RELATÓRIO

Nome: Henry Emanuel Leal Cagnini
Tema: Exercícios de Aplicação de Filtros

Enunciado: Crie um algoritmo que calcule a mediana dos tons de cinza de uma janela de NxN pixels, ao redor de um pixel (x,y). Inicie o processamento no canto inferior esquerdo da imagem e substitua todos os pixels da janela pelo valor da mediana. Após o processamento de um pixel, a janela deve mover-se N pixels para a direita. Ao atingir o lado direito da imagem, a janela deve mover-se N pixels para cima e reiniciar o processamento na margem esquerda a imagem.
Resposta: O algoritmo faz exatamente o que o enunciado solicita: varre a figura da esquerda para a direita, de baixo para cima, achando o tom de cinza mediano de cada janela, e substituindo todos os pixels da janela por este tom mediano. O algoritmo faz uso de uma imagem *buffer* para não influenciar nos valores de outras janelas. O algoritmo utiliza quicksort para achar o tom mediano de cinza.
Código desenvolvido:

```c
#define FILTER_SIDE 3
/**
 * Troca os valores entre duas variáveis.
 * @param a primeiro valor
 * @param b segundo valor
 */
void swap(unsigned char* a, unsigned char* b) {
  unsigned char t = *a;
  *a = *b;
  *b = t;
}
/**
 * Parte que faz propriamente a ordenação de valores.
 *
 * @param arr Array
 * @param low Menor índice desse array
 * @param high Maior índice desse array
 * @return
 */
unsigned char partition(unsigned char arr[], int low, int high) {
  unsigned char pivot = arr[high];
  int i = (low - 1);
  for (int j = low; j <= high - 1; j++) {
    if (arr[j] < pivot) {
      i++;
      swap(&arr[i], &arr[j]);
    }
  }
  swap(&arr[i + 1], &arr[high]);
  return (i + 1);
}
/**
 * Algoritmo principal do quicksort. Ordena um array in-place.
 *
 * @param arr Array de valores a serem ordenados.
 * @param low Menor índice desse array
 * @param high Maior índice desse array
 */
void quickSort(unsigned char arr[], int low, int high) {
  if (low < high) {
    int pi = partition(arr, low, high);
    quickSort(arr, low, pi - 1);
    quickSort(arr, pi + 1, high);
  }
}
```

```
/**
 * Pega (se possível) o valor grayscale de um pixel da imagem.
 *
 * @param i Posição no eixo x do pixel. Se exceder a imagem, o valor da borda
 *          na altura j será retornado no lugar.
 * @param j Posição no eixo y do pixel. Se exceder a imagem, o valor da borda
 *          na largura i será retornado no lugar.
 * @param img Imagem de onde o pixel será retirado
 * @return O valor (grayscale) do pixel na posição (i, j).
 */
unsigned char getGrayscalePixel(int i, int j, ImageClass img) {
  unsigned char r, g, b;
  if(i < 0) {
    i = 0;
  } else if(i >= img.SizeX()) {
    i = img.SizeX() - 1;
  }
  if(j < 0) {
    j = 0;
  } else if(j >= img.SizeY()) {
    j = img.SizeY() - 1;
  }
  img.ReadPixel((GLint)i, (GLint)j, r, g, b);
  unsigned char gray_pixel = (unsigned char)((0.3 * r) + (0.59 * g) + (0.11 * b));
  return gray_pixel;
}
/**
 * Captura uma determinada região da imagem.
 *
 * @param side Lado da máscara.
 * @param centerX Coordanada x do centro da máscara
 * @param centerY Coordanada y do centro da máscara
 * @param img Imagem
 * @return a máscara, com valores atualizados.
 */
unsigned char *getMask(int side, int centerX, int centerY, ImageClass img) {
  int min_x = centerX - (int)(side / 2);
  int max_x = centerX + (int)(side / 2);
  int min_y = centerY - (int)(side / 2);
  int max_y = centerY + (int)(side / 2);
  unsigned char *mask = new unsigned char [side * side];
  int counter = 0;
  for(int i = min_x; i <= max_x; i++) {
    for(int j = min_y; j <= max_y; j++) {
      mask[counter] = getGrayscalePixel(i, j, img);
      counter += 1;
    }
  }
  return mask;
}
/**
 * Pega o valor mediano de uma posição na imagem.
 *
 * @param side Lado da máscara
 * @param i Posição central no eixo x da máscara
 * @param j Posição central no eixo y da máscara
 * @param img Imagem
 * @return O valor mediano, quando aplicada uma máscara (de centro (i, j)) sobre uma imagem.
 */
unsigned char medianLapFilterToRegion(int side, int i, int j, ImageClass img) {
  unsigned char *mask = getMask(side, i, j, img);
  quickSort(mask, 0, side * side);
  if(((side * side) % 2) == 0) {
    return (mask[(side * side) / 2] + mask[((side * side) / 2) + 1]) / 2;
  }
  unsigned char new_pixel = mask[(side * side) / 2];
  delete[] mask;
  return new_pixel;
}
```

```
/**
* Aplica um filtro mediano sobre uma figura inteira. Pula de FILTER_SIDE em FILTER_SIDE
* Gera uma nova figura.
*
* @param img Imagem que será processada.
* @return A mesma imagem, agora pós passada do filtro mediano.
*/
ImageClass meanFilterLap(ImageClass img) {
  int length = FILTER_SIDE * FILTER_SIDE;
  ImageClass newImg = ImageClass(img.SizeX(), img.SizeY(), img.Channels());
  img.CopyTo(&newImg);
  int half_side = (int)FILTER_SIDE/2;
  for (int i = half_side; i < img.SizeX(); i += FILTER_SIDE) {
    for (int j = half_side; j < img.SizeY(); j += FILTER_SIDE) {
      unsigned char new_pixel = medianLapFilterToRegion(FILTER_SIDE, i, j, img);

      for(int k = (i - half_side); k <= (i + half_side); k++) {
        for(int l = (j - half_side); l <= (j + half_side); l++) {
          if(k < 0) {
            k = 0;
          } else if(k >= img.SizeX()) {
            k = img.SizeX() - 1;
          }
          if(l < 0) {
            l = 0;
          } else if(l >= img.SizeY()) {
            l = img.SizeY() - 1;
          }
//                    cout << "processed pixel (" << i << "," << j << ")" << endl;
          newImg.DrawPixel(k, l, new_pixel);
        }
      }
    }
  }
  return newImg;
}
```

Exemplos de Processamento:

Imagem processada com tamanho de janela = 3