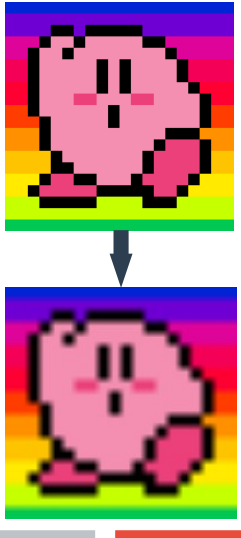
IIC2523 – Ayudantía de Go

Obejtivo

Crear un programa que lea una imagen y le aplique un filtro de difuminación usando threads.

1. **Leer Input**
   a) Flags
   b) Argumento línea de comandos
   c) Standard Input (STDIN)
2. **Aplicar Difuminación**
   a) Calcular difuminado para 1 pixel
   b) Manejar casos fuera del borde
3. **Concurrencia**
   a) Crear GoRoutines
   b) Crear canal de comunicación
4. **Guardar Nueva Imagen**



Detalles del problema

Para cada pixel queremos **calcular el promedio de color** usando los vecinos dentro de la máscara de tamaño n por m

Entradas del programa

1.Flag: número de threads

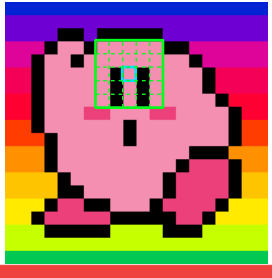
2.Argumento: ruta foto .png

3.STDIN: Tamaño máscara

La máscara debe tener **dimensiones impares**

Comando:

```
./blur.out –threads 5 image.png
```



Máscara de 5 por 5 aplicada al pixel seleccionado

Detalles del problema

Ejemplo:

`p={x:9, x:5}` → `{R:244, G:143, B:176}`

Vecinos

`v={R:0, G:0, B:0}`, son 6 (color negro)

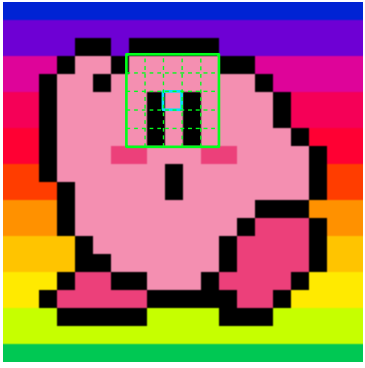
`v=p`, son 18 (mismo color)

Nuevo valor:

`R:=(6*0+19*244)/25` → `185.44` → `185`

`G:=(6*0+19*143)/25` → `108.68` → `109`

`B:=(6*0+19*179)/25` → `133.76` → `134`



Paso 0: Importar Librerías

Definir que el archivo es un ejecutable con el paquete `main`

Usar función `Import` para importar librerías

- Separadas por salto de línea
- Nombre entre comillas dobles

Creamos nuestro `main`

Paso 1: Leer Input

Leer flag de threads

- Librería `flag` lee el nombre “threads” y lo interpreta como `int`
- Devuelve un puntero

Leer `path` del archivo

- Usamos la librería `os` para la función `Args()`
- Devuelve un arreglo

```
[./blur.out, -threads, 5, image.png]
```

  - Leemos `Args[3]`

Paso 1: Leer Input

Leer máscara desde STDIN

- Creamos un lector de STDIN
- Leemos ambas entradas como strings.
- `reader.ReadString('\n')` lee hasta el separador y devuelve un `string` que incluye dicho separador
- Eliminamos el separador y convertimos el `string` a `int`.

Creamos un `struct` para guardar la máscara

Paso 1: Leer Input

Creamos la máscara

Leemos la Imagen

- Registramos el formato .png
- Abrimos el archivo usando `os.Open()`
- Decodificamos la imagen con `png.Decode()`

Paso 2: Aplicar Difuminado

Creamos función de difuminado

- Argumentos: imagen, máscara, coordenadas de un pixel
- Devuelve el color resultante de tipo `RGBA`
- Creamos un bucle doble que revisa los vecinos dentro de la máscara

Pero, que pasa con casos como `(0, 0)`?

La máscara se sale de los límites

Paso 2: Aplicar Difuminado

Manejar casos fuera de los límites:

- Opción uno: ignorar pixeles fuera de los límites → promedio debe considerar menos elementos
- Opción dos: extender el color de los límites → tomar el color del pixel válido más cercano

En este caso, seguimos la opción dos y creamos la función que maneja estos casos

Paso 2: Aplicar Difuminado

Terminamos la función de difuminado y devolvemos el nuevo color

Tener ojo con los cambios de tipo de datos

- `RGBA()` devuelve `uint32`
- La imagen usa `uint8`
- Hacemos el paso de `uint32` a `uint8` dividiendo por 257

Paso 3: Concurrencia

Necesitamos crear un objeto Image vacío

- Creamos una nueva imagen
- Usamos dimensiones (`Bounds`) de la imagen de entrada
- Usamos `Set()` para colorear un pixel

Nuestros threads van a escribir sobre este objeto

Paso 3: Concurrencia

GoRoutines

- Son threads livianos de Go
- Para iniciar una `GoRoutine` debemos anteponer "`go`" a una función

```
go f(x, y, z)
```

Creamos la función WritePixel

- Lee un Pixel desde un **`channel`** llamado `pix`
- Calcula el nuevo color
- Escribe en la nueva imagen

Importante: `range` lee del canal hasta que éste sea cerrado y luego, se termina el bucle

Paso 3: Concurrencia

Hacemos un bucle en la función main para crear las Goroutines

- Usamos el número de threads para crear una cantidad fija

Creamos un `struct` para guardar coordenadas de un pixel

Sólo nos falta el canal que guardará los pixeles que se deben difuminar

Paso 3: Concurrencia

Creamos un canal de tipo Pixel

Iteramos sobre la imagen; para cada par de coordenadas:

- Creamos un `Pixel`
- Lo enviamos al canal usando `<-`

Cerramos el canal usando `close()`

Paso 4: Guardar Nueva Imagen

Creamos un nuevo archivo

- Usamos `os.Create()` → entrega un objeto `File`

Codificamos el archivo

- Usamos `png.Encode()`
- Entregamos el objeto `File` y el objeto `Image`