Nombre: Giancarlo Vega Marín

Carné: 2020195338

Proyecto Individual

Desarrollo de una aplicación para la generación de gráficos y texto

Documentación de Diseño

Introducción

El proyecto consistía en desarrollar un programa en ensamblador, que realizara las operaciones matemáticas requeridas para una interpolación bilineal de imágenes. El programa obtenía el cuadrante 100x100 de una imagen 400x400 .jpg convertida a .img, realizaba las operaciones necesarias de los pixeles y devolvía el cuadrante interpolado en otro .img. El lenguaje de alto nivel se encargaba de obtener el cuadrante de la imagen, así como la visualización de ambas imágenes, la conversión de .jpg a .img y viceversa, y la transformación de la imagen a escala de grises. Toda la interpolación era hecha por el código en ensamblador.

Requerimientos funcionales

- **RF01 - Lectura de imágenes**: El sistema debe ser capaz de cargar una imagen en escala de grises con resolución fija de **400x400 píxeles**.
- **RF02 - División en cuadrantes**: La imagen debe ser dividida correctamente en **16 cuadrantes de 100x100 píxeles** cada uno, y se debe visualizar la división de cuadrantes en la imagen para saber cuál se va a interpolar.
- **RF03 - Exportación de cuadrante**: El cuadrante seleccionado debe exportarse a un archivo binario (.img) en un formato plano y sin encabezado.
- **RF04 - Interpolación bilineal**: El programa ensamblador (x86, ARM o RISCV solamente) debe escalar el cuadrante seleccionado a **400x400 píxeles** usando interpolación bilineal precisa.

- **RF05 - Escritura de salida**: El resultado interpolado debe escribirse en un nuevo archivo .img, que luego puede visualizarse como imagen (.jpg) en el lenguaje de alto nivel.
- **RF06 - Interfaz gráfica**: Se debe crear una interfaz gráfica que permita al usuario visualizar la imagen en escala de grises, escoger el cuadrante deseado y visualizar el resultado de la interpolación
- **RF07** – **Framework integrado:** El enlace y la compilación del código ensamblador debe de hacerse de forma automática por medio del lenguaje de alto nivel, sin necesidad de abrir un terminal y escribir los comandos.

Requerimientos técnicos

- **RT01 - Lenguaje ensamblador x86-64**: La interpolación debe implementarse en arquitectura ISA para **x86-64**, sin uso de bibliotecas externas.
- **RT02 - Precisión en interpolación**: Se debe implementar correctamente el algoritmo de interpolación bilineal, considerando bordes, redondeos y límites de píxeles.
- **RT03 - Manejo de archivos binarios**: El programa debe trabajar con archivos .img en formato binario sin cabecera, lo que implica manejo manual de lectura y escritura.
- **RT04 – Sistema Operativo:** El sistema debe ser ejecutable en sistemas Linux.

Estándares, normas y herramientas

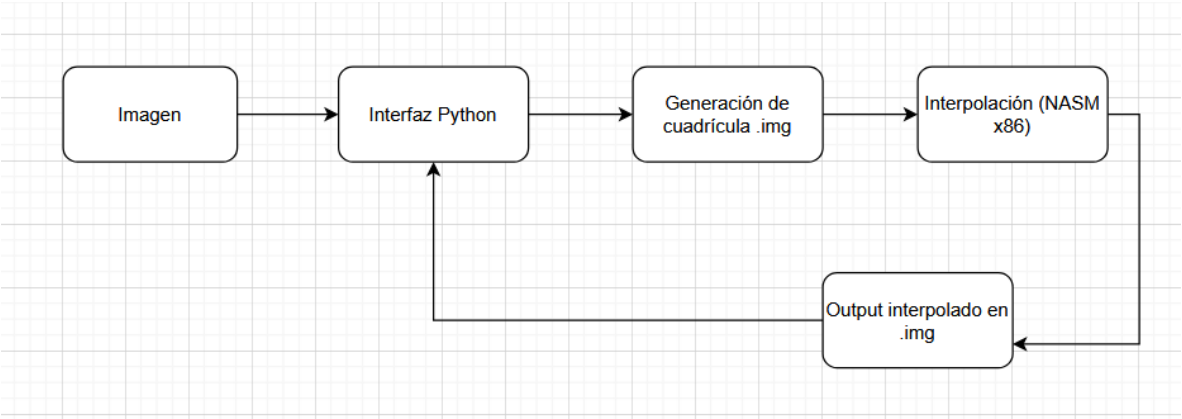
- Se deben usar formatos simples y documentados (.img como arreglo plano, .jpg para visualización), promoviendo portabilidad.
- El diseño debe ser documentado conforme a las guías de ingeniería del curso (estructura del ISA, claridad, justificación técnica)
- Control de versiones usando Git, siguiendo un flujo de ramas: master, develop y ramas de trabajo.

Soluciones

Para la evaluación y comparación de dos soluciones, se plantearán soluciones basadas en la escogencia del ISA para el desarrollo del código ensamblador, el cual realiza la interpolación del cuadrante seleccionado. Para ambas soluciones, se utiliza Python como el lenguaje de alto nivel.

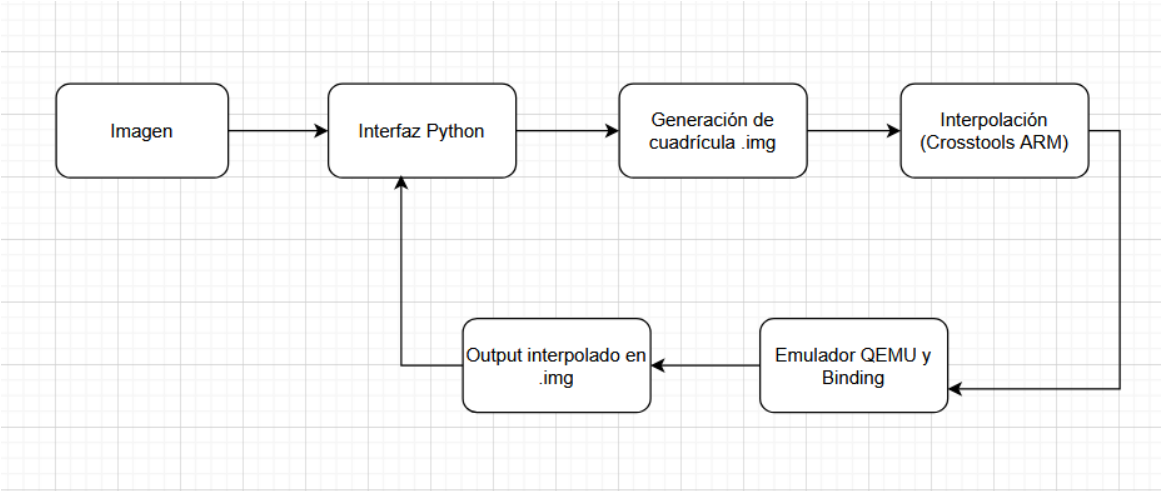
Solución 1: Arquitectura x86

Se plantea utilizar la arquitectura x86, para la elaboración del código ensamblador. Se utilizaría el ensamblador NASM para la escritura del código, en el sistema operativo Ubuntu Linux 24.04. El lenguaje de alto nivel Python, utiliza la librería "subprocess" realiza la compilación, enlace y ejecución de forma automática. Como herramienta de simulación y debugging, se utiliza GDB para comprobar el estado de los registros y de memoria.



Solución 2: Arquitectura ARM

Se plantea el uso de la arquitectura ARM, para la elaboración del código ensamblador. Usando el mismo sistema operativo, se opta por la herramienta CrossTools para la elaboración del código. En este caso, la librería de Python "subprocess" no realizaría todo el proceso de forma automática, debido a que el binario tiene una arquitectura diferente al sistema operativo. Por lo tanto, se deben emplear herramientas adicionales como QEMU para su ejecución, y también su simulación y debugging.



Comparación de soluciones

Comparando las dos soluciones propuestas, se pueden encontrar ventajas y limitaciones para ambas soluciones. Lo cual se va a analizar para encontrar la mejor solución para el proyecto.

Solución 1:

Entre las ventajas de utilizar x86 como la arquitectura para el ensamblador, se pueden numerar:

- Acceso a registros de propósito general grandes (RAX, RBX de 64 bits), ideales para operaciones matemáticas y dirección de memoria en imágenes grandes
- La arquitectura x86 está ampliamente documentada, con muchos ejemplos, artículos, videos y guías disponibles para cualquiera que desee aprender
- GDB es herramientas muy útil y robusta, con amplio soporte para inspeccionar registros y memoria, facilitando el proceso de depuración y pruebas del código.
- Al ser la misma arquitectura que el sistema operativo, el archivo .asm compilado se ejecuta fácilmente, sin necesidad de un simulador adicional para que funcione.

Entre las desventajas y limitaciones, se encuentran las siguientes:

- NASM requiere mayor detalle explícito en instrucciones, lo cual puede ser difícil de programar y comprender para alguien que no haya trabajado con esta arquitectura antes
- Las instrucciones son menos intuitivas que otras arquitecturas como ARM. Para un ingeniero sin conocimiento previo, puede ser difícil comprender en su totalidad los registros de propósito general, y los registros utilizados para el stack y queue.
- Aunque no específicamente una desventaja en el contexto de este proyecto, la arquitectura x86 consume mucha más energía a diferencia de las arquitecturas RISC.

Solución 2:

Las ventajas de optar por la arquitectura ARM, entre muchas de ellas, son las siguientes:

- ARM tiene instrucciones eficientes para manipulaciones de datos en memoria alineada, útil para procesamiento de imágenes.
- La arquitectura ARM es más fácil de aprender y entender, debido a su mayor simpleza en sintaxis, código y registros.
- Instrucciones más predecibles, estilo RISC, con menor variabilidad y ambigüedad.
- Eficiencia energética, ideal para dispositivos móvileso embebidos.
- Soporte amplio en hardware moderno (smartphones,IoT).

A su vez, presentan ciertas desventajas:

- Requiere entorno cruzado, herramientas de “cross-plantform” y emuladores adicionales debido a la diferencia en arquitectura en comparación con el sistema operativo, lo cual implica más líneas de código, y un proceso menos automático
- A diferencia de x86, se requiere más pasos para la depuración y simulación. En la primera solución, simplemente se suaba GDB y ya uno podía ver el flujo del código, los registros y la memoria. En cambio, con ARM se debe utilizar dos terminales, conectar a un puerto local, y usar

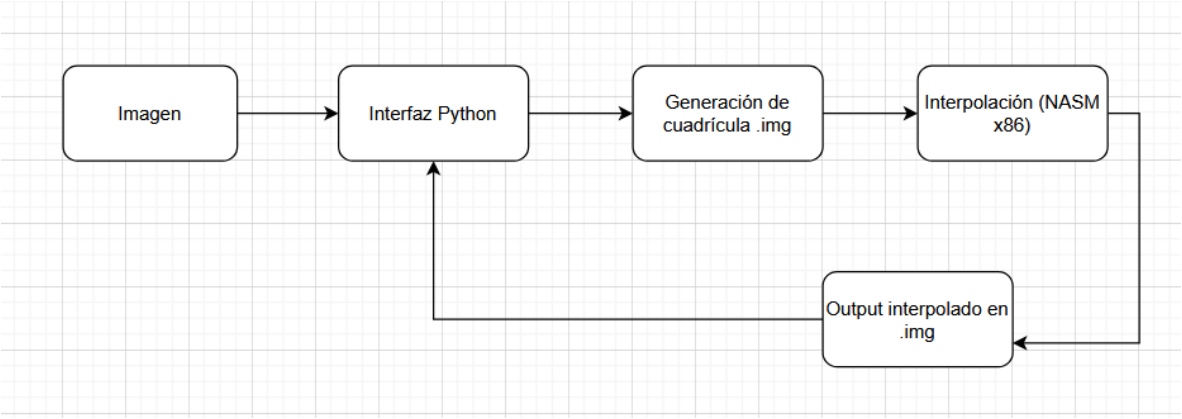
- Citando la primera desventaja, el framework integrado automático se ve perjudicado, lo cual puede causar problemas con los requerimientos establecidos por el profesor.

Solución Final

Comparando las ventajas y desventajas de ambas soluciones, y tomando cuenta el contexto como estudiante y las especificaciones del proyecto, se decide optar por la Solución 1, el cual es utilizar x86 como arquitectura para el ensamblador.

Las herramientas NASM y GDB serán muy útiles a la hora de la compilación y simulación, ya que el código para realizar la interpolación va a ser un código grande, donde las fallas y operaciones incorrectas son muy probables a suceder. Con estas herramientas, se desea un proceso menos tedioso para el estudiante, y más intuitivo para el profesor.

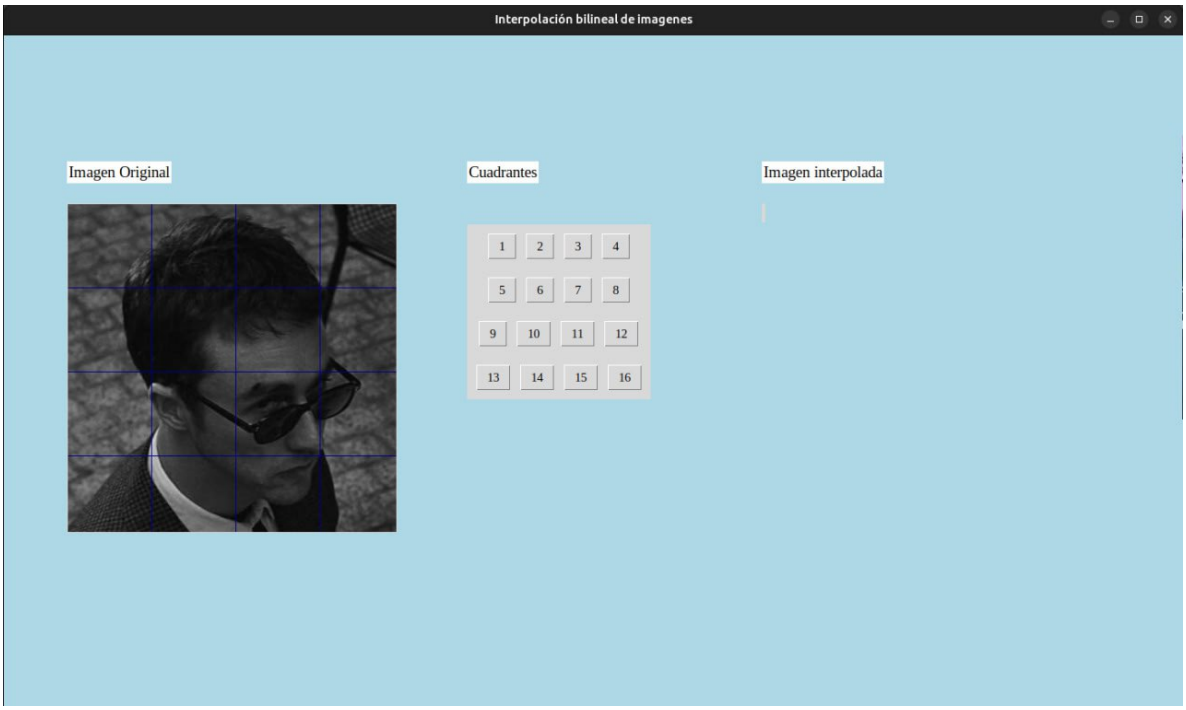
Además, la arquitectura permitirá que el proceso automático sea mucho más sencillo de implementar, utilizando las librerías os y subprocess en Python, el cual no era posible si se escogía la segunda solución.



Como se había mencionado antes, el lenguaje de alto nivel será Python. Sus bibliotecas, sus herramientas y su extensa documentación facilitarán el proceso para la visualización de imágenes, la generación de cuadrantes y la conversión de formato de imágenes. Usando un IDE como Pycharm, se pueden instalar las librerías fácilmente en el intérprete, aunque igualmente se puede compilar el código por medio de terminal.

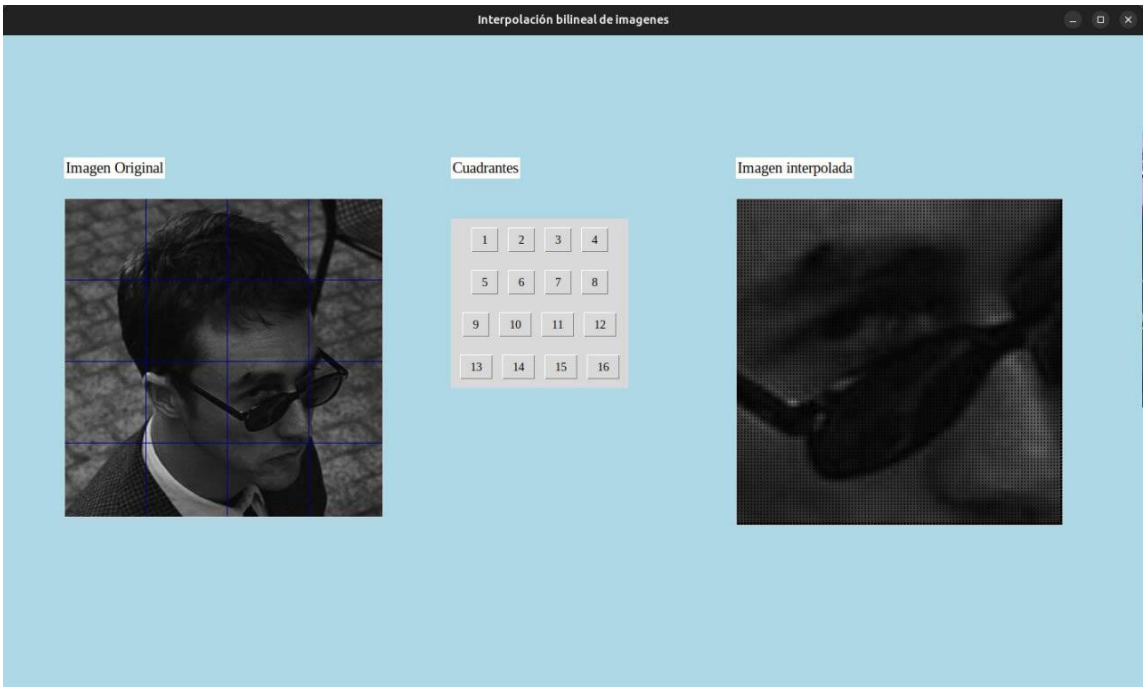
El código de Python toma la imagen inicial, y la convierte a escala de grises. Esto con el fin de poder usar cualquier imagen, sin necesidad de buscar herramientas online para hacer la conversión previamente.

En la imagen se ve el paso anteriormente mencionado.



Basándose en la cuadricula transparente encima de la imagen, y los botones que representan dichos cuadrantes, se escoge el cuadrante que se desea interpolar. Este proceso es automático, por lo cual no se debe abrir la terminal de Linux

El código ensamblador recibe el cuadrante en formato .img, y comienza a realizar las operaciones. Abre el archivo, guarda los pixeles en un buffer, obtiene los pixeles originales, y usando esos valores, realiza las interpolaciones verticales y horizontales. Después de eso, usando los valores recientemente interpolados, se obtienen las interpolaciones intermedias. Finalmente, se escriben los nuevos pixeles en un nuevo archivo .img, y el código de Python se encarga de convertir dicho archivo para su visualización



Referencias

Navas, M. (26 de noviembre, 2017). *Procesadores x86 vs ARM: diferencias y ventajas principales***.** ProfessionalReview. https://www.profesionalreview.com/2017/11/26/procesadores-x86-vs-arm-diferencias-ventajas-principales/

Moez, A. (1 de octubre, 2024). *An Introduction to Python Subprocess: Basics and Examples*. Datacamp. https://www.datacamp.com/tutorial/python-subprocess

Ramos, P. (28 de enero, 2014). *Instrucciones, registros y operadores en x86.* Welivesecurity. https://www.welivesecurity.com/la-es/2014/01/28/instrucciones-registros-operadores-x86/

NASM Tutorial. (s,f). LMU. https://cs.lmu.edu/~ray/notes/nasmtutorial/?ref=linuxtldr.com

https://android.googlesource.com/platform/prebuilts/gcc/linux-x86/host/i686-linux-glibc2.7-4.6/+/refs/heads/jb-mr1-dev/sysroot/usr/include/asm/unistd_32.h

https://android.googlesource.com/platform/prebuilts/gcc/linux-x86/host/i686-linux-glibc2.7-4.6/+/refs/heads/jb-mr1-dev/sysroot/usr/include/asm/unistd_64.h