Relatório Trabalho 1
Geometria Computacional

Matheus Fonseca Alexandre de Oliveira
RA: 1794027
Professor: Ricardo Dutra da Silva

Estrutura DCEL

A DCEL (*Doubly Connected Edge List*) é uma estrutura de dados para representar objetos geométricos (retas, pontos e planos), baseada em arestas (*Edges*).

A estrutura é composta de 3 elementos principais:

- *Edge* para representar uma aresta no plano.
- *Face* para representar uma face 2D.
- *Point* para representar um ponto no plano.

Que são implementadas da seguinte maneira:

```c
/* Estrutura DCEL */
typedef struct Edge
{
    /* Ponto de origem da aresta */
    struct Point *point;
    /* Ponteiro para a próxima aresta */
    struct Edge *next;
    /* Ponteiro para a aresta anterior*/
    struct Edge *prev;
    /* Ponteiro para a aresta gêmea */
    struct Edge *twin;
    /* Ponteiro para a face a sua esquerda */
    struct Face *left_face;
} Edge;

/* Estrutura do ponto no espaço 2D*/
typedef struct Point
{
    /* Valor x no espaço */
    double x;
    /* Valor y no espaço */
    double y;
    /* Aresta que tem como origem este ponto */
    struct Edge *edge;
} Point;

/* Estrutura da face no espaço 2D */
typedef struct Face
{
    /* Aresta que pertence a face */
    struct Edge *edge;
} Face;
```

Figura 1: Estrutura DCEL

A estrutura Edge representa uma aresta. Ela contém um ponto de origem, a próxima aresta na sequência (em sentido anti horário) e a aresta prévia na sequência (em sentido anti horário). A DCEL trabalha com arestas irmãs, por isso existe a Twin, aresta irmã da aresta atual. Ela tem como ponto de origem o ponto destino da aresta (por isso, não precisamos salvar na aresta o seu destino). Além disso, serve como uma auxiliar em diversas operações que serão descritas mais para frente.

A estrutura de ponto Point é mais simples. Como a DCEL se baseia em arestas, o ponto só precisa carregar consigo sua coordenada geográfica (x,y) e uma aresta que saí desse ponto. Como podem existir várias arestas partindo do mesmo ponto, só precisamos guardar uma delas.

A estrutura Face representa os semiplanos que são / podem ser criados com a DCEL. Eles são referenciados unicamente por uma aresta . Essa aresta tem o mesmo intuito que a aresta da estrutura de ponto: das várias arestas que pertencem a uma face, só precisamos guardar uma delas.

Funções de manipulação

Abaixo segue a implementação e a descrição rápida de cada função utilizada para a manipulação da estrutura:

0.1 Criações das estruturas

```
/*
*   Cria dinamicamente uma aresta
*   Aresta é isolada, sem pontos definidos.
*   Next de new_edge é sua twin. Prev de new_edge é sua twin.
*   Ambas apontam para a mesma face, que é todo o resto do plano (null)
*/
Edge* createEdge()
{
    // Aloca os espaços
    Edge *edge = new(Edge);
    Edge *edge_twin = new(Edge);

    // Associa os twins.
    edge->twin = edge_twin;
    edge_twin->twin = edge;

    // Associa os next e prev
    edge->next = edge->twin;
    edge->prev = edge->twin;

    edge_twin->next = edge_twin->twin;
    edge_twin->prev = edge_twin->twin;

    // Associa as faces
    edge->left_face = NULL;
    edge_twin->left_face = NULL;

    // Associa os pontos
    edge->point = NULL;
    edge_twin->point = NULL;

    return edge;
}
```

Figura 2: Função createEdge

Inicializa a estrutura de Edge. Cria duas "halfedges", e conecta ambas pelos campos de Twin, Next, e Prev. Os campos de Face e de Pontos são nulos.

```
/*
*   Cria dinamicamente um ponto
*   Ponto utiliza as coordenadas x,y
*   Inicialmente, não tem nenhuma aresta associdada. (null)
*/
Point* createPoint(double x, double y)
{
    Point *new_point = new(Point);
    new_point->x = x;
    new_point->y = y;
    new_point->edge = NULL;
    #ifdef CREATE_POINT_DEBUG
        cout << "Point created: " << new_point << "\n";
        cout << "Values: (" << new_point->x << "," << new_point->y << ")
        \n";
    #endif
    return new_point;
}
```

Figura 3: Função createPoint

Inicializa a estrutura de Point. Associa as coordenadas x,y com os parâmetros recebidos. A sua aresta é nula.

0.2 Funções Específicas

Nessa seção trata-se das funções explícitas exigidas pelo trabalho. Ao todo são quatro funções: Órbita de um Vértice, Arestas de uma Face, Inclusão de Arestas e Inclusão de Vértices.

0.2.1 Órbita de um vértice

Para percorrer todas as arestas na órbita de um ponto, a seguinte função é utilizada:

```
void percorreOrbita(Point *ponto)
{
    Edge *aresta;
    aresta = ponto->edge;
    int i = 0;
    do
    {
        // Aqui entra algum print / cout
        cout << "Aresta: " << i;
        cout << aresta->point->x << "," << aresta Edge *aresta
        cout << aresta->twin->point->x << "," << aresta->twin->point->y;
        cout << "\n";
        aresta = aresta->prev->twin;
        i++;
    } while (aresta != ponto->edge);

}
```

Figura 4: Função percorreOrbita

0.2.2 Arestas de uma Face

Para percorrer todas as arestas de uma determinada Face, a seguinte função é utilizada:

```
void percorreFace(Face *face)
{
    Edge *aresta;
    aresta = face->edge;
    int i = 0;
    do
    {
        // Aqui entra algum print / cout
        cout << "Aresta: " << i;
        cout << aresta->point->x << "," << aresta->point->y;
        cout << aresta->twin->point->x << "," << aresta->twin->point->y;
        cout << "\n";
        aresta = aresta->next;
        i++;
    } while (aresta != face->edge);

}
```

Figura 5: Função percorreFace

0.2.3 Inclusão de Arestas

Para associar uma aresta aos seus pontos, é utilizada uma outra função mais complexa, chamada connect. Essa função é a primeira específica no trabalho, conectar uma aresta a uma órbita de um vértice. A necessidade de se trabalhar com órbitas é porque as arestas conectadas aos pontos modificam os valores de next e prev de algumas halfedges.

A primeira parte dessa etapa é apenas a conexão de uma halfedge com seus pontos. É feito uma verificação a mais para checar se os pontos passados como parâmetro já tem alguma aresta associada ou não. No caso negativo, a halfedge que acabamos de criar passa a ser os ponteiros dessa aresta.

```
/*
*   Conecta 2 pontos do plano.
*   Para isso, cria uma aresta entre esses dois pontos.
*   Arruma os ponteiros das arestas
*   Arruma a órbita dos pontos.
*/
void connect(Point *vertex_A, Point *vertex_B)
{
    // Cria uma aresta
    Edge *edge;
    edge = createEdge();

    // Associa os pontos a essa aresta.
    edge->point = vertex_A;
    edge->twin->point = vertex_B;

    // Verifica se o ponto A e B já tem uma aresta associada
    if(vertex_A->edge != NULL && vertex_B->edge != NULL)
    {
        /*
        *   Acha aresta e1 e e2 (ver material de aula) para arrumar os ponteiros.
        *   Para isso, utiliza a função inCone_dcel
        */
        Edge *edge_1, *edge_2;
        edge_1 = inCone_dcel(edge);
        edge_2 = edge_1->twin->next;

        // Conecta as órbitas
        connectOrbit(edge_1, edge);
        connectOrbit(edge_2, edge->twin);
    }
    // Caso contrário, é a primeira aresta que o ponto recebe.
    else
    {
        if(vertex_A->edge == NULL)
        {
            // Associa ao ponto A a aresta criada.
            #ifdef CONNECT_DEBUG
                cout << "Points A: " << vertex_A;
                cout << " doesn't have any orbit.\n";
            #endif
            vertex_A->edge = edge;
        }

        if(vertex_B->edge == NULL)
        {
            // Associa ao ponto A a aresta criada.
            #ifdef CONNECT_DEBUG
                cout << "Points B: " << vertex_B;
                cout << " doesn't have any orbit.\n";
            #endif
            vertex_B->edge = edge->twin;
        }
    }

}
```

Figura 6: Função Connect

Já para o caso negativo, é necessário utilizar a função inCone e connectOrbit para ajustar os valores dos vetores. A função inCone e a função connectOrbit são detalhadas na seção de

Funções Auxiliares. No geral, a função inCone vai procurar as duas arestas que estão entre as arestas na qual estamos querendo conectar, e a função connectOrbit, de posse dessas arestas, atualiza devidamente os ponteiros.

0.2.4 Inclusão de Vértices

Não implementado até o presente momento. Os pontos adicionados podem se coincidir / e pontos no meio de arestas não são verificados

Funções Auxiliares

Funções adicionais implementadas como apoio deste trabalho:

```
/*
*   Verifica a condição de left de C para a aresta AB.
*/
bool left(Point *A, Point *B, Point *C)
{
    double area;
    area = ((B->x - A->x) * (C->y - A->y)) - ((C->x - A->x) * (B->y -
    A->y) );
    area = area/2;
    return (area > 0) ? true : false;
}

/*
*   Verifica a condição de leftOn de C para a aresta AB.
*/
bool leftOn(Point *A, Point *B, Point *C)
{
    double area;
    area = ((B->x - A->x) * (C->y - A->y)) - ((C->x - A->x) * (B->y -
    A->y) );
    area = area/2;
    return (area >= 0) ? true : false;
}
```

Figura 7: Funções Lefts

Utilizadas dentro da função inCone para verificar se uma determinada aresta está entre dois pontos (A esquerda de um e a direita de outro).

```c
/*
*   Apaga a existência da aresta entre os pontos A e B.
*   A aresta vai continuar existindo, mas as órbitas de A e B não vão
mais
*   estar relacionadas com essa aresta
*   Será necessário desalocar esse vetores manualmente.
*/
void disconnect(Point *vertex_A, Point *vertex_B)
{
    /*
    *   Busca aresta com origem em A, cuja sua irmã gêmea tem origem em B
    *   Se sua irmã não tem a origem em B, pega sua próxima da órbita
    */
   Edge *vertex_auxiliar;
   do
   {
       vertex_auxiliar = vertex_A->edge;
       if(vertex_auxiliar->twin->point == vertex_B)
       {
           break;
       }
        vertex_auxiliar = vertex_auxiliar->twin->next;
   } while ( vertex_auxiliar != vertex_A->edge);

    /*
    *   Uma vez com aresta encontrada, vamos encontrar e1 e e2.
    *   Aqui, elas já estão óbvias.
    *   e1 é a previous (da órbita).
    *   e2 é a next (da órbita).
    */

    Edge *e1, *e2;
    e1 = vertex_auxiliar->next;
    e2 = vertex_auxiliar->twin->next;
    disconnectOrbit(e1, vertex_auxiliar);
    disconnectOrbit(e2, vertex_auxiliar->next);
}
```

Figura 8: Função disconnect

Usada durante a exclusão de uma aresta. A função disconnect desassocia os valores dos ponteiros da aresta que vão ser removidos, e utiliza a disconnectOrbit para arrumar as órbitas dos pontos.

```c
/*
*   Arruma a órbita de 1,
*   Apagando a aresta 2 e arrumando os ponteiros corretamente
*/
void disconnectOrbit(Edge *edge_1, Edge *edge_2)
{
    edge_1->prev = edge_2->twin->prev;
    edge_1->twin->next = edge_2;
    edge_2->twin->prev->next = edge_1;
    edge_2->prev = edge_1->twin;
}
```

Figura 9: Função disconnectOrbit

Função utilizada para ajustar a órbita do vértice ligado ao edge_1.

```
/*
*   Verifica se a aresta que AB está dentro do polígono.
*   Para isso, verifica se essa aresta está dentro do cone formado por A,
A- e A+.
*   Achar A- e A+ é a nossa prioridade.
*/
Edge* inCone_dcel(Edge *edge)
{
    /*
    *   A aresta AB é encontrada usando a estrutura DCEL.
    *   A é a origem da edge, B é a origem de sua twin.
    */
    Point *ponto_inicial = edge->point;
    Point *ponto_final = edge->twin->point;

    /*
    *    Percorre a órbita de A
    *    A0 é aresta AA-
    *    A1 é a aresta AA+
    *    Sentido Anti Horário
    */
    Edge *vertex_A0 = ponto_inicial->edge->twin->next;
    Edge *vertex_A1 = vertex_A0->twin->next;

    #ifdef INCONE_DEBUG
        cout << "Incone Debug = ***\n";
        cout << "Vertex AO: " << vertex_A0->point->x << "," <<
        vertex_A0->point->y << "\n";
        cout << "Vertex A1: " << vertex_A1->point->x << "," <<
        vertex_A1->point->y << "\n";
    #endif
    /*
    *    Estrutura de do/while para achar as aretas A0 e A1 que queremos.
    *    While só encerra quando A0 se torna a aresta inicial de novo.
    */
```

Figura 10: Função InCone Parte 1

```
    do
    {
        if( leftOn(ponto_inicial, vertex_A1->point, vertex_A0->point))
        {
            /*
            * Se LeftOn retornar verdadeiro, é porque A0 A A1 é um vértice convexo
            * Checamos agora usando Left se são os vértices que queremos
            */
        if( left(ponto_inicial, ponto_final, vertex_A0->point) &&
            left(ponto_final, ponto_inicial, vertex_A1->point))
            {
                /*
                * Achamos nossas arestas E1 e E2
                * Retornamos apenas E1, pois E2 já sabemos como acessar
                */
                break;
            }
        }
        else
        {
            /*
            * Se leftOn retornar falso, é porque A0 A A1 é um vértice reflexo
            * Checamos agora usando um "Right" se são os vértices que queremos
            */
            if(!(leftOn(ponto_inicial, ponto_final, vertex_A1->point) &&
                leftOn(ponto_final, ponto_inicial, vertex_A0->point)))
                {
                    /*
                    * Achamos nossas arestas E1 e E2
                    * Retornamos apenas E1, pois E2 já sabemos como acessar
                    */
                    break;
                }
        }
        /*
        * Não achamos a aresta
        * Movemos as arestas A0 e A1 e checamos novamente.
        */
        vertex_A0 = vertex_A1;
        vertex_A1 = vertex_A1->twin->next;
    }
    while(vertex_A1 != ponto_inicial->edge);

    return vertex_A0;
}
```

Figura 11: Função InCone Parte 2

A função inCone é utilizada para encontrar duas arestas que formam um cone dentro da aresta que estamos tentando inserir. Utiliza lefts pra leftsOn.

```c
/*
*   Limpa os ponteiros da aresta, desalocando a memória
*   Cuidado com arestas que apontam para ela!, pode dar merda
*/
void freeEdge(Edge *edge)
{
    free(edge);
}

void freePoint(Point *point)
{
    free(point);
}
```

Figura 12: Funções de Liberação de Memória

```c
void splitFace()
{
    /* TODO: Implement */

}

void splitEdge()
{
    /* TODO: Implement */
}
```

Figura 13: Funções não implementadas