Instituto Politécnico Nacional
Escuela Superior de Cómputo
Ingeniería en
Sistemas Computacionales

Unidad de Aprendizaje:
VISIÓN POR COMPUTADORA

Grupo: 7CV6

Práctica 1:

Imagen digital

Alumno:

Bautista Ríos Alfredo

Docente: Saul De la O Torres

Fecha de entrega:
11/09/2025

Introducción

El procesamiento digital de imágenes constituye una de las áreas de mayor relevancia en la informática contemporánea. Desde sus aplicaciones en medicina, seguridad, industria automotriz, entretenimiento y redes sociales, hasta su papel central en la investigación científica y el desarrollo de sistemas de inteligencia artificial, la capacidad de manipular y analizar imágenes es hoy en día una competencia fundamental para cualquier ingeniero en computación. Dentro de este vasto campo, el estudio de los **modelos de color** y la separación de sus componentes básicos es uno de los primeros pasos hacia un entendimiento más profundo de cómo se representa y procesa la información visual en un sistema digital.

Una imagen digital puede entenderse como una matriz bidimensional de valores que representan la intensidad de la luz en diferentes longitudes de onda. En el modelo RGB (Red, Green, Blue), cada pixel se compone de tres valores de 8 bits que describen la contribución de los colores primarios rojo, verde y azul. La combinación de estos tres componentes en distintas proporciones permite representar un espectro muy amplio de colores, lo que hace que el modelo RGB sea el estándar en dispositivos electrónicos como cámaras, pantallas y software de edición. Manipular cada uno de estos canales por separado posibilita extraer información específica, realizar análisis estadísticos, aplicar filtros o incluso transformar la imagen hacia otros modelos de color como HSV o YCbCr.

Los materiales proporcionados en clase muestran de manera clara cómo implementar esta lógica en **Java**. Mediante clases como `LectorDeImagen` y `FrameComponenteRGB`, se ejemplifica la lectura de una imagen con `BufferedImage`, la extracción de sus pixeles como enteros empaquetados (`0x00RRGGBB`) y la posterior separación de cada canal usando enmascaramientos y corrimientos de bits. Así, por ejemplo, `(pix & 0x00ff0000) >> 16` nos da el valor del rojo, `(pix & 0x0000ff00) >> 8` el del verde y `(pix & 0x000000ff)` el del azul. Estas operaciones muestran cómo un concepto abstracto como el color se traduce en operaciones concretas sobre representaciones binarias.

La importancia de comprender este proceso va más allá de la teoría. Al implementar estos algoritmos en código, los estudiantes fortalecen su entendimiento de temas como estructuras de datos, operaciones bit a bit, representación numérica y manejo de memoria. Además, se sientan las bases para prácticas más complejas de visión por computadora, como la detección de bordes, el reconocimiento de patrones o la aplicación de filtros convolucionales. Por ello, el estudio de la lectura y separación de componentes RGB no solo tiene un valor pedagógico introductorio, sino que también prepara al alumno para abordar proyectos más avanzados en el área de la inteligencia artificial y el machine learning.

En esta práctica, se trasladó la lógica presentada en Java hacia una implementación en **C++ con GTK+**, lo que permitió reforzar el aprendizaje en un entorno diferente y con herramientas del ecosistema Linux. El uso de GTK hizo posible construir una interfaz gráfica que, además de cargar y mostrar imágenes, ofrece controles interactivos para modificar parámetros como el brillo y el contraste. Así, el alumno no solo trabaja con los conceptos de

manipulación de pixeles, sino que también adquiere experiencia en el diseño de interfaces de usuario, el manejo de eventos y la integración de bibliotecas externas.

Asimismo, se integró un módulo adicional para **exportar la matriz de pixeles a un archivo CSV**, lo cual permite trasladar los resultados del procesamiento a otros entornos de análisis como Python, R o incluso Excel. Esta funcionalidad representa un puente entre la teoría de la representación digital de imágenes y el análisis aplicado de datos, reforzando la interdisciplinariedad del aprendizaje. La práctica no se limita entonces a la manipulación visual de la imagen, sino que también prepara el terreno para la explotación analítica de la información contenida en ella.

En conclusión, la introducción al procesamiento digital de imágenes a través de la lectura, separación de componentes RGB, conversión a escala de grises y manipulación de brillo y contraste, constituye un ejercicio integral que combina teoría, programación y aplicación práctica. Este enfoque pedagógico, inspirado en los ejemplos de Java y trasladado a C++/GTK, ofrece al estudiante universitario una experiencia que lo prepara tanto para comprender los fundamentos matemáticos de la computación gráfica como para desarrollar herramientas aplicadas en el ámbito profesional.

Desarrollo

La práctica consistió en implementar en C++ con GTK+3 una aplicación capaz de leer imágenes, separar sus componentes RGB, convertirlas a escala de grises, ajustar brillo y contraste de forma interactiva y exportar los datos a CSV. A continuación se describe lo realizado, apoyándonos en los bloques de código más relevantes.

Lectura de imágenes

Para cargar una imagen desde el sistema de archivos se empleó la función `gdk_pixbuf_new_from_file()`, que genera un objeto `GdkPixbuf` con acceso a la información de los pixeles:

```
original_pixbuf = gdk_pixbuf_new_from_file(filename, &error);
int width  = gdk_pixbuf_get_width(original_pixbuf);
int height = gdk_pixbuf_get_height(original_pixbuf);
guchar *pixels = gdk_pixbuf_get_pixels(original_pixbuf);
```

Este bloque nos permitió acceder a las dimensiones de la imagen y a su arreglo de pixeles en memoria, de manera similar a como en Java se utilizaba `BufferedImage`.

Separación de componentes RGB

Siguiendo la lógica presentada en los materiales de clase, se recorrió la matriz de pixeles y se construyeron tres imágenes: rojo, verde y azul. En C++ esto se implementó de la siguiente forma:

```
guchar r = p[0];  // componente rojo
guchar g = p[1];  // componente verde
guchar b = p[2];  // componente azul

// Construcción de imágenes separadas
guchar *red_p   = red_pixels   + y * red_rowstride   + x * 3;
red_p[0] = r; red_p[1] = 0; red_p[2] = 0;

guchar *blue_p  = blue_pixels  + y * blue_rowstride  + x * 3;
blue_p[0] = 0; blue_p[1] = 0; blue_p[2] = b;

guchar *green_p = green_pixels + y * green_rowstride + x * 3;
green_p[0] = 0; green_p[1] = g; green_p[2] = 0;
```

Cada canal se almacena en un `GdkPixbuf` distinto y se despliega en la interfaz gráfica, permitiendo observar visualmente cómo se descompone la imagen original.

Conversión a escala de grises

Se creó también una versión en escala de grises mediante el promedio simple de los tres componentes:

```
guchar gray_value = (guchar)((r + g + b) / 3);
guchar *gray_p = gray_pixels + y * gray_rowstride + x * 3;
gray_p[0] = gray_value;
gray_p[1] = gray_value;
gray_p[2] = gray_value;
```

Este cálculo corresponde al modo `GRAY_8U` de la API interna tipo Java implementada en la práctica.

Ajuste de brillo y contraste

Para añadir interactividad, se incorporaron dos sliders que modifican el brillo y el contraste. El ajuste se aplica directamente a cada pixel con la siguiente lógica:

```
double bright   = v + brightness_value;
double scaled   = bright * contrast_value;
if (scaled < 0.0)   scaled = 0.0;
if (scaled > 255.0) scaled = 255.0;
d[c] = static_cast<guchar>(scaled);
```

Ajuste en la imagen original (brillo/contraste)

Se aplica sobre la copia base escalada en color (`base_color_pixbuf`), preservando alfa si existe:

```
void updateColorImage() {
    if (!base_color_pixbuf) return;
    int width      = gdk_pixbuf_get_width(base_color_pixbuf);
    int height     = gdk_pixbuf_get_height(base_color_pixbuf);
    gboolean alpha = gdk_pixbuf_get_has_alpha(base_color_pixbuf);
    int channels   = gdk_pixbuf_get_n_channels(base_color_pixbuf);
    int src_rs     = gdk_pixbuf_get_rowstride(base_color_pixbuf);
    guchar *src_px = gdk_pixbuf_get_pixels(base_color_pixbuf);

    GdkPixbuf *adj = gdk_pixbuf_new(GDK_COLORSPACE_RGB, alpha, 8, width, height);
    int dst_rs     = gdk_pixbuf_get_rowstride(adj);
    guchar *dst_px = gdk_pixbuf_get_pixels(adj);

    for (int y = 0; y < height; ++y) {
        for (int x = 0; x < width; ++x) {
            guchar *s = src_px + y * src_rs + x * channels;
            guchar *d = dst_px + y * dst_rs + x * channels;
            for (int c = 0; c < 3; ++c) {
                double v  = s[c];
                double br = v + brightness_value;
                double sc = br * contrast_value;
                if (sc < 0.0)   sc = 0.0;
                if (sc > 255.0) sc = 255.0;
                d[c] = static_cast<guchar>(sc);
            }
            if (alpha && channels == 4) d[3] = s[3];
        }
    }
    gtk_image_set_from_pixbuf(GTK_IMAGE(images[1]), adj);
    g_object_unref(adj);
}
```

Explicación: por cada píxel se calcula `v' = clamp((v + brillo) * contraste, 0, 255)` para R, G y B; si la imagen tiene alfa, se copia intacto al resultado.

Ajuste en escala de grises (brillo/contraste)

Se usa la base de grises precomputada (`base_grayscale_pixbuf`) y se renderiza el resultado:

```cpp
void updateGrayscaleImage() {
    if (!base_grayscale_pixbuf) return;
    int width  = gdk_pixbuf_get_width(base_grayscale_pixbuf);
    int height = gdk_pixbuf_get_height(base_grayscale_pixbuf);

    GdkPixbuf *adjusted_pixbuf = gdk_pixbuf_new(GDK_COLORSPACE_RGB, FALSE, 8, width, height);
    guchar *base_pixels = gdk_pixbuf_get_pixels(base_grayscale_pixbuf);
    guchar *adj_pixels  = gdk_pixbuf_get_pixels(adjusted_pixbuf);
    int base_rowstride  = gdk_pixbuf_get_rowstride(base_grayscale_pixbuf);
    int adj_rowstride   = gdk_pixbuf_get_rowstride(adjusted_pixbuf);

    for (int y = 0; y < height; y++) {
        for (int x = 0; x < width; x++) {
            guchar *base_p = base_pixels + y * base_rowstride + x * 3;
            guchar *adj_p  = adj_pixels  + y * adj_rowstride  + x * 3;
            double original = base_p[0];
            double bright   = original + brightness_value;
            double scaled   = bright * contrast_value;
            if (scaled < 0.0)   scaled = 0.0;
            if (scaled > 255.0) scaled = 255.0;
            guchar v = static_cast<guchar>(scaled);
            adj_p[0] = v; adj_p[1] = v; adj_p[2] = v;
        }
    }
    gtk_image_set_from_pixbuf(GTK_IMAGE(images[5]), adjusted_pixbuf);
    g_object_unref(adjusted_pixbuf);
}
```

Explicación: se toma el valor de gris (R=G=B) por píxel y se aplica la misma fórmula lineal de brillo/contraste.

Extracción de canales RGB y generación de base en grises

Bajo la lógica de máscaras/corrimientos, se separan planos R, G, B y se construye una base de grises:

```
void extractChannelsWith24BitManipulation(GdkPixbuf *pixbuf, int width, int height) {
    if (!pixbuf) return;
    int channels = gdk_pixbuf_get_n_channels(pixbuf);
    int rowstride = gdk_pixbuf_get_rowstride(pixbuf);
    guchar *pixels = gdk_pixbuf_get_pixels(pixbuf);
    if (channels < 3) {
        g_print("La imagen debe tener al menos 3 canales RGB\n");
        return;
    }

    GdkPixbuf *red_pixbuf   = gdk_pixbuf_new(GDK_COLORSPACE_RGB, FALSE, 8, width, height);
    GdkPixbuf *blue_pixbuf  = gdk_pixbuf_new(GDK_COLORSPACE_RGB, FALSE, 8, width, height);
    GdkPixbuf *green_pixbuf = gdk_pixbuf_new(GDK_COLORSPACE_RGB, FALSE, 8, width, height);
    base_grayscale_pixbuf   = gdk_pixbuf_new(GDK_COLORSPACE_RGB, FALSE, 8, width, height);

    guchar *red_pixels   = gdk_pixbuf_get_pixels(red_pixbuf);
    guchar *blue_pixels  = gdk_pixbuf_get_pixels(blue_pixbuf);
    guchar *green_pixels = gdk_pixbuf_get_pixels(green_pixbuf);
    guchar *gray_pixels  = gdk_pixbuf_get_pixels(base_grayscale_pixbuf);

    int red_rowstride   = gdk_pixbuf_get_rowstride(red_pixbuf);
    int blue_rowstride  = gdk_pixbuf_get_rowstride(blue_pixbuf);
    int green_rowstride = gdk_pixbuf_get_rowstride(green_pixbuf);
    int gray_rowstride  = gdk_pixbuf_get_rowstride(base_grayscale_pixbuf);
```

```
    for (int y = 0; y < height; y++) {
        for (int x = 0; x < width; x++) {
            guchar *p = pixels + y * rowstride + x * channels;
            guchar r = p[0], g = p[1], b = p[2];
            uint32_t pixel_data_b = ((uint32_t)b) & 0xFF;
            uint32_t pixel_data_g = (((uint32_t)g) << 8) & 0xFF00;
            uint32_t pixel_data_r = (((uint32_t)r) << 16) & 0xFF0000;
            guchar fB = (guchar)(pixel_data_b);
            guchar fG = (guchar)(pixel_data_g >> 8);
            guchar fR = (guchar)(pixel_data_r >> 16);

            guchar *red_p   = red_pixels   + y * red_rowstride   + x * 3;
            red_p[0] = fR; red_p[1] = 0;   red_p[2] = 0;

            guchar *blue_p  = blue_pixels  + y * blue_rowstride  + x * 3;
            blue_p[0] = 0;  blue_p[1] = 0; blue_p[2] = fB;

            guchar *green_p = green_pixels + y * green_rowstride + x * 3;
            green_p[0] = 0; green_p[1] = fG; green_p[2] = 0;

            guchar gray_value = (guchar)((fR + fG + fB) / 3);
            guchar *gray_p = gray_pixels + y * gray_rowstride + x * 3;
            gray_p[0] = gray_value; gray_p[1] = gray_value; gray_p[2] = gray_value;
        }
    }

    gtk_image_set_from_pixbuf(GTK_IMAGE(images[2]), red_pixbuf);
    gtk_image_set_from_pixbuf(GTK_IMAGE(images[3]), blue_pixbuf);
    gtk_image_set_from_pixbuf(GTK_IMAGE(images[4]), green_pixbuf);

    g_object_unref(red_pixbuf);
    g_object_unref(blue_pixbuf);
    g_object_unref(green_pixbuf);
}
```

Explicación: se separan los canales y se prepara `base_grayscale_pixbuf` (promedio simple). Esto alimenta los paneles R, G, B y la base para los ajustes de grises.

Selector de modo para exportación y mapeo a la API

El combo del menú define el modo de exportación; `comboToMode()` traduce la selección:

```
IntImageMode comboToMode() const {
    int idx = gtk_combo_box_get_active(GTK_COMBO_BOX(mode_combo));
    switch (idx) {
        case 0: return IntImageMode::COLOR_RGB_PACKED;
        case 1: return IntImageMode::RED_8U;
        case 2: return IntImageMode::GREEN_8U;
        case 3: return IntImageMode::BLUE_8U;
        case 4: return IntImageMode::GRAY_8U;
        default: return IntImageMode::COLOR_RGB_PACKED;
    }
}
```

Explicación: se mapea 1:1 la opción del combo a los modos de salida para el CSV.

API para obtener matrices/vecores

Matriz 2D por modo (incluye empaquetado `0x00RRGGBB`) y conversión a vector 1D:

```cpp
std::vector<std::vector<uint32_t>> getImagenInt(IntImageMode mode) const {
    if (!base_color_pixbuf) return {};
    const int width      = gdk_pixbuf_get_width(base_color_pixbuf);
    const int height     = gdk_pixbuf_get_height(base_color_pixbuf);
    const int channels   = gdk_pixbuf_get_n_channels(base_color_pixbuf);
    const int rowstride  = gdk_pixbuf_get_rowstride(base_color_pixbuf);
    const gboolean alpha = gdk_pixbuf_get_has_alpha(base_color_pixbuf);
    const guchar* pixels = gdk_pixbuf_get_pixels(base_color_pixbuf);
    (void)alpha;

    std::vector<std::vector<uint32_t>> out(height, std::vector<uint32_t>(width, 0));
    for (int y = 0; y < height; ++y) {
        const guchar* row = pixels + y * rowstride;
        for (int x = 0; x < width; ++x) {
            const guchar* p = row + x * channels;
            const uint32_t r = p[0];
            const uint32_t g = p[1];
            const uint32_t b = p[2];
            switch (mode) {
                case IntImageMode::COLOR_RGB_PACKED:
                    out[y][x] = (r << 16) | (g << 8) | (b);
                    break;
                case IntImageMode::RED_8U:   out[y][x] = r; break;
                case IntImageMode::GREEN_8U: out[y][x] = g; break;
                case IntImageMode::BLUE_8U:  out[y][x] = b; break;
                case IntImageMode::GRAY_8U:  out[y][x] = (r + g + b) / 3; break;
            }
        }
    }
    return out;
}
```

```cpp
static std::vector<uint32_t> convertirInt2DA1D(const std::vector<std::vector<uint32_t>>& m) {
    if (m.empty()) return {};
    std::vector<uint32_t> v;
    v.reserve(m.size() * m[0].size());
    for (const auto& row : m) v.insert(v.end(), row.begin(), row.end());
    return v;
}
```

Explicación: reproduce el patrón del material en Java: obtener una **matriz** por canal o en **color empaquetado** y convertirla a **vector** cuando se requiera.

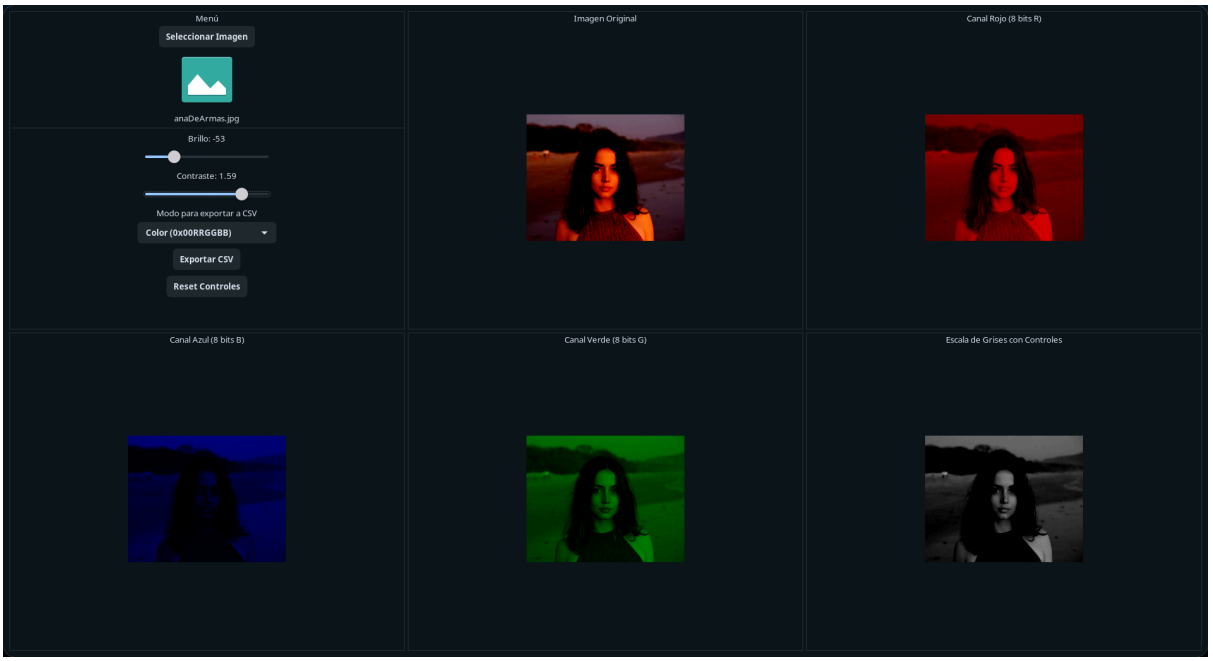
Exportación a CSV

El diálogo de guardado se dispara con el botón “Exportar CSV” y llama a `exportCSV()`:

```cpp
bool exportCSV(IntImageMode mode, const char* path) {
    auto M = getImagenInt(mode);
    if (M.empty()) return false;

    std::ofstream ofs(path);
    if (!ofs.is_open()) return false;

    const size_t H = M.size();
    const size_t W = M[0].size();

    for (size_t y = 0; y < H; ++y) {
        for (size_t x = 0; x < W; ++x) {
            ofs << M[y][x];
            if (x + 1 < W) ofs << ',';
        }
        ofs << '\n';
    }
    ofs.close();
    return true;
}
```

Explicación: se vuelca la **matriz 2D** (una fila por renglón de la imagen escalada) en formato CSV decimal; queda lista para análisis en Python/R/Excel. Si luego quieres versión **hex** para `COLOR_RGB_PACKED`, la adaptamos fácil.

Resultado



En la figura se observa la interfaz gráfica completa del sistema desarrollado en C++ con GTK. El panel **Menú**, ubicado en la parte superior izquierda, contiene los controles principales: el botón *Seleccionar Imagen* para cargar un archivo, un ícono representativo que se actualiza al seleccionar la imagen, los deslizadores de **Brillo** y **Contraste**, el selector de **Modo para exportar a CSV** y los botones *Exportar CSV* y *Reset Controles*.

En los demás recuadros se despliegan las diferentes representaciones de la imagen:

- **Imagen Original**, la cual se actualiza dinámicamente con los cambios de brillo y contraste.

- **Canal Rojo**, **Canal Verde** y **Canal Azul**, que muestran de forma independiente cada componente del modelo RGB.

- **Escala de Grises con Controles**, que además de la conversión a gris refleja en tiempo real los ajustes realizados con los sliders.

Enlace al proyecto

https://github.com/elsrdelanoche/imageAnalysis/tree/main/practica_1

Conclusiones

La realización de esta práctica me permitió no solo comprender la teoría detrás de la lectura y separación de imágenes en el modelo RGB, sino también vivir la experiencia de implementarla en un lenguaje diferente al de los ejemplos proporcionados en clase. Mientras que en los materiales de apoyo se mostró cómo hacerlo en Java y operaciones de enmascaramiento, al trasladar la lógica a C++ con GTK+ descubrí la importancia de la adaptabilidad y la transferencia de conocimiento entre diferentes entornos de programación. Esta experiencia refuerza la idea de que los principios fundamentales de la informática, en este caso, la representación binaria del color y la manipulación de bits que son universales y pueden aplicarse en distintos lenguajes y librerías.

Un aprendizaje clave fue constatar cómo las operaciones aparentemente abstractas, como los corrimientos de bits o las máscaras lógicas, tienen un impacto directo y visible en la manipulación de imágenes. Al implementar las funciones para separar los canales rojo, verde y azul, comprendí con mayor claridad la relación entre la representación numérica de los pixeles y el resultado gráfico que se observa en la pantalla. Esto me ayudó a afianzar mi dominio en temas que muchas veces parecen teóricos, como la aritmética de enteros y el uso de operadores bit a bit, al verlos materializados en transformaciones visuales inmediatas.

La incorporación de controles para modificar brillo y contraste en tiempo real añadió un componente de interactividad muy valioso. A través de los sliders, pude experimentar cómo pequeñas variaciones en los valores de cada pixel alteran la percepción visual de la imagen. Esto me permitió reflexionar sobre el papel del preprocesamiento en aplicaciones más complejas, como el reconocimiento de imágenes o el entrenamiento de modelos de

machine learning, donde ajustes simples de brillo y contraste pueden marcar la diferencia en la calidad de los datos de entrada.

En el plano académico, la práctica me brindó una oportunidad para consolidar habilidades en programación gráfica, manejo de librerías externas y diseño de interfaces. La experiencia de trabajar con GTK+ me permitió entender cómo se estructura un proyecto que combina lógica de procesamiento con elementos de interacción visual. Desde el manejo de eventos en los botones hasta la actualización dinámica de los pixbufs, cada paso supuso un reto que me acercó más a la realidad del desarrollo de software profesional.

Revisité conceptos básicos de la representación digital de imágenes y el modelo RGB; por otro, construí una herramienta funcional capaz de cargar imágenes, separarlas en sus canales, ajustar sus propiedades visuales y exportar datos para análisis posterior. Esta integración de saberes me motiva a seguir explorando el campo del procesamiento digital de imágenes y me deja preparado para abordar problemas más complejos en visión por computadora, inteligencia artificial y análisis de datos visuales.

Referencias

- Material de clase: **Lectura de imágenes mediante un lenguaje de programación**.
- Material de clase: **Separación de componentes de frecuencia según el modelo RGB**.
- Documentación oficial GTK: *GdkPixbuf Reference Manual*.
- Gonzalez, R. C., & Woods, R. E. (2018). *Digital Image Processing*. Pearson.