Universidade Tecnológica Federal do Paraná – UTFPR, Campus Curitiba - Centro
Alunas: Amanda Jury Nakamura RA: 2582686
Ana Julia Molinos Leite da Silva RA: 2582694
Julia Kamilly de Oliveira RA: 2588005
Disciplina: ICSF13 – Fundamentos de Programação 1
Prof. Bogdan Tomoyuki Nassu, Profa. Leyza Baldo Dorini, Prof. Daniel Fernando Pigatto

Relatório - Projeto 03 – Wii sensor bar

O objetivo deste projeto foi criar uma função denominada `detectaSensorBar` que tem como parâmetros de entrada um ponteiro para uma imagem, e as coordenadas (X e Y) de 2 aglomerados de *pixels*. Ao final, essa função devia retornar o ângulo formado entre os 2 agrupamentos.

A maior parte deste projeto foi feita presencialmente com todas as integrantes do grupo na sala de aula e em encontros na biblioteca. A lógica foi desenvolvida a partir de conversas, desenhos e abstração conjunta. O código foi digitado pela Ana Julia, que o compartilhava e atualizava suas versões no google drive onde todas tínhamos acesso.

Ao ler o documento do projeto, a primeira dúvida foi em relação ao ângulo solicitado. Utilizamos a abstração e imaginamos os eixos X e Y e então traçamos uma reta entre o centro de cada aglomerado e obtivemos um ângulo formado entre o eixo X e essa reta. Também utilizamos dessa abstração para realizar o cálculo do ângulo através de cálculos trigonométricos.

Outros pontos considerados antes de se iniciar o código foram:

- Como encontrar os dois agrupamentos em meio aos ruídos gerados nas imagens?
- Após essa identificação, como definir os centros, dados que os agrupamentos não eram regulares?
- Sabemos onde encontrar ângulo, mas como calculá-lo?

A partir destes pontos, foram criadas funções adicionais para auxiliar na `detectaSensorBar.`

Função 00 - double detectaSensorBar (Imagem1C* img, Coordenada* l, Coordenada* r)

Função que contêm as demais. Chama a função `removeRuido`, `criaMatrizRotulos` e `defineDoisMaiores` para identificar os aglomerados desejados. A função `calculaCoordenadaCentro` é chamada duas vezes para identificar os centros de cada um dos aglomerados (direito e esquerdo). Além disso, é utilizada uma estrutura condicional ("*if*") para testar se a localização dos aglomerados está correta.

Ao final, desaloca a memória reservada para a matriz rotulo através da chamada da função `destroiMatrizRotulo.`

Função 01 - void removeRuido (Imagem1C* matriz)

A primeira função criada por nós tem o objetivo de remover os ruídos presentes na imagem. A função percorre toda a imagem verificando a cor de cada *pixel* e caso essa cor seja maior que o limite estabelecido, ela é transformada em preto. O primeiro teste foi realizado com o limite sendo 127, a metade do valor de um *pixel* branco e ao final foi definido por uma macro e

escolhido através de testes realizados com a variação da constante RANDOM_SEED_OFFSET, elegendo o valor limite cuja imagens geradas após remoção do ruído tinham a menor redução dos aglomerados principais.

Função 02 - int criaMatrizRotulos (Imagem1C* imagem, int* rotulo)**

Para encontrar os dois maiores agrupamentos foi utilizado o método de Rotulação por Componente Conexo, apresentado durante as aulas dedicadas ao desenvolvimento do projeto, como alternativa ao método de conquista. Esse método permitiu rotular cada aglomerado com um número diferente. Isso foi feito ao comparar o conteúdo do índice com as laterais, caso as laterais estivessem vazias eles eram preenchidos com o rótulo, e caso as laterais já estivessem preenchidas, o índice era preenchido com o menor rótulo dentre os laterais. O processo semelhante foi feito percorrendo a matriz do canto inferior direito até o superior esquerdo.

Um dos problemas encontrados foi que realizar a rotulação somente uma vez não era eficaz e acarretava *segmentation fault* na função seguinte. Por isso utilizamos uma flag (flag_rotula) que indicava se ainda haviam *pixels* brancos, em caso positivo, a rotulação ocorria novamente. Ao final da função, é retornada a matriz imagem com todos os aglomerados rotulados.

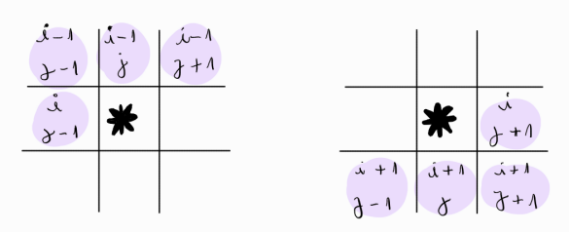

Figura 1- Comparações para preenchimento dos rótulos, de cima para baixo e da esquerda para direita, de baixo para cima e da direita para esquerda, respectivamente.

Função 03 - void defineDoisMaiores(int matriz, unsigned long n_linhas, unsigned long n_col, int tam, Rotulo* maior, Rotulo* segundo_maior)**

Foi utilizado um vetor contador de tamanho [tam], correspondente ao número de rótulos encontrados na função anterior somado a 1, para contagem de quantas vezes cada rótulo apareceu. Um *loop* foi usado para percorrer a matriz da imagem rotulada e outro para armazenar o rótulo dos 2 maiores aglomerados. Além do problema de acesso indevido a memória, decorrente da presença de pixels brancos quando a rotulação era realizada apenas uma vez na função anterior, outro problema enfrentado nessa função foi a eleição incorreta dos dois maiores rótulos. A lógica inicial inicializava as variáveis que armazenavam o maior e o segundo maior rótulo com a posição 0, no entanto, a posição 0 do vetor contador contava todas as vezes que o fundo aparecia na matriz rótulo, resultando em um valor muito alto, o que adulterava o funcionamento da função. Esse erro foi compreendido imprimindo-se o maior e o segundo maior rótulo, além de seus tamanhos, e o vetor contador de rótulos (quando possível, devido ao tamanho) das imagens em que o centro calculado correspondia a aglomerados que não eram os maiores observados nas imagens. Para resolvê-lo, foi necessário apenas zerar a posição 0 do vetor contador.

Função 04 - Coordenada calculaCoordenadaCentro (int matriz, unsigned long n_linhas, unsigned long n_col, Rotulo* rotulo)**

A função é chamada duas vezes, uma para cada agrupado. As coordenadas dos centros foram calculadas baseado na Média ponderada, ou seja, “peso” de cada linha e coluna em relação ao todo.

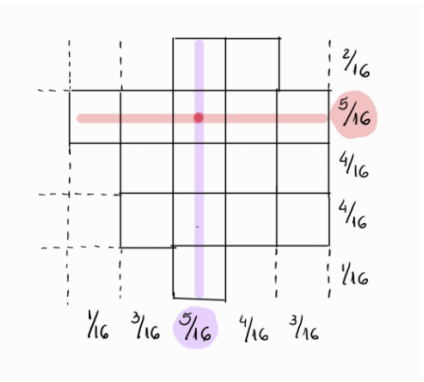

Figura 2 - Abstração através de média ponderada para encontrar o centro do aglomerado.

Ao percorrer a matriz, toda vez que o rótulo era encontrado, acrescentava-se 1 na variável `contribuição`, para calcular a contribuição de cada linha ou coluna. Essas contribuições eram utilizadas para se calcular a média ponderada e encontrar qual linha e qual coluna correspondiam ao meio do aglomerado, obtendo, portanto, as coordenadas da posição central. Esse processo continuava até que houvesse uma coluna/linha inteira sem o rótulo, indicando que o aglomerado havia terminado (flag `continua_buscando`).

Função 05 - double calculaAngulo (Coordenada esquerda, Coordenada direita)

O cálculo do ângulo solicitado foi feito através de noções de cálculo trigonométrico, uma vez que as linhas formadas entre os aglomerados e o eixo X formavam um triângulo retângulo. A partir disso era possível obter os valores de variação de X (cateto adjacente, largura) e Y (cateto oposto, altura) para o cálculo da tangente. A função atan da biblioteca <math.h> foi utilizada retornando um valor double em radianos correspondentes ao ângulo solicitado.

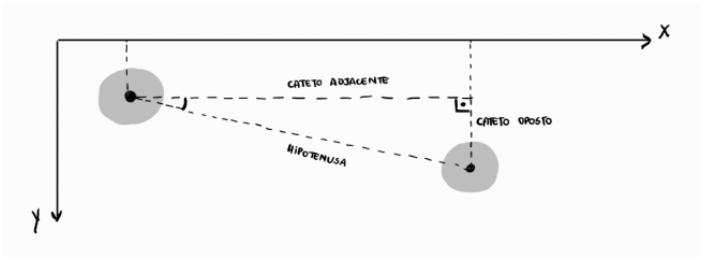

Figura 3 - Triângulo retângulo formado pela intersecção das linhas.

Função 06 - void destroiMatrizRotulo (int** matriz, int n_linhas)

Neste projeto foi utilizado alocação dinâmica para a matriz rótulo, sendo nela que a maior parte do processo foi feito. Ao final da função `detectaSensorBar` esta função é chamada para desalocar o espaço de memória utilizado.

Conclusões

Este projeto foi o mais longo e trabalhoso do semestre e utilizou todos os conhecimentos obtidos durante a disciplina. Por esse motivo, também foi o mais interessante em que todo o aprendizado foi utilizado em conjunto.

A lógica utilizada foi desenvolvida através de debates, e na maioria das vezes funcionava corretamente. Ocorriam algumas dificuldades quando buscávamos passar a lógica para o código,

e raras vezes em relação a sintaxe, porém, esses erros eram mais facilmente achados, uma vez que o código era revisado por 3 pessoas. Além do mais, isso tudo contribuía para interação entre as componentes do grupo que precisavam conversar com mais frequência em situações assim.

Ao final do projeto, definimos N_TESTES como 1000. O projeto compila e executa perfeitamente com um erro médio de aproximadamente 0.08, sendo que poucos testes tiveram erros e quando ocorriam, eram pequenos. Os resultados são extremamente satisfatórios para nós.