Implementação de DCEL para Subdivisões Planares

Autor: Ludwig Aumann

Introdução

Este artigo apresenta uma implementação completa de uma estrutura de dados DCEL (Doubly Connected Edge List) para representação e validação de subdivisões planares. O sistema resolve problemas fundamentais em geometria computacional:

1. **Construção de DCEL**: criação da estrutura a partir de descrições de faces
2. **Validação topológica**: verificação de propriedades de subdivisões planares válidas
3. **Detecção de problemas**: identificação de malhas abertas, não-planares ou superpostas

Definições

Antes de descrever a implementação, definimos os conceitos fundamentais:

- **DCEL**: estrutura que representa subdivisões planares usando vértices, semi-arestas e faces
- **Semi-aresta (Half-edge)**: aresta direcionada que conecta dois vértices
- **Twins**: par de semi-arestas que representam a mesma aresta geométrica em direções opostas
- **Subdivisão planar**: partição do plano em regiões (faces) delimitadas por arestas
- **Malha fechada**: todas as arestas possuem twins (sem fronteiras)

Estrutura de Dados

1. Componentes Básicos

A implementação utiliza três estruturas principais conectadas por ponteiros:

```
struct Point {
    int x, y;  // coordenadas inteiras para evitar imprecisao numerica
};

struct Vertex {
    Point position;
    int index;
    HalfEdge* incidentEdge;  // uma semi-aresta que parte deste vertice
};

struct Face {
    int index;
    HalfEdge* outerComponent;  // semi-aresta do contorno externo
};

struct HalfEdge {
    int index;
```

```
    Vertex* origin;           // vertice de origem
    HalfEdge* twin;          // semi-aresta gemea (direcao oposta)
    Face* incidentFace;      // face à esquerda desta semi-aresta
    HalfEdge* next;          // proxima semi-aresta no contorno da face
    HalfEdge* prev;          // semi-aresta anterior no contorno da face
};
```

Conceito de outerComponent

O `outerComponent` é um ponteiro para uma semi-aresta arbitrária do contorno da face, servindo como **ponto de entrada** para navegação.

Por que é necessário?

- Faces são definidas por ciclos de semi-arestas conectadas via `next`/`prev`
- Sem uma referência inicial, seria impossível acessar as semi-arestas da face
- `outerComponent` garante que sempre temos um "ponto de partida" para navegar pelo contorno

Como é usado?

```
// navegar ao redor de uma face
HalfEdge* start = face->outerComponent;
HalfEdge* current = start;
do {
    // processar semi-aresta atual
    current = current->next;
} while (current != start);  // volta ao inicio = ciclo completo
```

2. Estratégia de Indexação

O sistema utiliza uma estratégia de conversão de índices para compatibilidade:

- **Entrada**: índices baseados em 1 (padrão acadêmico)
- **Interno**: índices baseados em 0 (padrão de arrays em C++)
- **Saída**: índices baseados em 1 (conversão de volta)

```
static int inputToInternal(int inputIndex) { return inputIndex - 1; }
static int internalToOutput(int internalIndex) { return internalIndex + 1; }
```

Algoritmos de Construção

1. Carregamento de Dados

O algoritmo `loadFromInput()` processa a entrada em três etapas:

```
bool DCEL::loadFromInput() {
    // le cabecalho: numero de vertices e faces
    int nVertices, nFaces;
    scanf("%d %d", &nVertices, &nFaces);

    // le coordenadas dos vertices
    for (int i = 0; i < nVertices; i++) {
        int x, y;
        scanf("%d %d", &x, &y);
```

```
        vertices.push_back(std::make_unique<Vertex>(Point(x, y), i));
    }

    // le cada face como sequencia de vertices ate encontrar quebra de linha
    for (int i = 0; i < nFaces; i++) {
        std::vector<int> faceVertices;
        int vertexIndex;
        while (scanf("%d", &vertexIndex) == 1) {
            faceVertices.push_back(inputToInternal(vertexIndex));
            if (getchar() == '\n') break;
        }
        faceVertexIndices.push_back(std::move(faceVertices));
    }

    return constructDCEL();
}
```

Complexidade: O(n + m), onde n é o número de vértices e m é o número total de vértices em todas as faces.

2. Criação de Semi-arestas

O algoritmo `createHalfEdges()` constrói as semi-arestas e estabelece relações twins:

```
bool DCEL::createHalfEdges() {
    int halfEdgeIndex = 0;

    // itera sobre cada face para criar suas half-edges
    for (size_t faceIdx = 0; faceIdx < faces.size(); faceIdx++) {
        const auto& faceVertices = faceVertexIndices[faceIdx];

        // cria half-edges para cada par de vertices consecutivos da face
        for (size_t i = 0; i < numVertices; i++) {
            int fromIdx = faceVertices[i];
            int toIdx = faceVertices[(i + 1) % numVertices];

            // chave para identificar arestas gemeas (twins)
            EdgeKey key(fromIdx, toIdx);

            auto halfEdge = std::make_unique<HalfEdge>();
            halfEdge->origin = vertices[fromIdx].get();
            halfEdge->incidentFace = faces[faceIdx].get();

            // mapeia half-edges para encontrar twins posteriormente
            edgeMap[key] = halfEdge;
        }
    }

    // conecta as half-edges gemeas (twins) usando o mapeamento criado
    for (auto& [key, edgePair] : edgeMap) {
        if (edgePair.first && edgePair.second) {
            edgePair.first->twin = edgePair.second;
            edgePair.second->twin = edgePair.first;
```

```
        }
    }
}
```

Normalização de Arestas com EdgeKey

A `EdgeKey` resolve o problema de identificar semi-arestas que representam a mesma aresta geométrica:

```
struct EdgeKey {
    int from, to;
    EdgeKey(int f, int t) : from(std::min(f, t)), to(std::max(f, t)) {}

    bool operator<(const EdgeKey& other) const {
        return from < other.from || (from == other.from && to < other.to);
    }
};
```

Problema: Semi-arestas são direcionadas (A→B e B→A), mas representam a mesma aresta geométrica.

Solução: EdgeKey normaliza sempre para `(min, max)`, garantindo que A→B e B→A gerem a mesma chave.

Exemplo: Para vértices 2 e 5:

- Semi-aresta 2→5 gera EdgeKey(2,5)
- Semi-aresta 5→2 gera EdgeKey(2,5)
- Sistema reconhece automaticamente que são twins

Algoritmo de mapeamento:

```
std::map<EdgeKey, std::pair<HalfEdge*, HalfEdge*>> edgeMap;

for (cada semi-aresta he) {
    EdgeKey key(he->origin->index, he->destination()->index);

    if (edgeMap.find(key) == edgeMap.end()) {
        edgeMap[key] = {he, nullptr};  // primeira semi-aresta
    } else {
        edgeMap[key].second = he;      // segunda semi-aresta (twin)
    }
}

// conecta twins
for (auto& [key, edgePair] : edgeMap) {
    if (edgePair.first && edgePair.second) {
        edgePair.first->twin = edgePair.second;
        edgePair.second->twin = edgePair.first;
    }
}
```

Benefícios:

- Identificação automática de twins em O(log n) por lookup

- Detecção de arestas órfãs (EdgeKey com apenas uma semi-aresta)
- Contagem eficiente de faces por aresta para validação de planaridade
- Garantia de consistência topológica

Complexidade: O(m), onde m é o número total de semi-arestas.

3. Ligação de Cadeias Circulares

O algoritmo `linkHalfEdgeChains()` conecta as semi-arestas de cada face formando ciclos ordenados. **Objetivo**: estabelecer ponteiros `next` e `prev` para permitir navegação ao redor de cada face.

Como funciona a ligação

Para cada face, começamos a navegar pelas semi-arestas usando `face->outerComponent`, e então:

1. **Ponto inicial**: `face->outerComponent` nos dá a primeira semi-aresta da face
2. **Busca da próxima**: Para a semi-aresta atual A→B, procuramos outra semi-aresta B→C que:
    - Pertence à mesma face (`incidentFace == face`)
    - Começa onde a atual termina (`origin == targetVertex`)
3. **Conexão**: Estabelecemos `atual->next = próxima` e `próxima->prev = atual`
4. **Avanço**: Movemos para a próxima semi-aresta
5. **Repetição**: Continuamos até voltar à semi-aresta inicial (ciclo fechado)

Resultado: Uma cadeia circular onde cada semi-aresta aponta para a próxima no contorno da face.

```
void DCEL::linkHalfEdgeChains() {
    // conecta as half-edges de cada face em uma cadeia circular ordenada
    for (const auto& face : faces) {
        HalfEdge* start = face->outerComponent;
        if (!start) continue;

        HalfEdge* current = start;
        // previne loops infinitos em casos mal formados
        std::vector<HalfEdge*> faceEdges;

        do {
            faceEdges.push_back(current);

            // busca a proxima half-edge que comeca onde a atual termina
            HalfEdge* next = nullptr;
            Vertex* targetVertex = current->destination();

            for (const auto& he : halfEdges) {
                if (he->incidentFace == face.get() &&
                    he->origin == targetVertex) {
                    next = he.get();
                    break;
                }
            }

            // estabelece conexoes bidirecionais entre half-edges consecutivas
```

```cpp
            current->next = next;
            if (next) {
                next->prev = current;
            }
            current = next;

            // continua ate fechar o ciclo ou detectar problema
        } while (current && current != start &&
                 faceEdges.size() < halfEdges.size());
    }
}
```

Proteções contra Loops Infinitos

O algoritmo implementa várias proteções para evitar comportamentos estranhos em estruturas mal formadas:

1. Contador de segurança:

```cpp
std::vector<HalfEdge*> faceEdges;  // rastreia semi-arestas visitadas
// ...
} while (current && current != start &&
         faceEdges.size() < halfEdges.size());  // para se visitar muitas semi-arestas
```

2. Condições de parada múltiplas:

- `current == nullptr` : não encontrou próxima semi-aresta (cadeia quebrada)
- `current == start` : voltou ao início (ciclo fechado corretamente)
- `faceEdges.size() >= halfEdges.size()` : visitou semi-arestas demais (loop infinito)

Cenários problemáticos detectados:

- **Cadeia quebrada**: Semi-aresta sem sucessora válida → para com `current == nullptr`
- **Múltiplos caminhos**: Semi-aresta com múltiplas sucessoras → primeira encontrada é escolhida
- **Referência circular**: Face aponta para si mesma → detectado pelo contador de segurança
- **Semi-aresta órfã**: Não pertence a ciclo válido → detectado na validação posterior

Complexidade: $O(m^2)$ no pior caso, devido à busca linear da próxima semi-aresta.

Algoritmos de Validação

A implementação verifica se a DCEL satisfaz os critérios fundamentais de uma subdivisão planar válida:

Critérios de Validação

1. Orientação dos vértices: Os vértices das faces devem estar em ordem anti-horária, sendo o lado "externo" aquele onde esta ordem se mantém.

2. Subdivisão completa: A malha deve ser uma subdivisão completa do plano, onde cada aresta aparece como fronteira de exatamente duas faces (incluindo uma face externa).

3. Integridade geométrica: As faces não devem se auto-intersectar e seus interiores devem ser disjuntos.

Saídas de erro correspondentes:

- `"aberta"`: alguma aresta é fronteira de somente uma face (malha incompleta)
- `"não subdivisão planar"`: alguma aresta é fronteira de mais/menos de duas faces
- `"superposta"`: alguma face tem auto-interseção ou intersecta outras faces

Face Externa e Subdivisão Completa

Uma subdivisão planar completa **sempre inclui uma face externa** que representa a região infinita fora da malha. Esta face garante que:

- Cada aresta de fronteira tenha exatamente duas faces incidentes (interna + externa)
- A contagem de faces por aresta seja sempre 2 em malhas fechadas
- A estrutura seja topologicamente consistente

Exemplo: Um triângulo isolado cria 2 faces: a face triangular interna e a face externa infinita que a envolve.

1. Verificação de Malha Fechada

O algoritmo `hasOpenEdges()` verifica propriedades fundamentais das semi-arestas:

```
bool DCEL::hasOpenEdges() const {
    for (const auto& he : halfEdges) {
        // half-edge sem twin indica aresta de fronteira (malha aberta)
        if (!he->twin) return true;

        // relacao twin deve ser simetrica
        if (he->twin->twin != he.get()) return true;

        // twins devem pertencer a faces diferentes
        if (he->incidentFace == he->twin->incidentFace) return true;
    }
    return false;
}
```

Complexidade: O(m), onde m é o número de semi-arestas.

2. Verificação de Planaridade

O algoritmo `isNonPlanarSubdivision()` conta quantas faces cada aresta toca:

```
bool DCEL::isNonPlanarSubdivision() const {
    // conta quantas faces cada aresta geometrica toca para validar planaridade
    std::map<EdgeKey, int> edgeFaceCount;

    for (const auto& he : halfEdges) {
        // normaliza aresta para chave unica independente da direcao
        EdgeKey key(he->origin->index, he->destination()->index);
        edgeFaceCount[key]++;
    }
```

```
    // em subdivisao planar valida cada aresta deve tocar exatamente 2 faces
    for (const auto& [key, count] : edgeFaceCount) {
        if (count != 2) return true;
    }
    return false;
}
```

Aqui reutilizamos a EdgeKey para um propósito diferente: cada semi-aresta incrementa o contador da sua aresta geométrica correspondente. Como twins sempre geram a mesma EdgeKey, contamos efetivamente quantas semi-arestas (e portanto faces) cada aresta física toca.

Em uma subdivisão planar válida, cada aresta deve separar exatamente duas faces. Contagens diferentes indicam:

- **Contagem = 1**: aresta de fronteira (malha aberta)
- **Contagem > 2**: estrutura impossível em 2D

Complexidade: O(m log m), devido ao uso de `std::map`.

3. Detecção de Interseções

O algoritmo `hasIntersectingFaces()` verifica cruzamentos entre semi-arestas para garantir que as faces não se auto-intersectem e que seus interiores sejam disjuntos:

```
bool DCEL::hasIntersectingFaces() const {
    for (size_t i = 0; i < halfEdges.size(); i++) {
        for (size_t j = i + 1; j < halfEdges.size(); j++) {
            const auto& he1 = halfEdges[i];
            const auto& he2 = halfEdges[j];

            // pula casos validos (twins, adjacencias)
            if (areValidlyAdjacent(he1, he2)) continue;

            if (Geometry::segmentsIntersect(
                    he1->getSegmentStart(), he1->getSegmentEnd(),
                    he2->getSegmentStart(), he2->getSegmentEnd())) {
                return true;
            }
        }
    }
    return false;
}
```

Complexidade: $O(m^2)$, onde m é o número de semi-arestas.

O algoritmo detecta interseções impróprias que violam a propriedade de interiores disjuntos.

Algoritmos Geométricos

1. Cálculo de Orientação

Base para todos os testes geométricos:

```cpp
namespace Geometry {
    enum class Orientation {
        COLINEAR, HORARIO, ANTIHORARIO
    };

    Orientation orientation(const Point& a, const Point& b, const Point& c) {
        // calcula o produto vetorial 2D para determinar orientacao
        long long val = (long long)(b.y - a.y) * (c.x - b.x) -
                       (long long)(b.x - a.x) * (c.y - b.y);
        if (val == 0) return Orientation::COLINEAR;
        return (val > 0) ? Orientation::ANTIHORARIO : Orientation::HORARIO;
    }
}
```

O produto vetorial 2D corresponde ao dobro da área do triângulo formado pelos três pontos. O sinal determina a orientação.

Complexidade: O(1).

2. Interseção de Segmentos

Algoritmo para detectar interseções:

```cpp
bool segmentsIntersect(const Point& p1, const Point& q1,
                      const Point& p2, const Point& q2) {
    // casos triviais: endpoints coincidentes
    if (p1 == p2 || p1 == q2 || q1 == p2 || q1 == q2) {
        return false;
    }

    // calcula orientacoes para teste geral
    Orientation o1 = orientation(p1, q1, p2);
    Orientation o2 = orientation(p1, q1, q2);
    Orientation o3 = orientation(p2, q2, p1);
    Orientation o4 = orientation(p2, q2, q1);

    // teste geral: pontos em lados opostos
    if (o1 != o2 && o3 != o4) return true;

    // casos especiais com colinearidade
    if (o1 == Orientation::COLINEAR && onSegment(p1, p2, q1)) return true;
    if (o2 == Orientation::COLINEAR && onSegment(p1, q2, q1)) return true;
    if (o3 == Orientation::COLINEAR && onSegment(p2, p1, q2)) return true;
    if (o4 == Orientation::COLINEAR && onSegment(p2, q1, q2)) return true;

    return false;
}
```

Complexidade: O(1).

Análise de Complexidade

Complexidade Temporal

- **Carregamento**: O(n + m)
- **Construção de semi-arestas**: O(m)
- **Ligação de cadeias**: $O(m^2)$
- **Validação de fechamento**: O(m)
- **Validação de planaridade**: O(m log m)
- **Detecção de interseções**: $O(m^2)$
- **Complexidade total**: $O(m^2)$

Complexidade Espacial

- **Vértices**: O(n)
- **Faces**: O(f)
- **Semi-arestas**: O(m)
- **Mapeamento de arestas**: O(m)
- **Complexidade total**: O(n + f + m)

Referências

1. de Berg, M., Cheong, O., van Kreveld, M., & Overmars, M. (2008). *Computational Geometry: Algorithms and Applications*. Springer-Verlag.