
                                   = 6,075 KB
                                   ≈ 5.93 MB sin comprimir
```

Sección 5: Mostrar Imagen en Canvas de Tkinter

```python
def mostrar_en_canvas(canvas, ventana, imagen_rgb, foto_tk_ref, margen=0.9):
    ### Redimensiona imagen y la muestra centrada en un canvas de Tkinter
    ### foto_tk_ref es una lista [None] para mantener la referencia al ImageTk
    ventana.update()
    ancho_canvas = canvas.winfo_width()
    alto_canvas = canvas.winfo_height()
    if ancho_canvas < 10:
        ancho_canvas, alto_canvas = 640, 480
```

```
    alto, ancho, _ = imagen_rgb.shape
    ratio = min(ancho_canvas / ancho, alto_canvas / alto) * margen
    vista_previa = cv2.resize(imagen_rgb, (int(ancho * ratio), int(alto *
ratio)))

    foto_tk_ref[0] = ImageTk.PhotoImage(Image.fromarray(vista_previa))
    canvas.delete("all")
    canvas.create_image(ancho_canvas // 2, alto_canvas // 2,
image=foto_tk_ref[0], anchor=tk.CENTER)
```

Explicación:

Esta función es **compartida** por `gui_upload.py` y `gui_viewer.py`. Antes, cada GUI tenía su propia versión duplicada. Ahora vive en `image_processor.py` como función reutilizable.

Paso	Código	Qué hace
1	`ventana.update()`	Fuerza a Tkinter a calcular las dimensiones reales del canvas
2	`canvas.winfo_width()`	Obtiene el ancho actual del canvas en píxeles
3	`min(ratio_ancho, ratio_alto) * margen`	Calcula la escala manteniendo proporción con 10% de margen
4	`cv2.resize(imagen_rgb, ...)`	Redimensiona la imagen con OpenCV
5	`Image.fromarray(vista_previa)`	Convierte NumPy array a PIL Image
6	`ImageTk.PhotoImage(...)`	Convierte PIL Image a formato Tkinter
7	`canvas.create_image(...)`	Muestra la imagen centrada en el canvas

¿Por qué `foto_tk_ref` es una lista `[None]`?

```
### Si fuera una variable normal, Python la trataría como local
foto_tk = None
def mostrar():
    foto_tk = ImageTk.PhotoImage(...)  ### Crea variable LOCAL, no modifica la
exterior
    ### Al salir de la función, foto_tk se borra → imagen desaparece del canvas

### Con lista, modificamos el contenido sin crear variable nueva
foto_tk = [None]
def mostrar():
    foto_tk[0] = ImageTk.PhotoImage(...)  ### Modifica el contenido de la lista
    ### La referencia se mantiene viva → imagen se muestra correctamente
```

---

gui_upload.py - Interfaz de Captura

Propósito

Ventana gráfica para capturar imágenes desde la cámara web, mostrar feed en vivo, guardar la foto en la carpeta `img/`, y almacenar los datos RGB en Supabase. Usa **OpenCV para captura de cámara y procesamiento**. La conversión a ImageTk para Tkinter se delega a `mostrar_en_canvas()` en `image_processor.py`.

Imports

```
import tkinter as tk
from tkinter import messagebox
import cv2
import os
from datetime import datetime
from image_processor import imagen_a_string_rgb, calcular_tamano_imagen,
mostrar_en_canvas
from database import guardar_imagen
```

Import	Uso	Cuándo se usa
`tkinter`	Biblioteca GUI estándar de Python	Ventanas, botones, canvas
`messagebox`	Ventanas de alerta/información	Errores, confirmaciones
`cv2`	OpenCV - Captura de cámara y procesamiento	Cámara, conversión BGR/RGB, guardar foto
`os`	Rutas de archivos	Construir ruta a carpeta `img/`
`datetime`	Marca de tiempo	Nombre único para cada foto
`mostrar_en_canvas`	Mostrar imagen en canvas de Tkinter	Feed en vivo y preview de foto
`guardar_imagen`	Insertar datos en Supabase	Guardar imagen descompuesta

Arquitectura: Función con Closures

El módulo usa una **función principal** `abrir_ventana_captura()` que contiene funciones internas (closures). Las variables compartidas se manejan como **listas de un elemento** para poder modificarlas desde las funciones internas.

```
def abrir_ventana_captura(parent=None):
    ### Crea y muestra la ventana de captura de imagenes con camara

    ### Variables compartidas (listas para poder modificarlas en closures)
    captura = [None]        ### cv2.VideoCapture
    camara_activa = [False] ### flag para el loop
```

```
    imagen_actual = [None]  ### numpy array RGB de la foto capturada
    foto_tk = [None]        ### referencia a ImageTk para que no se borre
```

¿Por qué listas en vez de variables normales?

```
### Con variable normal, Python crea una variable LOCAL dentro de la closure
camara_activa = False
def abrir_camara():
    camara_activa = True  ### Crea variable LOCAL, no modifica la exterior

### Con lista, modificamos el CONTENIDO sin crear variable nueva
camara_activa = [False]
def abrir_camara():
    camara_activa[0] = True  ### Modifica el contenido de la lista exterior
```

Sección 1: Abrir Cámara y Feed en Vivo

```
def abrir_camara():
    ### Inicializa la camara y arranca el feed en vivo
    if camara_activa[0]:
        return

    captura[0] = cv2.VideoCapture(0)  ### 0 para la camara por defecto

    if not captura[0].isOpened():
        messagebox.showerror("Error", "No se pudo acceder a la camara")
        captura[0] = None
        return

    camara_activa[0] = True
    actualizar_feed()

def actualizar_feed():
    ### Lee un frame de la camara y lo muestra en el canvas
    if not camara_activa[0] or captura[0] is None or not captura[0].isOpened():
        return

    ret, frame = captura[0].read()  ### lee un frame de la camara
    if ret:
        imagen_rgb = cv2.cvtColor(frame, cv2.COLOR_BGR2RGB)  ### convierte BGR
a RGB
        mostrar_en_canvas(canvas, ventana, imagen_rgb, foto_tk, 0.95)

    ventana.after(30, actualizar_feed)  ### ~33 FPS
```

Explicación detallada:

1. Abrir la cámara con OpenCV

```
captura[0] = cv2.VideoCapture(0)
```

- `cv2.VideoCapture(0)` abre la cámara por defecto del sistema (índice 0)
- Similar al estilo del profesor en scripts `video.py` y `*_live.py`
- Retorna un objeto de captura que permite leer frames

2. Loop de actualización (Tkinter-friendly)

```
def actualizar_feed():
    if not camara_activa[0] or captura[0] is None or not captura[0].isOpened():
        return

    ret, frame = captura[0].read()
    if ret:
        imagen_rgb = cv2.cvtColor(frame, cv2.COLOR_BGR2RGB)
        mostrar_en_canvas(canvas, ventana, imagen_rgb, foto_tk, 0.95)

    ventana.after(30, actualizar_feed)
```

- **No usa** `while True` (bloquearía la GUI de Tkinter)
- Usa `ventana.after(30, ...)` para programar la siguiente actualización
- Cada 30ms (~33 FPS) lee un nuevo frame de la cámara
- Convierte BGR→RGB antes de mostrar
- Usa `mostrar_en_canvas()` de `image_processor.py` para redimensionar y mostrar

3. Tomar foto, guardar en img/ y detener cámara

```
def tomar_foto():
    ### Captura el frame actual, detiene la camara y muestra la foto
    ret, frame = captura[0].read()
    imagen_actual[0] = cv2.cvtColor(frame, cv2.COLOR_BGR2RGB)

    ### Detener camara
    detener_camara()

    ### Guardar foto en img/
    carpeta_img = os.path.join(os.path.dirname(os.path.abspath(__file__)),
"img")
    marca_tiempo = datetime.now().strftime("%Y%m%d_%H%M%S")
    ruta_foto = os.path.join(carpeta_img, f"foto_{marca_tiempo}.png")
    imagen_bgr = cv2.cvtColor(imagen_actual[0], cv2.COLOR_RGB2BGR)
    cv2.imwrite(ruta_foto, imagen_bgr)
```

```
    ### Mostrar info y preview
    calcular_tamano_imagen(imagen_actual[0])
    mostrar_en_canvas(canvas, ventana, imagen_actual[0], foto_tk)
```

- Captura el frame actual como numpy array RGB
- Detiene la cámara liberando el recurso con `cap.release()`
- Guarda la foto en `img/foto_YYYYMMDD_HHMMSS.png` (convierte RGB→BGR para `cv2.imwrite`)
- Muestra la ruta real del archivo en la interfaz
- Muestra preview estática usando `mostrar_en_canvas()`

Sección 2: Guardar en Base de Datos

```
def guardar_en_bd():
    ### Descompone la imagen en RGB y la guarda en Supabase
    if imagen_actual[0] is None:
        messagebox.showwarning("Advertencia", "No hay imagen para guardar")
        return

    try:
        btn_guardar.config(state=tk.DISABLED, text="Procesando...")
        ventana.update()

        cadena_rgb, ancho, alto = imagen_a_string_rgb(imagen_actual[0])
        id_imagen = guardar_imagen(ancho, alto, cadena_rgb)

        lbl_id.config(text=f"Imagen guardada con ID: {id_imagen}", fg="green")
        messagebox.showinfo("Exito", f"Imagen guardada correctamente.\nID:
{id_imagen}")

    except Exception as e:
        messagebox.showerror("Error", f"Error al guardar:\n{str(e)}")

    finally:
        btn_guardar.config(state=tk.NORMAL, text="Guardar en Base de Datos")
```

Flujo completo:

```
imagen_actual[0] (NumPy array RGB)
        │
        ▼
imagen_a_string_rgb()  [NumPy]
        │
        ├──→ imagen.flatten()
        ├──→ ",".join(map(str, aplanado))
        │
        ▼
(cadena_rgb, ancho, alto)
        │
```

```
        ▼
guardar_imagen(ancho, alto, cadena_rgb)  [Supabase]
        │
        ▼
ID generado
```

¿Por qué `ventana.update()`?

```
btn_guardar.config(text="Procesando...")
ventana.update()  ### Sin esto, el texto no cambia hasta que termine
```

Tkinter es de un solo hilo. Si no llamas a `.update()`, los cambios visuales se quedan "pendientes" hasta que termine la función:

```
### Sin update()
btn_guardar.config(text="Procesando...")  ### Se queda pendiente
time.sleep(5)  ### Usuario ve el botón sin cambiar

### Con update()
btn_guardar.config(text="Procesando...")
ventana.update()  ### Cambia inmediatamente
time.sleep(5)  ### Usuario VE "Procesando..."
```

---

gui_viewer.py - Interfaz de Visualización

Propósito

Ventana para consultar imágenes por ID, reconstruirlas desde la base de datos usando **NumPy**, y mostrarlas. Usa la función compartida `mostrar_en_canvas()` de `image_processor.py` para la visualización.

Imports

```
import tkinter as tk
from tkinter import messagebox
from image_processor import string_rgb_a_imagen, mostrar_en_canvas
from database import obtener_imagen
```

Import	Uso
`tkinter`	Ventana, canvas, labels, botones
`messagebox`	Alertas de error/advertencia

Import	Uso
`string_rgb_a_imagen`	Reconstruir imagen desde string RGB
`mostrar_en_canvas`	Redimensionar y mostrar en canvas
`obtener_imagen`	Consultar imagen por ID en Supabase

Nota: Este módulo **no importa** `cv2` ni `PIL` directamente. Todo el procesamiento de imagen y la conversión a ImageTk se delegan a `image_processor.py`.

Arquitectura: Función con Closures

Igual que `gui_upload.py`, usa una función principal con closures:

```
def abrir_ventana_visor(parent=None):
    ### Crea y muestra la ventana de consulta y reconstruccion de imagenes

    ### Variables
    imagen_actual = [None]
    foto_tk = [None]
```

Sección Principal: Consultar y Reconstruir Imagen

```
def consultar_imagen():
    ### Consulta la imagen por ID y la reconstruye
    id_texto = entrada_id.get().strip()

    if not id_texto:
        messagebox.showwarning("Advertencia", "Ingresa un ID de imagen")
        return

    try:
        id_imagen = int(id_texto)
    except ValueError:
        messagebox.showerror("Error", "El ID debe ser un numero entero")
        return

    try:
        ### Consultar base de datos
        datos = obtener_imagen(id_imagen)

        ### Extraer datos
        ancho = datos["width"]
        alto = datos["height"]
        cadena_rgb = datos["rgb_data"]

        ### Reconstruir imagen (retorna numpy array RGB)
        imagen_actual[0] = string_rgb_a_imagen(cadena_rgb, ancho, alto)
```

```
        ### Mostrar imagen reconstruida
        mostrar_en_canvas(canvas, ventana, imagen_actual[0], foto_tk)

    except Exception as e:
        messagebox.showerror("Error", f"Error al consultar:\n{str(e)}")
```

Flujo de reconstrucción completo:

```
ID ingresado por usuario (texto)
        │
        ├─→ Validar que sea entero
        │
        ▼
obtener_imagen(id)  [Consulta Supabase]
        │
        ▼
{
  "id": 1,
  "width": 640,
  "height": 480,
  "rgb_data": "255,0,0,0,255,0,...",
  "created_at": "2024-..."
}
        │
        ▼
string_rgb_a_imagen(cadena_rgb, ancho, alto)  [NumPy]
        │
        ├─→ cadena_rgb.split(",")            ### String → Lista
        ├─→ np.array(valores, dtype=uint8)   ### Lista → Array NumPy
        ├─→ arr.reshape((alto, ancho, 3))    ### 1D → 3D
        │
        ▼
Array NumPy RGB (alto, ancho, 3) uint8
        │
        ▼
mostrar_en_canvas()  [image_processor.py]
        │
        ├─→ cv2.resize()              ### Redimensionar
        ├─→ Image.fromarray()          ### NumPy → PIL
        ├─→ ImageTk.PhotoImage()       ### PIL → ImageTk
        │
        ▼
Mostrar en Tkinter Canvas
```

Detalle de cada paso:

1. Validación del ID

```
id_texto = entrada_id.get().strip()

if not id_texto:
    messagebox.showwarning("Advertencia", "Ingresa un ID de imagen")
    return

try:
    id_imagen = int(id_texto)
except ValueError:
    messagebox.showerror("Error", "El ID debe ser un numero entero")
    return
```

Casos que maneja:

Input	Resultado
`" 123 "`	`id_imagen = 123` (strip elimina espacios)
`"abc"`	ValueError → Mensaje de error
`""`	Warning "Ingresa un ID"
`"12.5"`	ValueError (no es entero)
`"0"`	`id_imagen = 0` (válido aunque probablemente no exista)

2. Consultar base de datos

```
datos = obtener_imagen(id_imagen)
```

Internamente ejecuta:

```
respuesta = cliente_local.table("images")\
    .select("*")\
    .eq("id", id_imagen)\
    .execute()
```

Retorna un diccionario:

```
{
    "id": 1,
    "width": 640,
    "height": 480,
    "rgb_data": "255,0,0,0,255,0,0,0,255,...",  ### String MUY largo
    "created_at": "2024-01-15T10:30:00.000Z"
}
```

3. Reconstruir y mostrar imagen

```
### Reconstruir imagen desde string
imagen_actual[0] = string_rgb_a_imagen(cadena_rgb, ancho, alto)

### Mostrar en canvas (redimensiona, convierte a ImageTk, centra)
mostrar_en_canvas(canvas, ventana, imagen_actual[0], foto_tk)
```

La reconstrucción (`string_rgb_a_imagen`) y la visualización (`mostrar_en_canvas`) están en `image_processor.py`, manteniendo la separación de responsabilidades.

¿Se pierde calidad?

```
### NO, si guardas y reconstruyes en el mismo formato

Cámara → cap.read() → Array NumPy → flatten() → String
                                        ↓
                                   [255, 0, 0, ...]
                                        ↑
String → split() → Array NumPy → reshape() → Imagen reconstruida

Comparación:
np.array_equal(imagen_original, imagen_reconstruida)  ### True
```

---

main.py - Punto de Entrada

Propósito

Ventana principal que permite abrir las otras dos interfaces (captura y visor).

Imports

```
import tkinter as tk
from gui_upload import abrir_ventana_captura
from gui_viewer import abrir_ventana_visor
```

Los imports son a **nivel de módulo** (no lazy imports dentro de funciones), ya que las dependencias son directas.

Estructura

```python
def main():
    ### Funcion principal - crea la ventana y arranca la aplicacion

    root = tk.Tk()
    root.title("Image Decomposer")
    root.geometry("400x300")
    root.resizable(False, False)

    ### Centrar ventana
    root.update_idletasks()
    ancho = root.winfo_width()
    alto = root.winfo_height()
    x = (root.winfo_screenwidth() // 2) - (ancho // 2)
    y = (root.winfo_screenheight() // 2) - (alto // 2)
    root.geometry(f"{ancho}x{alto}+{x}+{y}")

    ### Botones
    tk.Button(
        text="Cargar Imagen",
        command=lambda: abrir_ventana_captura(root),
    ).pack()

    tk.Button(
        text="Ver Imagen",
        command=lambda: abrir_ventana_visor(root),
    ).pack()

    root.mainloop()


if __name__ == "__main__":
    main()
```

Explicación de `if __name__ == "__main__":`

```python
if __name__ == "__main__":
    main()
```

- `__name__` es una variable especial de Python
- Vale `"__main__"` solo cuando ejecutas el archivo directamente
- Si importas el archivo desde otro módulo, `__name__` será el nombre del módulo

```python
### Si ejecutas: python main.py
__name__ == "__main__"  ### True, ejecuta main()

### Si importas: from main import main
__name__ == "main"  ### False, no ejecuta main()
```

Conceptos Avanzados de OpenCV

1. BGR vs RGB - ¿Por qué OpenCV es diferente?

Historia:
OpenCV fue creado a finales de los 90. Las cámaras de video analógicas de esa época usaban señales BGR (Blue-Green-Red) por razones de compatibilidad con televisores antiguos.

Implicaciones:

```
# Leer imagen
imagen = cv2.imread("foto.jpg")  # Lee en BGR

# Ver el color de un píxel rojo puro
print(imagen[0, 0])  # [0, 0, 255] ← [B, G, R]

# Sin conversión, se vería azul
plt.imshow(imagen)  # ❌ Colores invertidos

# Conversión correcta
imagen_rgb = cv2.cvtColor(imagen, cv2.COLOR_BGR2RGB)
plt.imshow(imagen_rgb)  # ☑ Colores correctos
```

Tabla de conversiones comunes:

Desde	Hacia	Código OpenCV
BGR	RGB	`cv2.COLOR_BGR2RGB`
RGB	BGR	`cv2.COLOR_RGB2BGR`
BGR	Grises	`cv2.COLOR_BGR2GRAY`
RGB	HSV	`cv2.COLOR_RGB2HSV`
Grises	BGR	`cv2.COLOR_GRAY2BGR`

2. Shape, Dtype y Size en NumPy

Shape - Dimensiones del array

```
imagen.shape  # (alto, ancho, canales)
```

Diferentes tipos de imágenes:

```
# Imagen a color
imagen_rgb.shape = (480, 640, 3)
                   alto  ancho  RGB

# Imagen en escala de grises
imagen_gray.shape = (480, 640)
                    alto  ancho (sin canal)

# Imagen con transparencia
imagen_rgba.shape = (480, 640, 4)
                    alto  ancho  RGBA

# Video frame (mismo que imagen)
frame.shape = (1080, 1920, 3)
              alto   ancho  RGB
```

Dtype - Tipo de datos

```
imagen.dtype  # uint8, uint16, float32, etc.
```

Tipos comunes:

Tipo	Rango	Bytes	Uso
`uint8`	0 - 255	1	**Imágenes estándar RGB**
`uint16`	0 - 65,535	2	Imágenes médicas, RAW
`float32`	0.0 - 1.0	4	Procesamiento intermedio
`float64`	0.0 - 1.0	8	Alta precisión (raro)

Conversiones:

```
# uint8 (0-255) a float32 (0.0-1.0)
imagen_float = imagen.astype(np.float32) / 255.0

# float32 (0.0-1.0) a uint8 (0-255)
imagen_uint8 = (imagen_float * 255).astype(np.uint8)
```

Size - Total de elementos

```
imagen.size = alto × ancho × canales
```

Cálculo de memoria:

```
imagen.shape = (1080, 1920, 3)
imagen.dtype = uint8

Total elementos = 1080 × 1920 × 3 = 6,220,800
Bytes por elemento = 1 byte (uint8)
Memoria total = 6,220,800 bytes = 6.22 MB

# Verificar
imagen.nbytes  # 6220800
```

3. Indexing y Slicing - Acceso a píxeles

Sintaxis básica

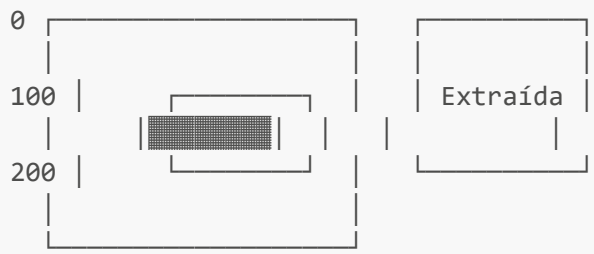
```
imagen[fila, columna, canal]
```

Ejemplos:

```
# Acceder a un píxel específico (fila 100, columna 200)
pixel = imagen[100, 200]  # [R, G, B]

# Acceder al valor rojo de ese píxel
rojo = imagen[100, 200, 0]

# Cambiar un píxel a blanco
imagen[100, 200] = [255, 255, 255]

# Región rectangular (ROI - Region of Interest)
region = imagen[100:200, 300:400]  # Filas 100-199, Columnas 300-399

# Primer canal (rojo) completo
canal_rojo = imagen[:, :, 0]

# Invertir imagen verticalmente
imagen_invertida = imagen[::-1, :, :]

# Invertir imagen horizontalmente (espejo)
imagen_espejo = imagen[:, ::-1, :]
```

Visualización de slicing:

```
Imagen completa:         Región [100:200, 300:400]:
  0  100 200 300 400         300      400
```



4. Flatten y Reshape - Transformaciones

Flatten - 3D a 1D

```
flat = imagen.flatten()
```

Visualización:

```
imagen.shape = (2, 3, 3)  # 2 filas, 3 columnas, 3 canales

Imagen 3D:
Fila 0: [[255,0,0], [0,255,0], [0,0,255]]
Fila 1: [[255,255,0], [0,255,255], [255,0,255]]

flat.shape = (18,)  # Vector 1D

[255, 0, 0, 0, 255, 0, 0, 0, 255, 255, 255, 0, 0, 255, 255, 255, 0, 255]
 └ P0 ┘  └ P1 ┘  └ P2 ┘  └  P3  ┘  └  P4  ┘  └  P5  ┘
```

Orden de lectura (row-major / C order):

```
1. Lee fila 0, columna 0, todos los canales: [255, 0, 0]
2. Lee fila 0, columna 1, todos los canales: [0, 255, 0]
3. Lee fila 0, columna 2, todos los canales: [0, 0, 255]
4. Lee fila 1, columna 0, todos los canales: [255, 255, 0]
... y así sucesivamente
```

Reshape - Cambiar forma sin copiar datos

```
nueva_forma = arr.reshape((nuevo_alto, nuevo_ancho, 3))
```

Ejemplo práctico:

```
# Vector 1D de 18 elementos
arr = np.array([255,0,0, 0,255,0, 0,0,255, 255,255,0, 0,255,255, 255,0,255])

# Reshape a diferentes formas (todas válidas)
img_2x3 = arr.reshape((2, 3, 3))  # 2×3×3 = 18 ☑
img_3x2 = arr.reshape((3, 2, 3))  # 3×2×3 = 18 ☑
img_1x6 = arr.reshape((1, 6, 3))  # 1×6×3 = 18 ☑

# Reshape inválido
img_4x4 = arr.reshape((4, 4, 3))  # 4×4×3 = 48 ❌ ValueError
```

Reshape no copia datos (eficiente):

```
original = np.array([1, 2, 3, 4, 5, 6])
reshaped = original.reshape((2, 3))

reshaped[0, 0] = 99
print(original)  # [99, 2, 3, 4, 5, 6] ← ¡También cambió!

# Son dos "vistas" del mismo bloque de memoria
```

5. Interpolación en resize()

Cuando redimensionas una imagen, necesitas "inventar" píxeles nuevos (aumentar tamaño) o "combinar" píxeles existentes (reducir tamaño).

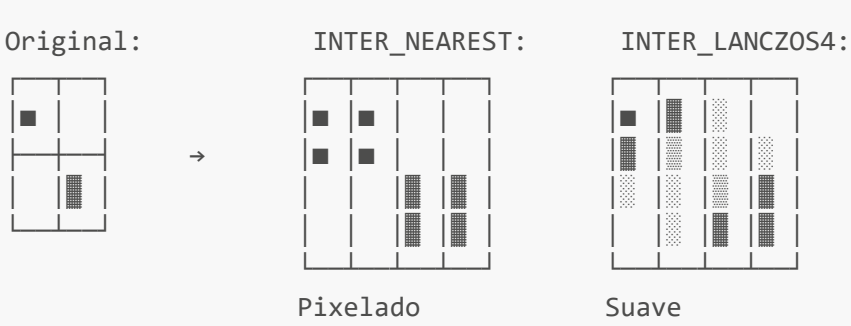
Métodos disponibles

```
cv2.resize(imagen, (nuevo_ancho, nuevo_alto), interpolation=METODO)
```

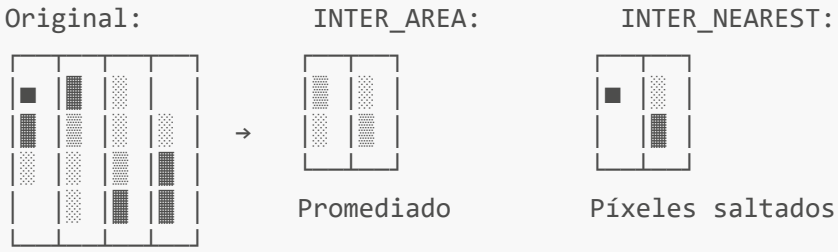
Método	Calidad	Velocidad	Mejor para
INTER_NEAREST	★	⚡⚡⚡⚡⚡	Píxel art, imágenes pequeñas, aumentar tamaño conservando píxeles
INTER_LINEAR	★★★	⚡⚡⚡⚡	Uso general, buen balance
INTER_CUBIC	★★★★	⚡⚡⚡	Reducir tamaño, alta calidad
INTER_LANCZOS4	★★★★★	⚡⚡	**Máxima calidad, previews, impresión**
INTER_AREA	★★★★	⚡⚡⚡⚡	Reducir tamaño rápidamente

Ejemplos visuales

Aumentar tamaño (upscaling) 2x2 → 4x4:



Reducir tamaño (downscaling) 4x4 → 2x2:



¿Cuál usar?

```
# Máxima calidad (previews, interfaz usuario)
cv2.resize(img, (ancho, alto), interpolation=cv2.INTER_LANCZOS4)

# Balance calidad/velocidad (procesamiento en tiempo real)
cv2.resize(img, (ancho, alto), interpolation=cv2.INTER_LINEAR)

# Reducir tamaño rápido (miniaturas, batch processing)
cv2.resize(img, (ancho, alto), interpolation=cv2.INTER_AREA)

# Píxel art / imágenes de juegos retro
cv2.resize(img, (ancho, alto), interpolation=cv2.INTER_NEAREST)
```

6. Operaciones matemáticas con arrays

Operaciones elemento por elemento

```
# Dividir por 2 (oscurecer imagen)
imagen_oscura = imagen // 2

# Multiplicar por 1.5 (aclarar)
imagen_clara = np.clip(imagen * 1.5, 0, 255).astype(np.uint8)

# Invertir colores (negativo)
```

```
imagen_negativa = 255 - imagen

# Binarizar (blanco o negro)
imagen_binaria = np.where(imagen > 127, 255, 0).astype(np.uint8)
```

Operaciones entre imágenes

```
# Promedio de dos imágenes (blend 50/50)
blend = cv2.addWeighted(img1, 0.5, img2, 0.5, 0)

# Diferencia absoluta
diferencia = cv2.absdiff(img1, img2)

# Máscara (mostrar solo donde mask > 0)
resultado = cv2.bitwise_and(imagen, imagen, mask=mascara)
```

Estadísticas

```
# Valor mínimo, máximo, promedio
min_val = np.min(imagen)
max_val = np.max(imagen)
promedio = np.mean(imagen)

# Por canal
promedio_r = np.mean(imagen[:, :, 0])
promedio_g = np.mean(imagen[:, :, 1])
promedio_b = np.mean(imagen[:, :, 2])

# Desviación estándar (contraste)
std = np.std(imagen)
```

7. Espacios de color

OpenCV puede convertir entre múltiples espacios de color:

RGB vs HSV

```
# RGB: Red, Green, Blue (colores luz)
rgb = [255, 0, 0]  # Rojo puro

# HSV: Hue, Saturation, Value (tono, saturación, brillo)
hsv = cv2.cvtColor(imagen_rgb, cv2.COLOR_RGB2HSV)
```

¿Cuándo usar HSV?

HSV es mejor para:

- Detección de objetos por color (range de colores)
- Ajustar brillo sin cambiar el color
- Segmentación por color

```
# Ejemplo: Detectar objetos rojos
hsv = cv2.cvtColor(imagen, cv2.COLOR_BGR2HSV)
rojo_bajo = np.array([0, 100, 100])
rojo_alto = np.array([10, 255, 255])
mascara = cv2.inRange(hsv, rojo_bajo, rojo_alto)
```

Escala de grises

```
# Conversión a grises
gris = cv2.cvtColor(imagen_rgb, cv2.COLOR_RGB2GRAY)

# Fórmula: Gris = 0.299*R + 0.587*G + 0.114*B
# (Pondera más el verde porque el ojo humano es más sensible)
```

Resumen de Flujos Completos

Flujo de Captura

```
1. Usuario: main.py → Click "Cargar Imagen"
2. GUI: gui_upload.py → abrir_ventana_captura()
3. Usuario: Click "Abrir Cámara"
4. GUI: cv2.VideoCapture(0) → Feed en vivo con mostrar_en_canvas() (loop con
after(30))
5. Usuario: Click "Tomar Foto"
6. GUI: cap.read() → frame BGR → cv2.cvtColor(BGR → RGB) → Array NumPy RGB
7. GUI: cap.release() → Cámara liberada
8. GUI: cv2.imwrite() → Guarda foto en img/foto_YYYYMMDD_HHMMSS.png
9. GUI: calcular_tamano_imagen() → Imprime dimensiones y tamaño
10. GUI: mostrar_en_canvas() → Mostrar preview
11. Usuario: Click "Guardar en Base de Datos"
12. Procesador: imagen.flatten() → Vector 1D → ",".join() → String RGB
13. Database: guardar_imagen() → INSERT en Supabase
14. Database: Retorna ID
15. GUI: Mostrar ID generado
```

Flujo de Consulta

```
1. Usuario: main.py → Click "Ver Imagen"
2. GUI: gui_viewer.py → abrir_ventana_visor()
3. Usuario: Ingresa ID → Click "Consultar"
4. GUI: Validar ID (int)
5. Database: obtener_imagen() → SELECT de Supabase
6. Database: Retorna {width, height, rgb_data}
7. Procesador: cadena_rgb.split(",") → Lista
8. Procesador: np.array(..., uint8) → Array 1D
9. Procesador: arr.reshape(alto, ancho, 3) → Array 3D
10. GUI: mostrar_en_canvas() → Redimensionar + ImageTk → Mostrar
11. Usuario: Ve imagen reconstruida (idéntica al original)
```

Comparación: OpenCV vs PIL

Aspecto	OpenCV (cv2)	PIL/Pillow
Formato de datos	NumPy array (alto, ancho, 3)	Objeto Image
Orden de canales	BGR por defecto	RGB
Rendimiento	⚡⚡⚡⚡⚡ C/C++ optimizado	⚡⚡⚡ Python
Operaciones	Miles (filtros, detección, etc.)	Básicas (abrir, guardar, redimensionar)
Integración NumPy	Nativa	Requiere conversión
Lectura de archivos	`cv2.imread()`	`Image.open()`
Redimensionar	`cv2.resize()`	`img.thumbnail()` / `img.resize()`
Conversión de color	`cv2.cvtColor()`	`img.convert()`
Mostrar en Tkinter	❌ Necesita conversión a PIL	☑ Via ImageTk

Decisión de arquitectura:

```
Procesamiento pesado → OpenCV (rápido, potente)
Mostrar en Tkinter → PIL/ImageTk (único compatible)
```

Conceptos Clave de Python Usados

Concepto	Dónde se usa	Explicación
Singleton	`database.py`	Una sola instancia de conexión a BD
Closures	`gui_upload.py`, `gui_viewer.py`	Funciones internas que acceden a variables externas

Concepto	Dónde se usa	Explicación
Listas mutables	`[None]` en GUIs	Truco para modificar variables desde closures
Try/except	Todos los handlers de GUI	Manejo de errores robusto
Global	`global cliente`	Modificar variable global (singleton)
f-strings	`f"ID: {id}"`	Interpolación de strings
map()	`map(str, aplanado)`	Aplicar función a iterable
Lambda	Tkinter callbacks	Funciones anónimas
### comentarios	Todos los archivos	Comentarios estilo del curso

Conceptos Clave de OpenCV/NumPy Usados

Concepto	Código	Explicación
Shape	`imagen.shape`	Dimensiones: (alto, ancho, canales)
Dtype	`dtype=np.uint8`	Tipo de datos: 0-255
Slicing	`imagen[:, :, 0]`	Extraer canal rojo
Flatten	`imagen.flatten()`	3D → 1D
Reshape	`arr.reshape((h,w,3))`	1D → 3D
cvtColor	`cv2.cvtColor(img, cv2.COLOR_BGR2RGB)`	Conversión BGR→RGB
resize	`cv2.resize(img, (w,h), interpolation)`	Cambiar tamaño
VideoCapture	`cv2.VideoCapture(0)`	Capturar frames de cámara
imwrite	`cv2.imwrite(path, img)`	Guardar imagen en disco

Apéndice: Tamaños de Imágenes Comunes

Resolución	Dimensiones	Píxeles	Memoria sin comprimir (RGB)
HD Ready	1280 × 720	921,600	2.64 MB
Full HD	1920 × 1080	2,073,600	5.93 MB
2K	2560 × 1440	3,686,400	10.55 MB
4K UHD	3840 × 2160	8,294,400	23.73 MB
8K UHD	7680 × 4320	33,177,600	94.92 MB

Nota: Archivos PNG/JPG son mucho más pequeños debido a compresión (3x - 20x).