Discussion 05

Half-edges and Ray Tracing

Computer Graphics and Imaging
UC Berkeley CS 184/284A

Week 5 Announcements

<u>Exam on Monday 3:30 - 5:00 PM</u>

Content Covered

2. Drawing Triangles
3. Sampling and Aliasing
4. Transforms
5. Texture Mapping
6. Rasterization Pipeline
7. Bezier Curves and Surfaces
8. Mesh Representations and Geometry Processing
9. Ray Tracing
10. Ray Tracing - Acceleration Structures

No Radiometry and Photometry
No Monte Carlo

Half-Edge Background

Preferred way to represent 3D shapes: Meshes

Many ways to represent meshes:

- List of triangles
- List of points + indexed triangle
- Triangle neighbor

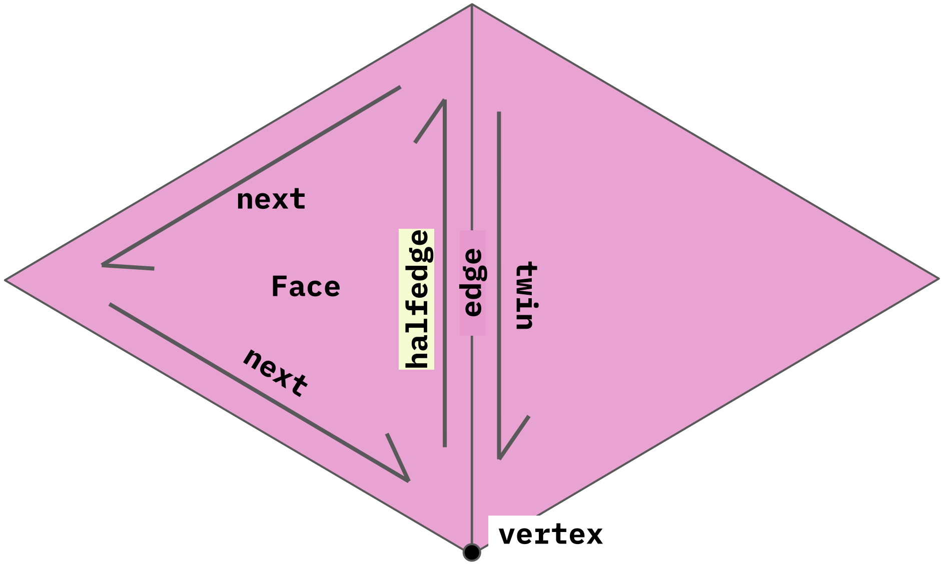
Downsides: hard to edit, store redundant data, hard to navigate

Half-Edges

The Half-Edge Data Structure

```
struct Halfedge {
        Halfedge *twin,
        Halfedge *next;
        Vertex *vertex;
        Edge *edge;
        Face *face;
}
```

Key idea: two half edges act as **"glue"** between mesh elements



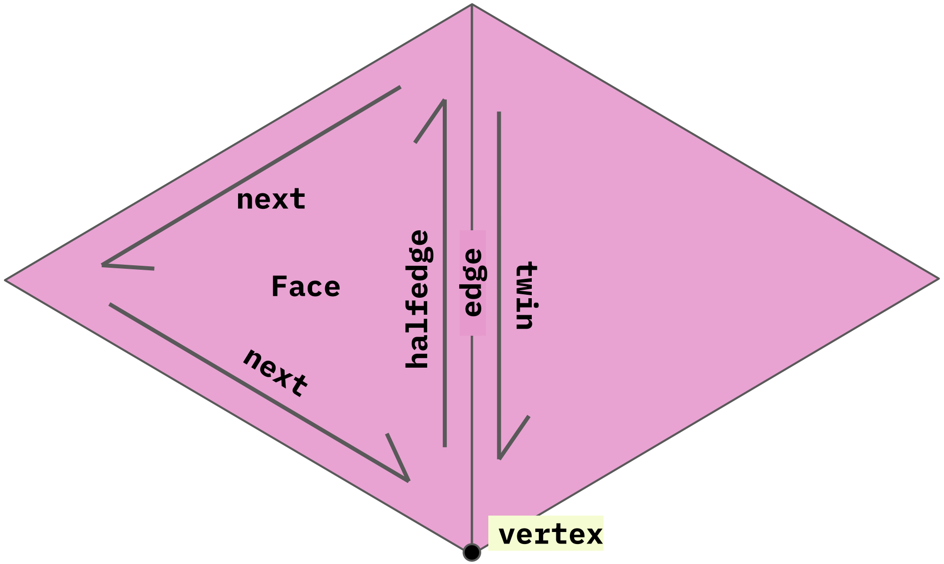
Each vertex, edge, and face points to one of its half edges

The Half-Edge Data Structure

```
struct Halfedge {
       Halfedge *twin,
       Halfedge *next;
       Vertex *vertex;
       Edge *edge;
       Face *face;
}

struct Vertex {
       Point pt;
       Halfedge *halfedge;
}
```

Key idea: two half edges act as **"glue"** between mesh elements



Each vertex, edge, and face points to one of its half edges

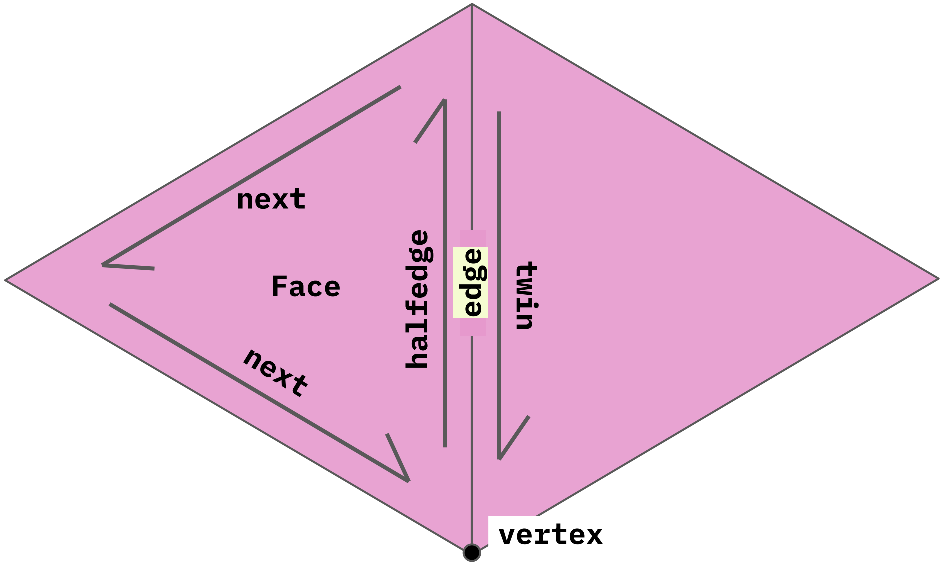
The Half-Edge Data Structure

```
struct Halfedge {
        Halfedge *twin,
        Halfedge *next;
        Vertex *vertex;
        Edge *edge;
        Face *face;
}

struct Vertex {
        Point pt;
        Halfedge *halfedge;
}

struct Edge {
        Halfedge *halfedge;
}
```

Key idea: two half edges act as **“glue”** between mesh elements



Each vertex, edge, and face points to one of its half edges

The Half-Edge Data Structure

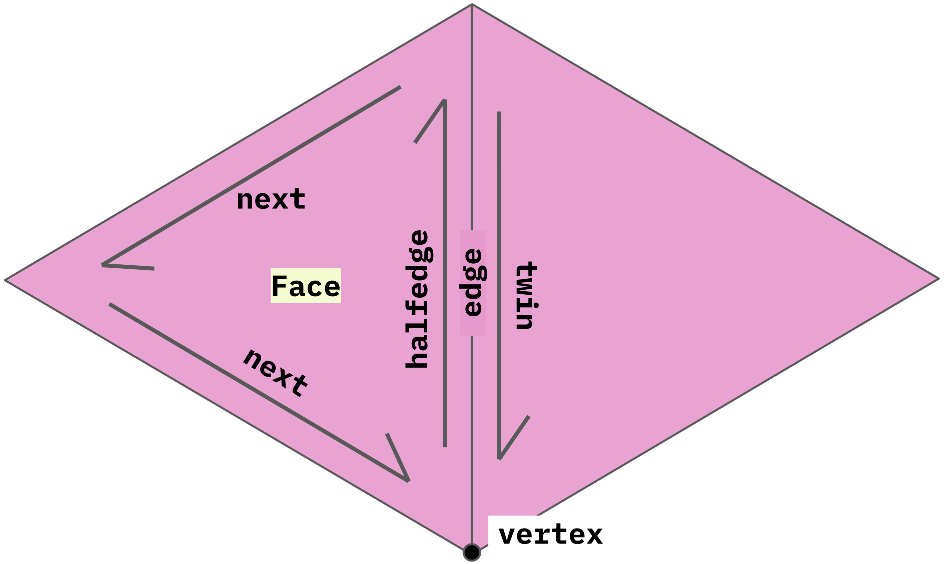
```
struct Halfedge {
        Halfedge *twin,
        Halfedge *next;
        Vertex *vertex;
        Edge *edge;
        Face *face;
}

struct Vertex {
        Point pt;
        Halfedge *halfedge;
}

struct Edge {
        Halfedge *halfedge;
}

struct Face {
        Halfedge *halfedge;
}
```

Key idea: two half edges act as **"glue"** between mesh elements



Each vertex, edge, and face points to one of its half edges

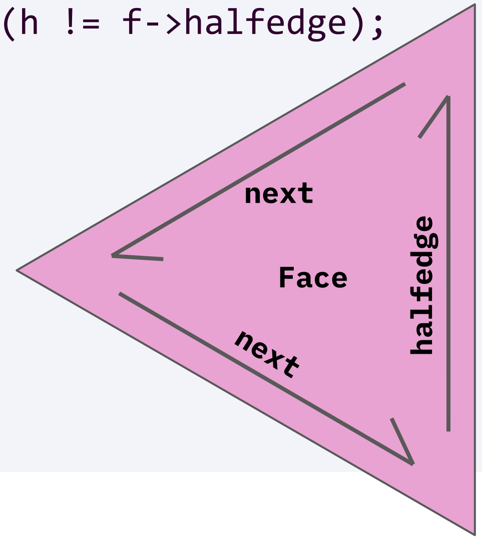
The Half-Edge Data Structure & Mesh Traversal

Use **`twin`** and **`next`** pointers to move around the mesh

You can process vertex, edge, and/or face pointers

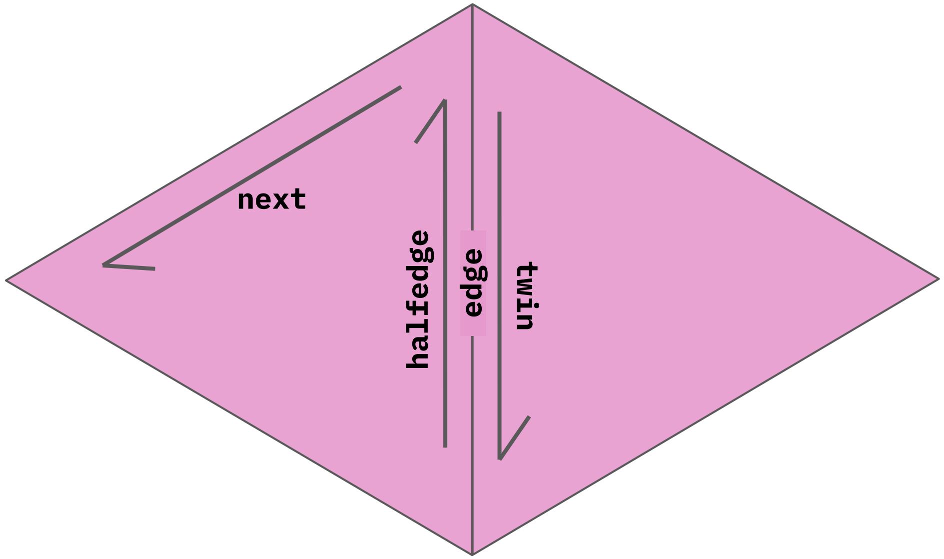
Example 1: Process all vertices of a face

```
Halfedge *h = f->halfedge;
do {
    process(h->vertex);
    h = h->next;
} while (h != f->halfedge);
```



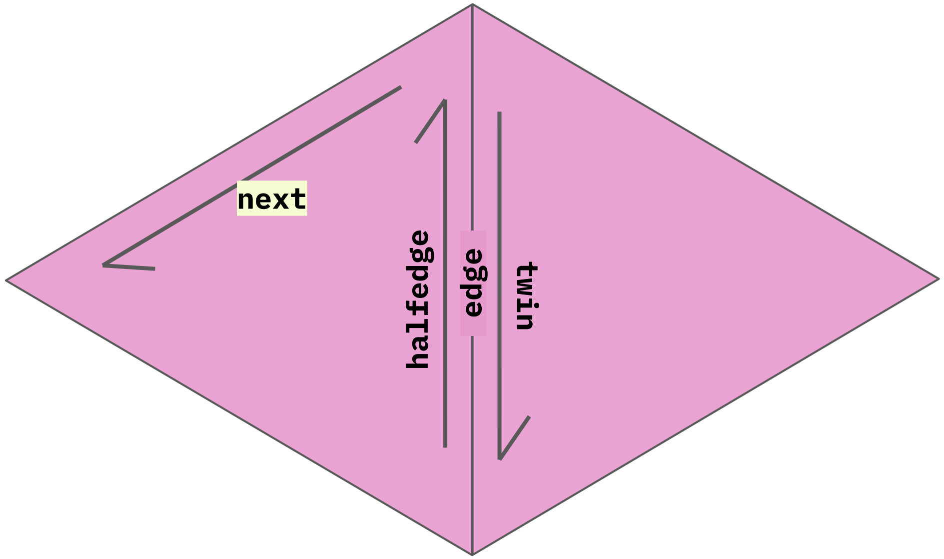
Mesh Traversal

- Use `twin` and `next` pointers to move around the mesh.



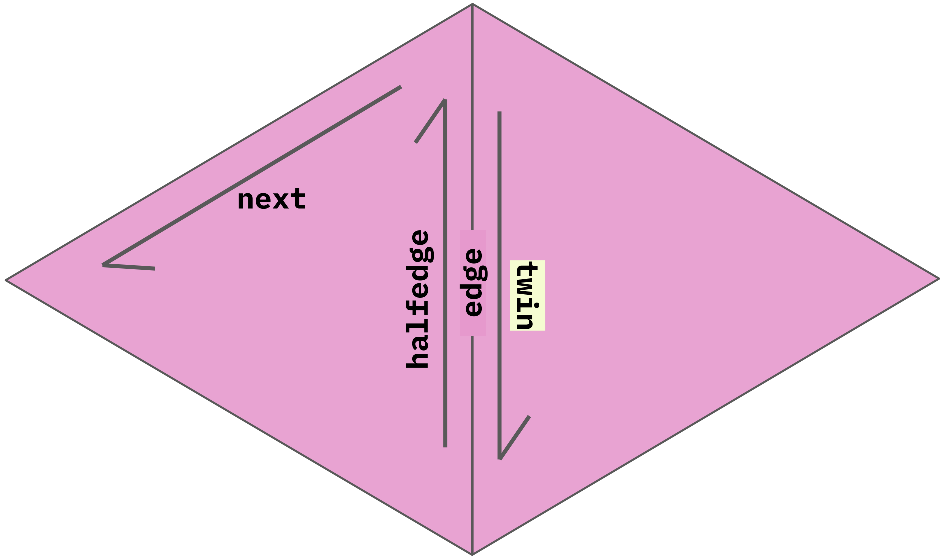
Mesh Traversal

- Use `twin` and `next` pointers to move around the mesh.
- `h->next()` to access the halfedge <u>ahead</u> of `h`.



Mesh Traversal

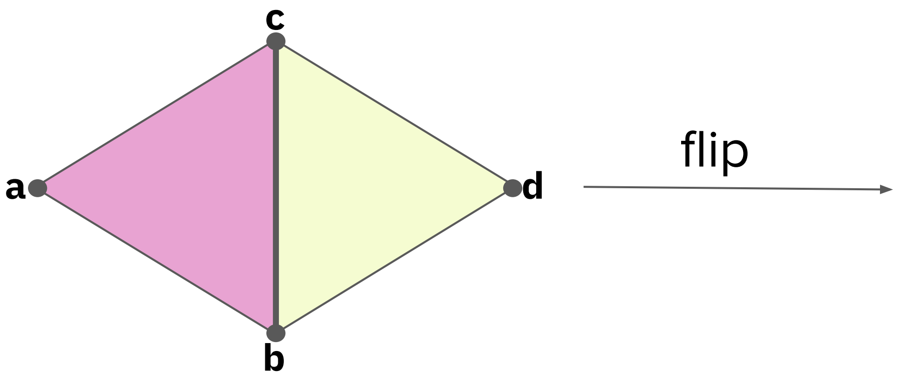
- Use `twin` and `next` pointers to move around the mesh.
- `h->next()` to access the halfedge <u>ahead</u> of `h`, going <u>counterclockwise</u>.
- `h->twin()` to access the halfedge that <u>shares an edge</u> with `h`.



Local Operations

Local Operations - Edge Flip

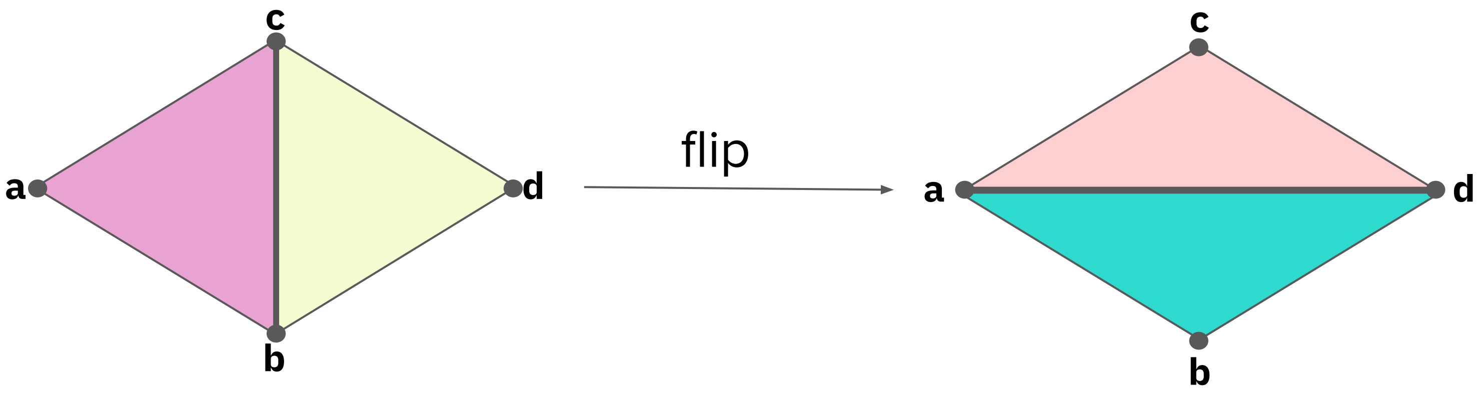
Triangles `(a, b, c)`, `(b, d, c)` become `(a, d, c)`, `(a, b, d)`:



- Long list of pointer reassignments
- However, no elements need to be created or destroyed

Local Operations - Edge Flip

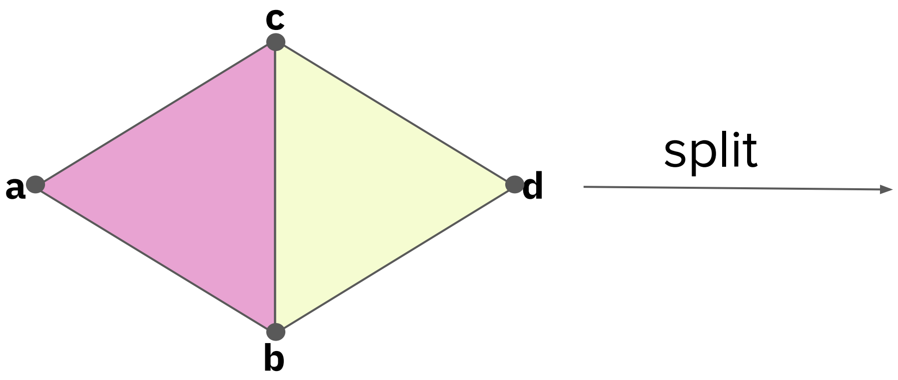
Triangles `(a, b, c)`, `(b, d, c)` become `(a, d, c)`, `(a, b, d)`:



- Long list of pointer reassignments
- However, no elements need to be created or destroyed

Local Operations - Edge Split

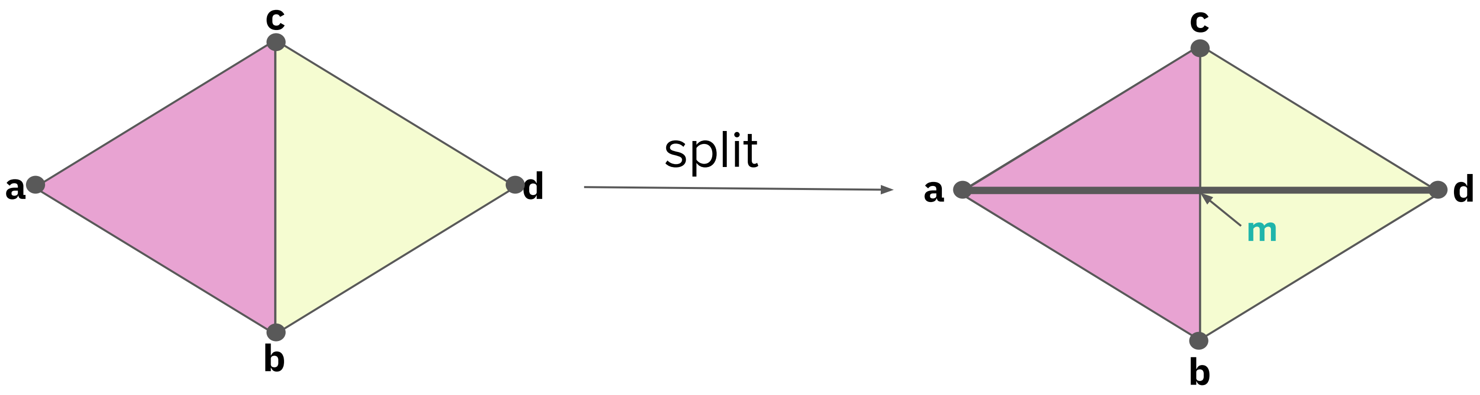
Insert midpoint `m` of edge `(c, b)`, connect to get four triangles:



- This time, you have to add elements
- Again, there are a lot of pointer reassignments you'll have to make!

Local Operations - Edge Split

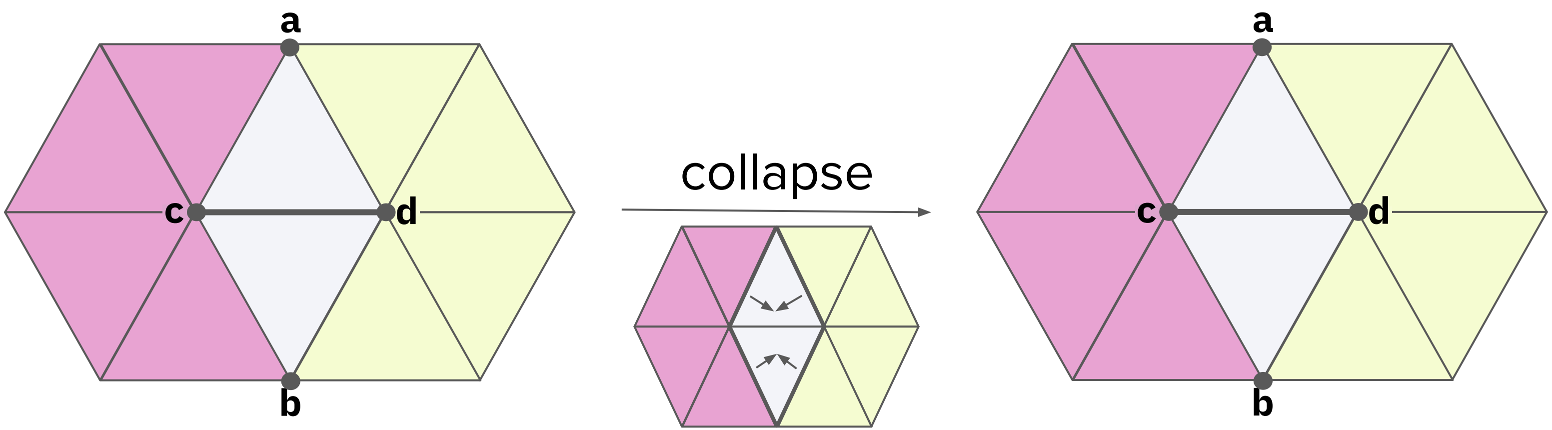
Insert midpoint `m` of edge **`(c, b)`**, connect to get four triangles:



- This time, you have to add elements
- Again, there are a lot of pointer reassignments you'll have to make!

Local Operations - Edge Collapse

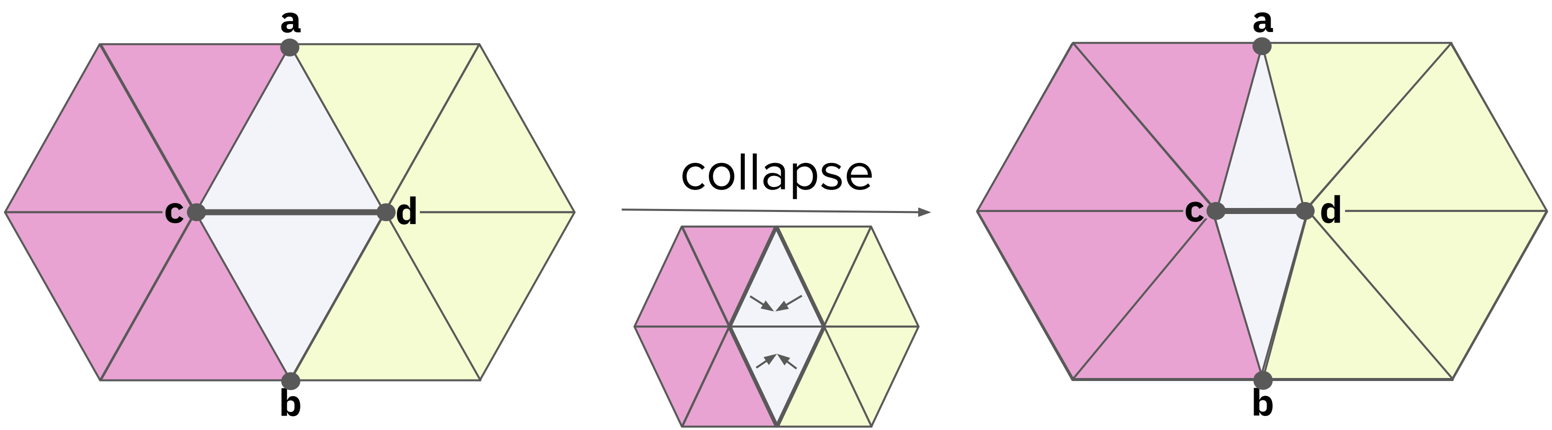
Replace edge `(c, d)` with a single vertex `m`:



- This time, you have to delete elements
- Again, there are a lot of pointer reassignments you'll have to make!

Local Operations - Edge Collapse

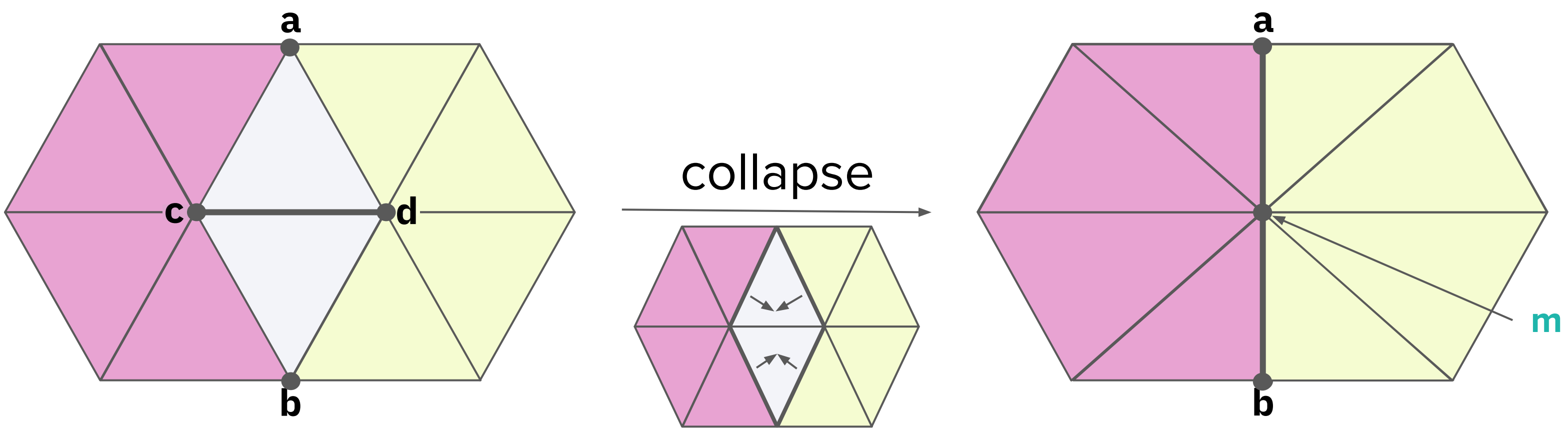
Replace edge `(c, d)` with a single vertex `m`:



- This time, you have to delete elements
- Again, there are a lot of pointer reassignments you'll have to make!

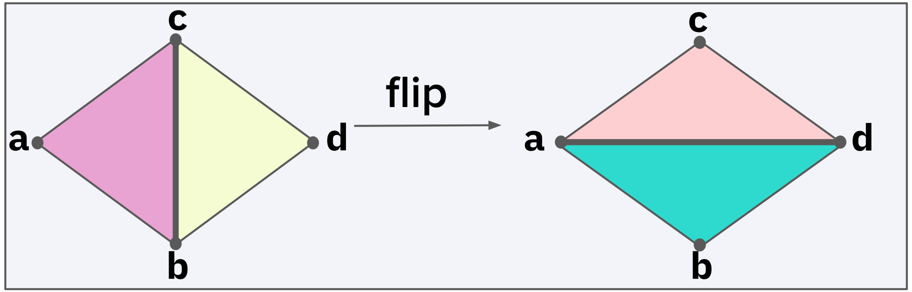
Local Operations - Edge Collapse

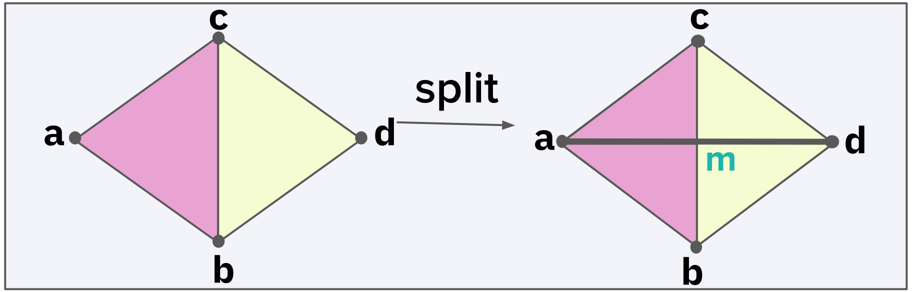
Replace edge `(c, d)` with a single vertex `m`:

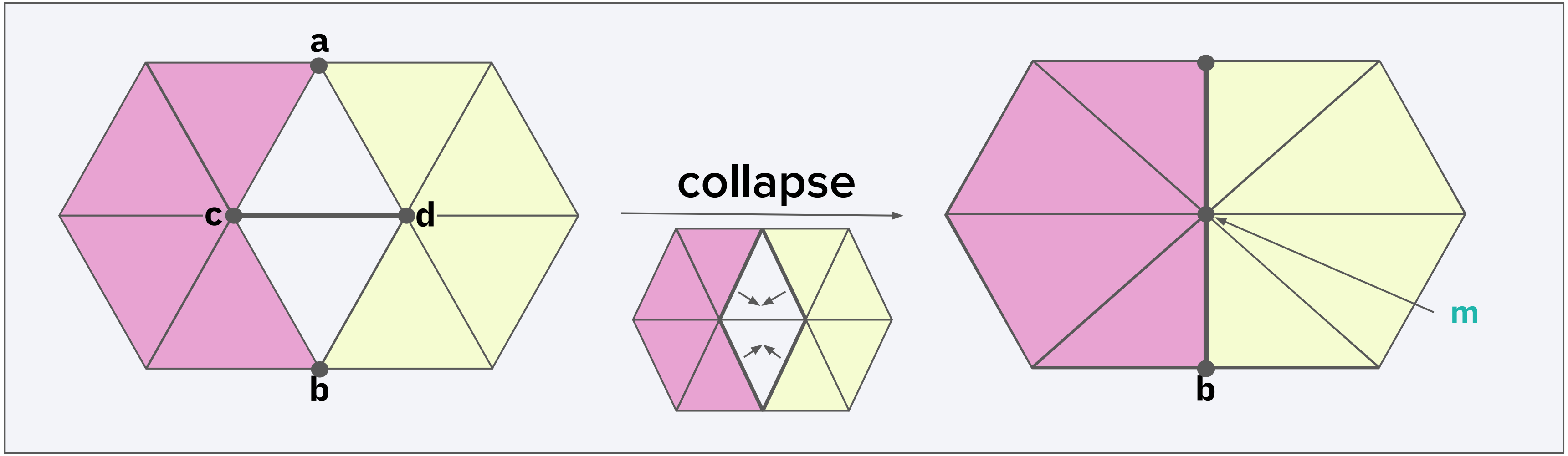


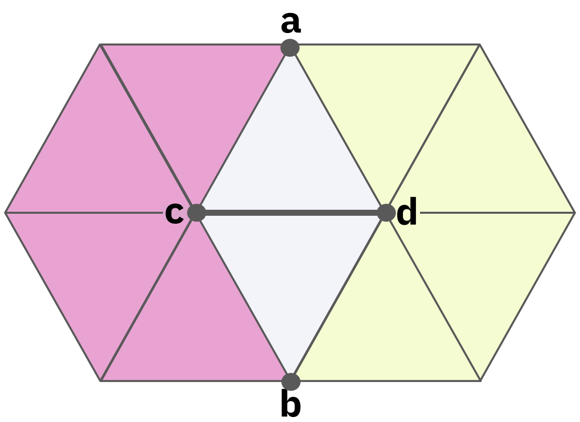
- This time, you have to delete elements
- Again, there are a lot of pointer reassignments you'll have to make!

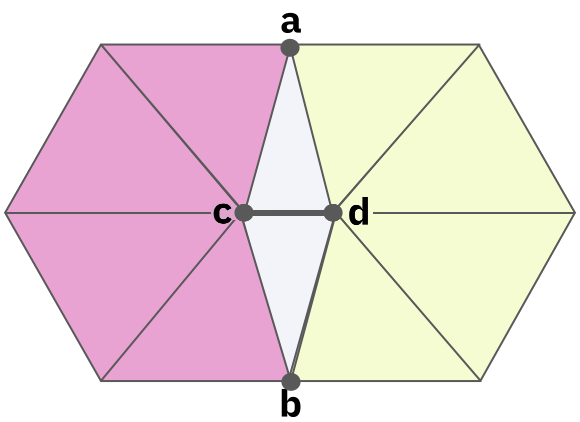
Local Operations

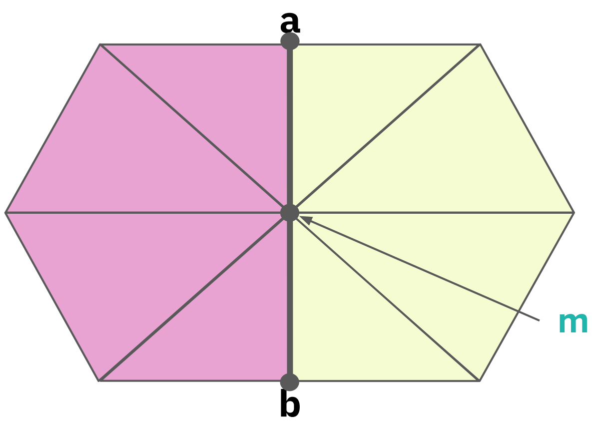







a
c
d
b

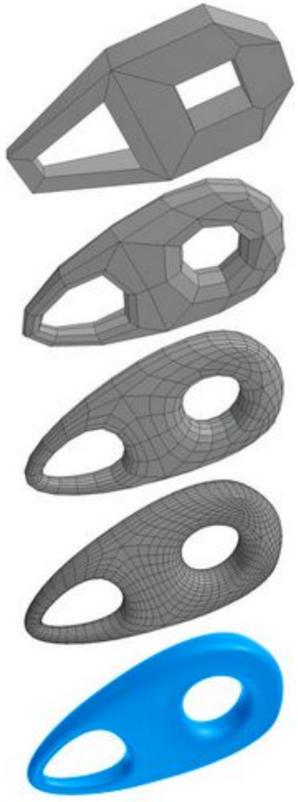

a
c
d
b


a
m
b

Subdivision

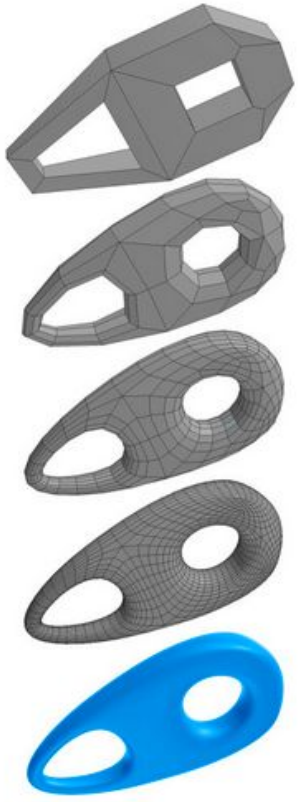
Subdivision Motivation

- It's expensive to create smooth meshes by hand.
- Instead…
    a. Start with control cage — a coarse mesh.
    b. Smooth algorithmically.

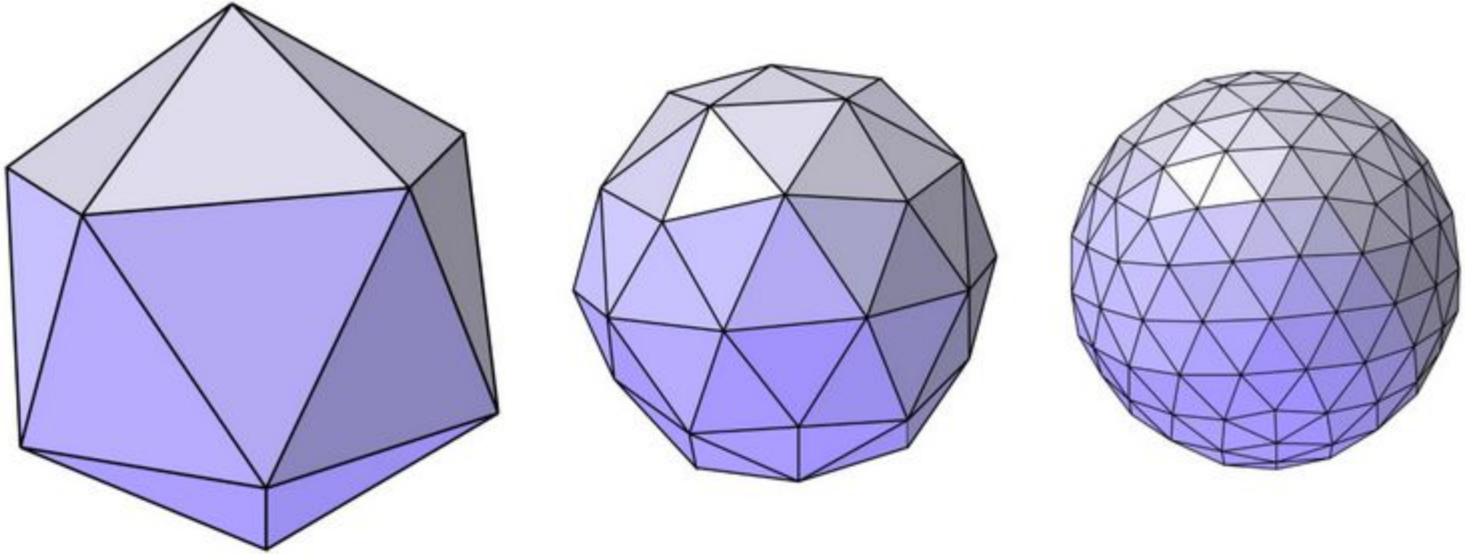


Subdivision Motivation

- It's expensive to create smooth meshes by hand.
- Instead...
  a. Start with control cage — a coarse mesh.
  b. Smooth algorithmically.
- Techniques:
  - Loop subdivision ➔ triangular meshes.
  - Catmull-Rom subdivision ➔ quad meshes.



Loop Subdivision Example



Simon Fuhrman

Loop Subdivision

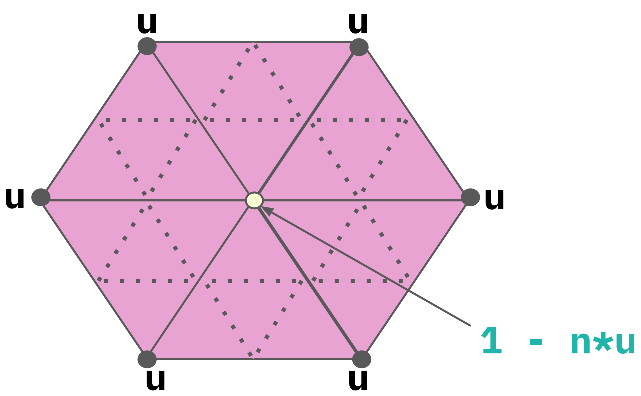
1. Split each triangle face into four ➔ new triangles, new vertices!

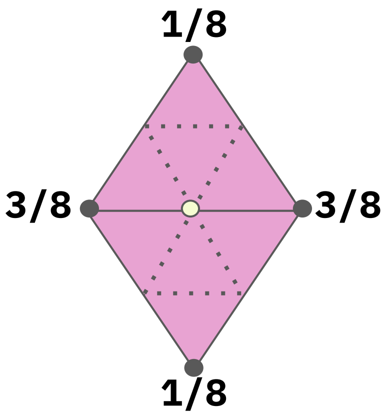
Loop Subdivision

1. Split each triangle face into four → new triangles, new vertices!
2. Update *old* and *new* vertex positions as weighted sum.

```
n: vertex degree

u: 3/16 if n = 3,
3/(8n) otherwise
```





Old vertices

$$v'_{old} = (1 - nu)v_{old} + \sum_{v_j \in N(v_{old})} uv_j$$

New vertices

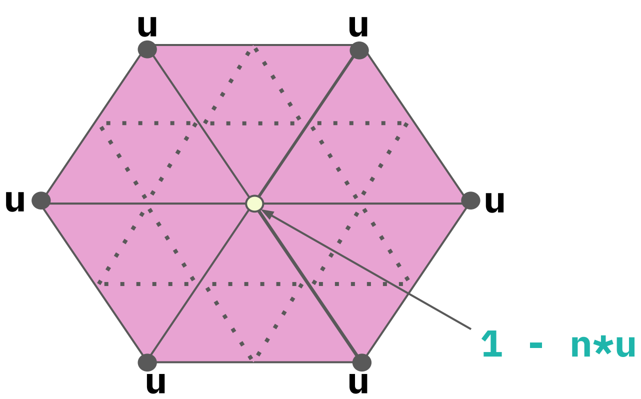
$$v'_{new} = \frac{3}{8}(v_{left} + v_{right}) + \frac{1}{8}(v_{up} + v_{down})$$

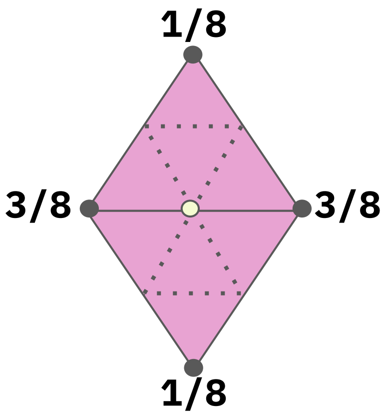
Loop Subdivision

1. Split each triangle face into four → new triangles, new vertices!
2. Update *old* and *new* vertex positions as <u>weighted sum</u>.

```
n: vertex degree

u: 3/16 if n = 3,
3/(8n) otherwise
```





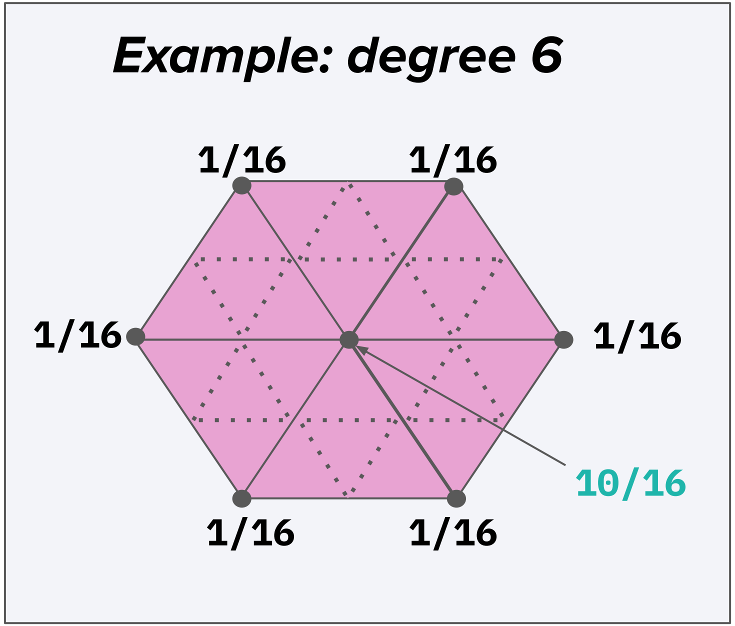
Old vertices

$$v'_{old} = (1 - nu)v_{old} + \sum_{v_j \in N(v_{old})} uv_j$$

New vertices

$$v'_{new} = \frac{3}{8}(v_{left} + v_{right}) + \frac{1}{8}(v_{up} + v_{down})$$

Loop Subdivision



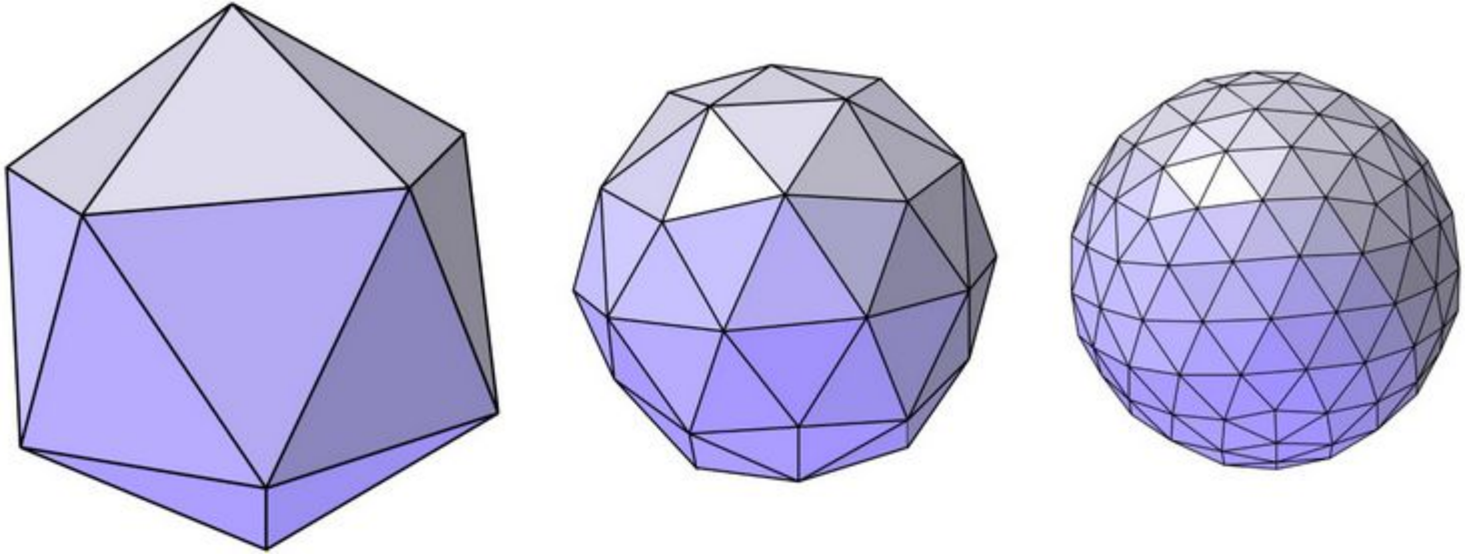
N = 6: vertex degree

u: 3/16 if n = 3, 3/(8n) otherwise

$$v'_{old} = (1 - nu)v_{old} + \sum_{v_j \in N(v_{old})} uv_j$$

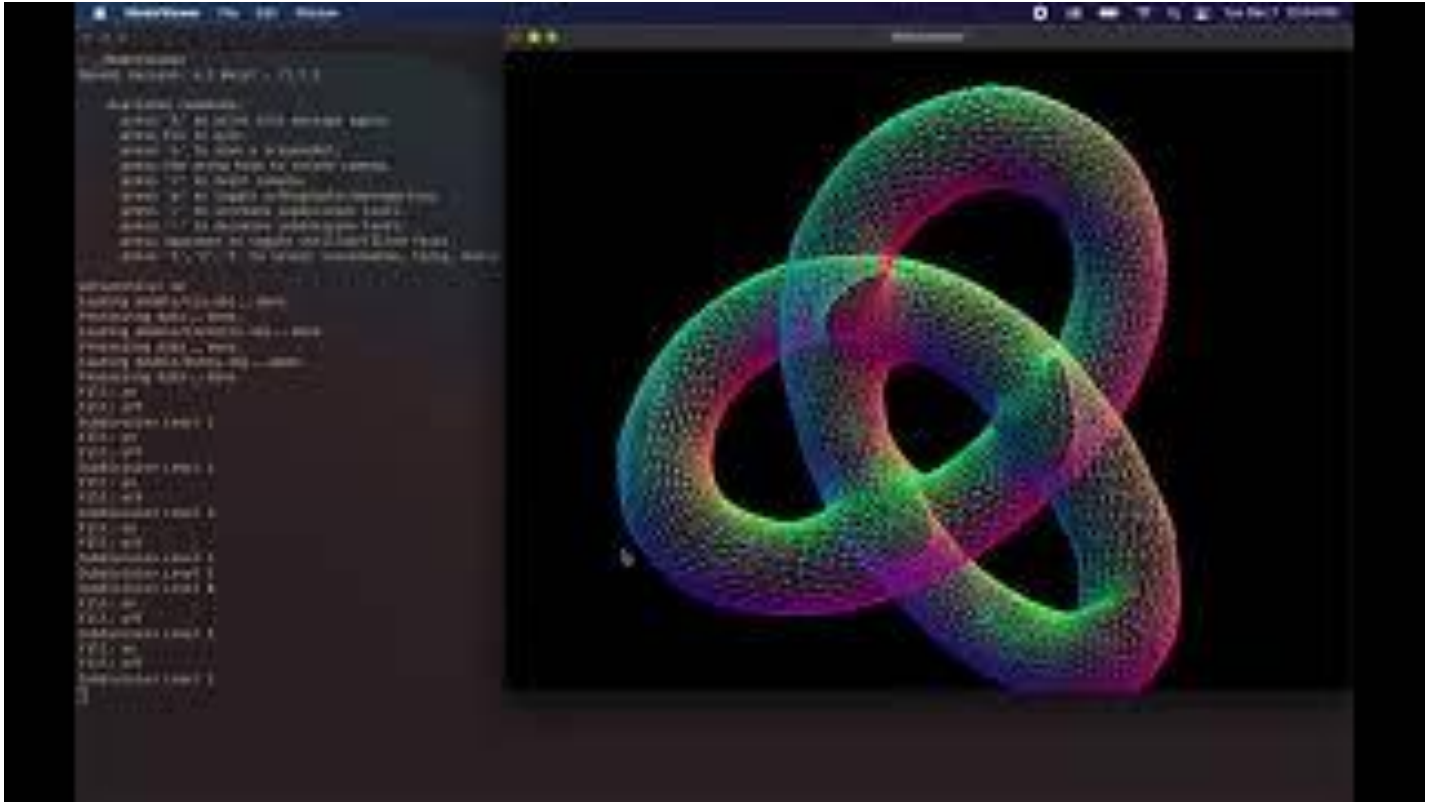
$$v'_{old} = (\frac{10}{16})v_{old} + \frac{1}{16}\sum_{j=1}^{6} v_j$$

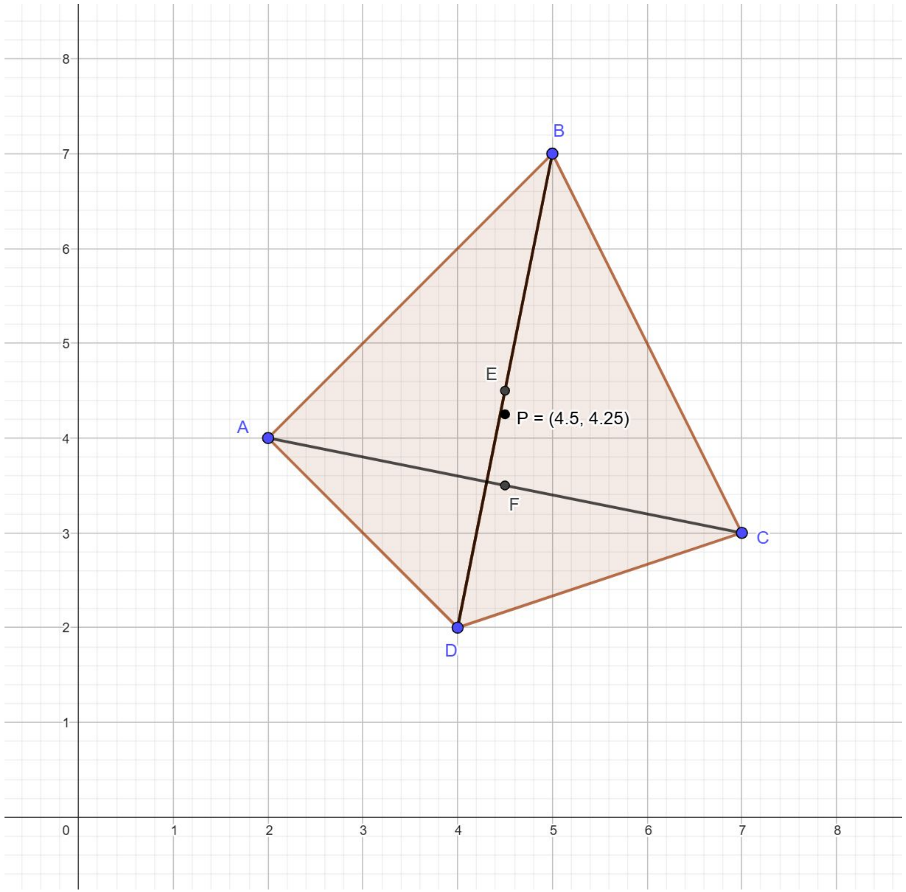
Loop Subdivision Example



Simon Fuhrman

Loop Subdivision Example





https://geogebra.org/classic/dkcv5cs8

Loop subdivision was developed at Pixar!

- https://graphics.pixar.com/library/SEC/supplemental.pdf

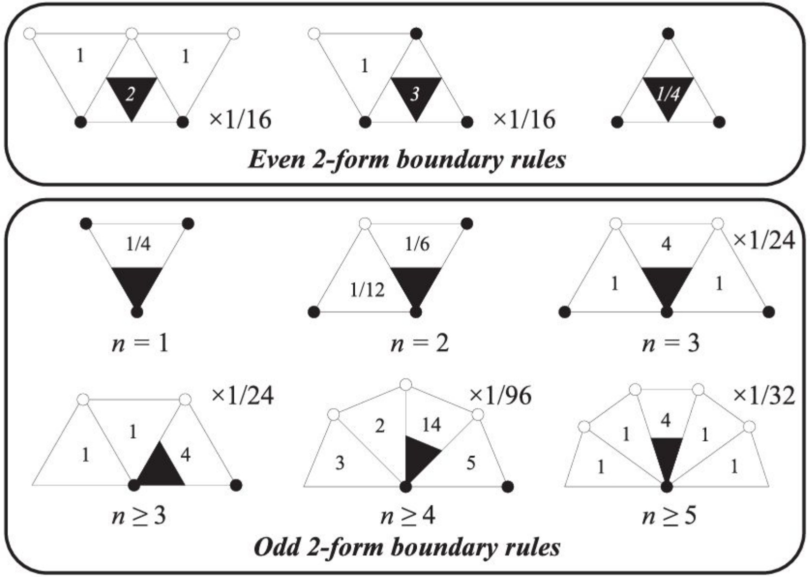
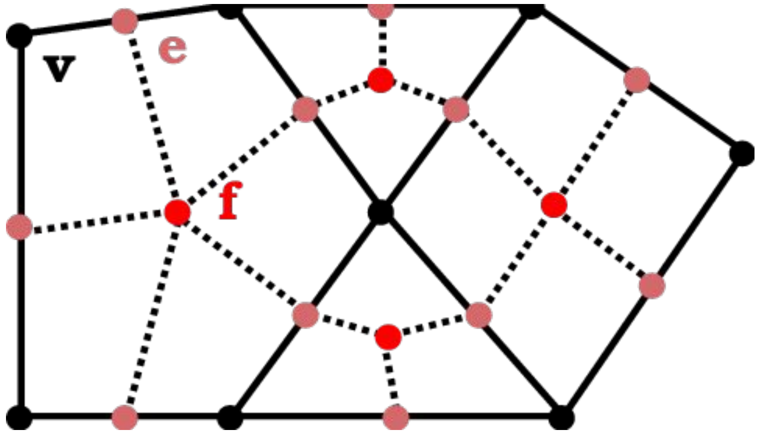


Figure 3: *Loop subdivision rules for 2-forms.*

Catmull-Clark Subdivision

Designed for meshes with variable polygons (triangles/quadrilaterals/pentagons)



Procedure:

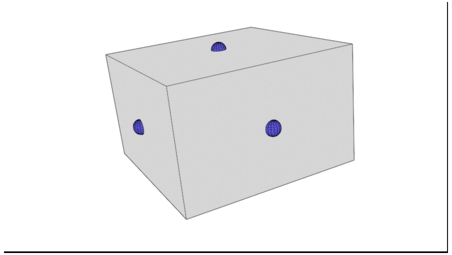
1. Add vertex in each face
2. Add midpoint to each edge
3. Connect all new vertices
4. Adjust vertex positions to weighted average

Catmull-Clark Subdivision

Designed for meshes with variable polygons (triangles/quadrilaterals/pentagons)

1. **Add face point:**

For each face, add a face point and set its position to be the average of all original points in the same face.

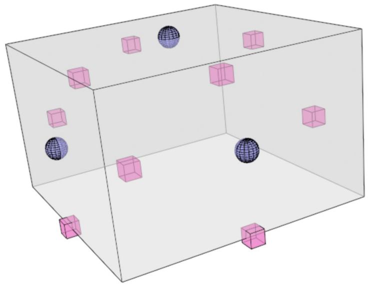


Catmull-Clark Subdivision

Designed for meshes with variable polygons (triangles/quadrilaterals/pentagons)

2. Add edge point:

For each edge, add an edge point and set its position to be the average of 2 neighboring face points (AF) and the midpoint of the edge (ME).



Catmull-Clark Subdivision

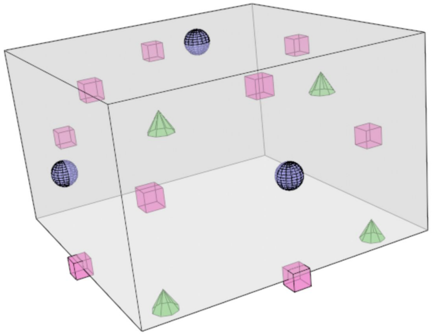
Designed for meshes with variable polygons (triangles/quadrilaterals/pentagons)

3. Move original vertices:

For each original vertex (P), take the average (F) of all *n* neighboring face points, and the average (R) of all *n* midpoints on neighboring edges (Note: edge midpoint is not the same as edge point!)

Move each vertex to

$$\frac{F + 2R + (n-3)P}{n}$$

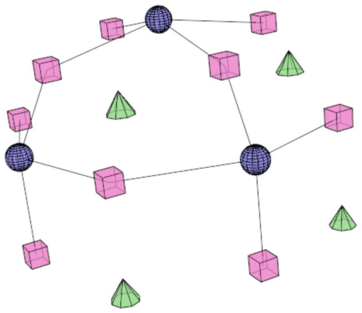


Catmull-Clark Subdivision

Designed for meshes with variable polygons (triangles/quadrilaterals/pentagons)

4. Form new edges and faces:

Connect each new face point to the new edge points of all original edges defining the original face

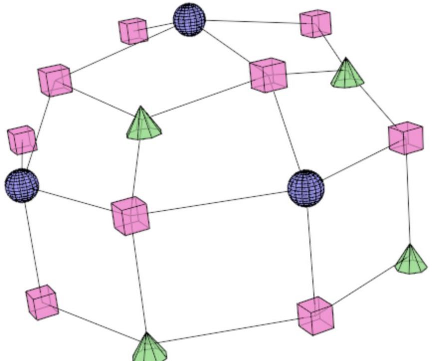


Catmull-Clark Subdivision

Designed for meshes with variable polygons (triangles/quadrilaterals/pentagons)

4. Form new edges and faces:

Connect each new vertex point to the new edge points of all original edges incident on the original vertex

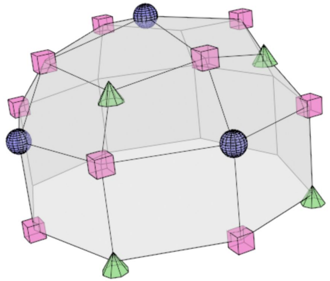


Catmull-Clark Subdivision

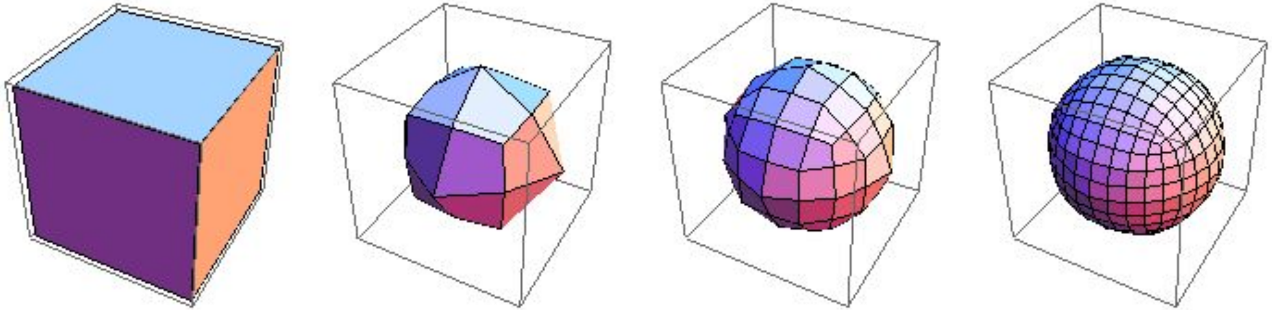
Designed for meshes with variable polygons (triangles/quadrilaterals/pentagons)

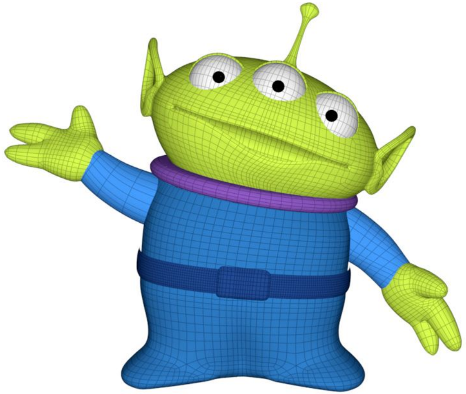
4. Form new edges and faces:

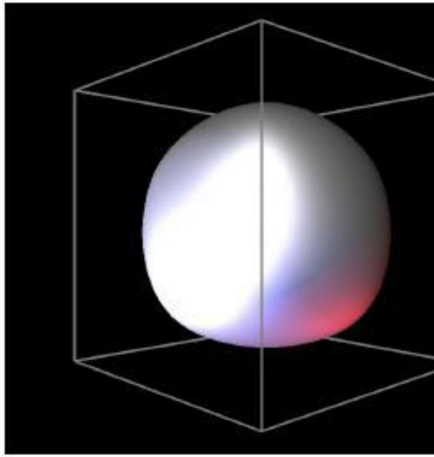
Define new faces as enclosed by the new edges.



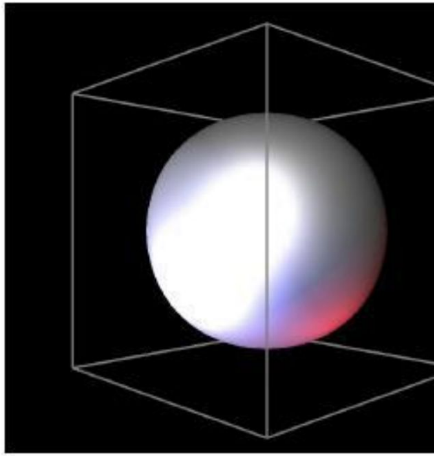
Catmull-Clark Subdivision Example







Loop

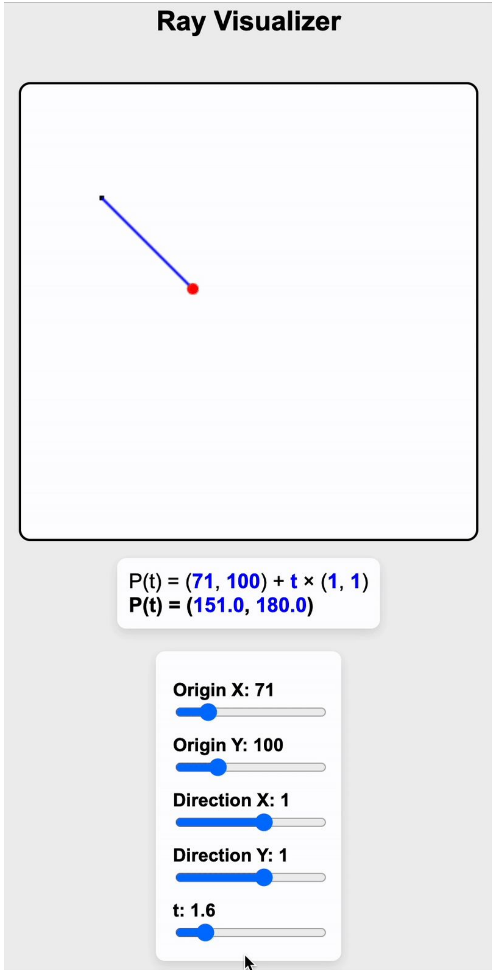


Ray Tracing Basics

Ray Equation

$$\boldsymbol{r}(t) = \boldsymbol{o} + t\boldsymbol{d}$$

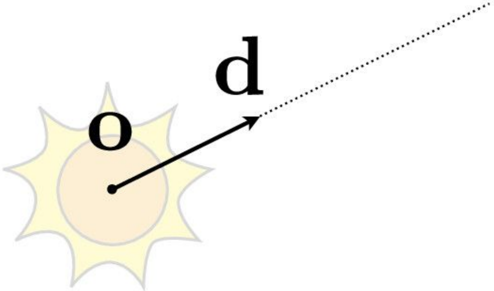
- Where $\boldsymbol{o}$ is the position of the <u>origin</u>.
- $\boldsymbol{d}$ is the <u>direction</u> of the ray.
- t is <u>time</u>, and is ≥ 0.



Ray Equation

Ray is defined by its origin and a direction vector

Example:



Ray equation:

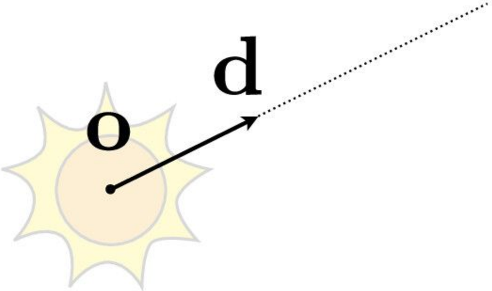
$$\mathbf{r}(t) = \mathbf{o} + t\mathbf{d} \qquad 0 \leq t < \infty$$

point along ray **"time"** **origin** **unit direction**

Ray Equation

Ray is defined by its origin and a direction vector

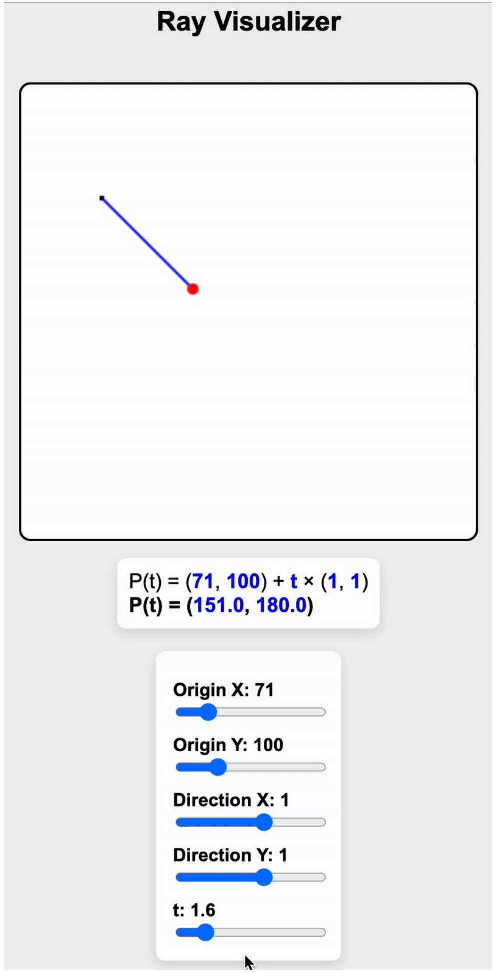
Example:



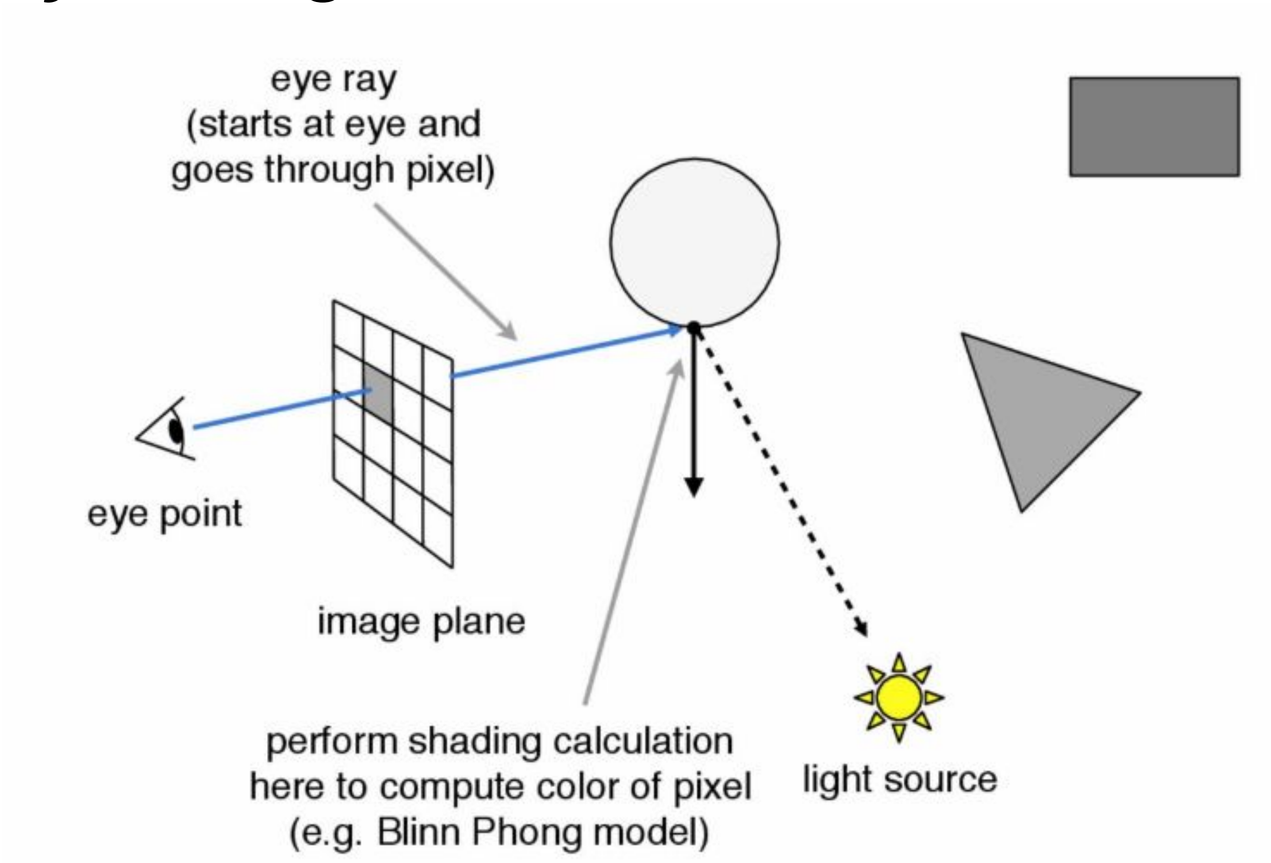
Ray equation:

$$\mathbf{r}(t) = \mathbf{o} + t\mathbf{d} \qquad 0 \leq t < \infty$$

point along ray "time" origin unit direction

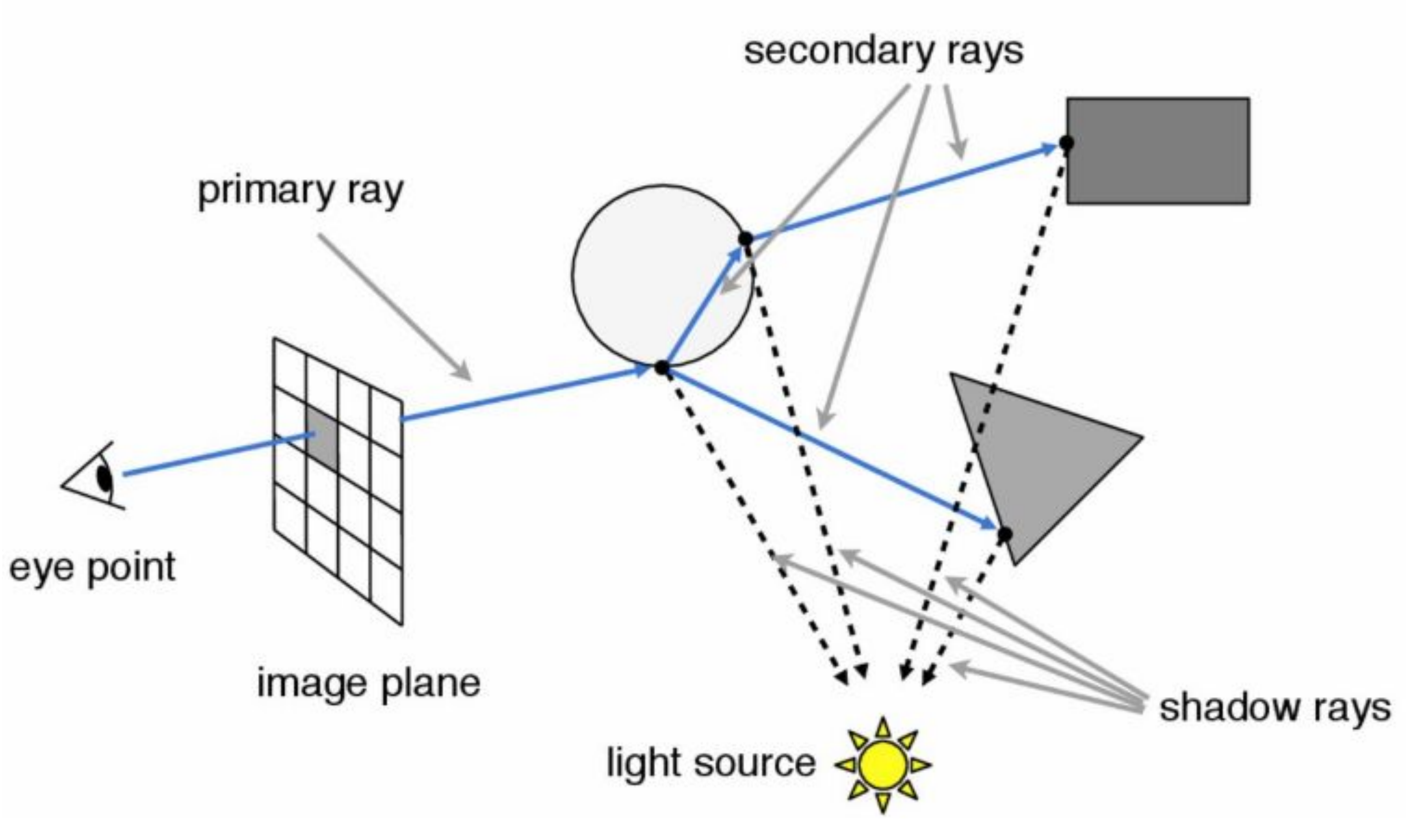


Ray Tracing



eye ray
(starts at eye and
goes through pixel)
eye point
image plane
perform shading calculation
here to compute color of pixel
(e.g. Blinn Phong model)
light source

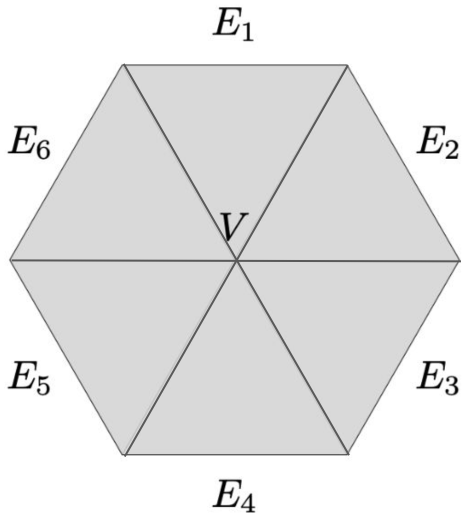
Ray Tracing


secondary rays
primary ray
eye point
image plane
shadow rays
light source

Worksheet Question 1.1 & 1.2

1. Starting from a given vertex, traverse the mesh and return a `std::vector` containing all of the edges that are opposite that vertex. In the diagram below, $E_1, \ldots, E_6$ are the edges opposite V. **Hint:** Start with the vertex's halfedge, then perform a do-while loop until we return to the original halfedge.

```
std::vector<EdgeIter> getOppositeEdges(VertexIter v) {
```

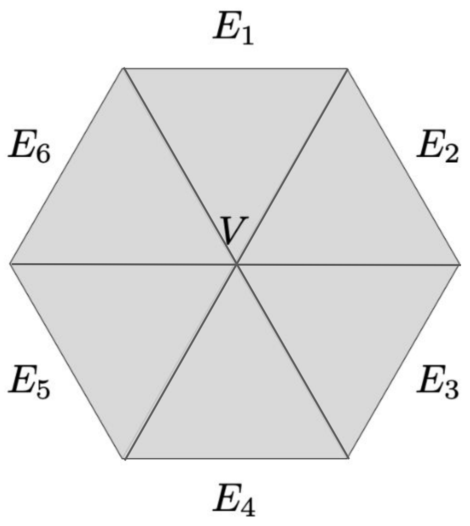


Note: We denote e.g. a pointer to an Edge by `EdgeIter`. So the following initialization is valid, given `EdgeIter e`:

```
HalfedgeIter h = e→halfedge();
```

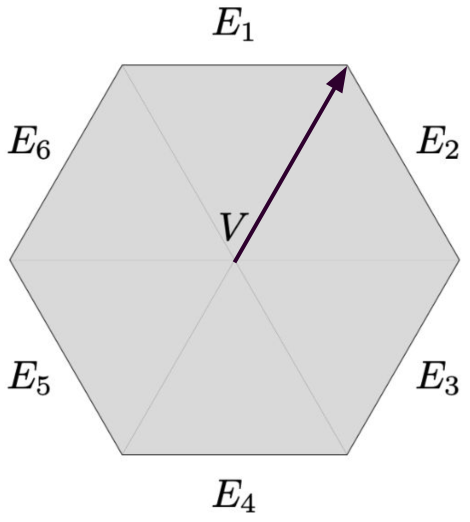
1. Starting from a given vertex, traverse the mesh and return a `std::vector` containing all of the edges that are opposite that vertex. In the diagram below, $E_1, \ldots, E_6$ are the edges opposite V. **Hint:** Start with the vertex's halfedge, then perform a do-while loop until we return to the original halfedge.

```
std::vector<EdgeIter> getOppositeEdges(VertexIter v)
{
    std::vector<EdgeIter> edges;
    HalfedgeIter h = v->halfedge();
    HalfedgeIter start_h = h;
    do {
        edges.push_back(h->next()->edge());
        h = h->next()->next()->twin();
    }  while (h != start_h);
    return edges;
}
```



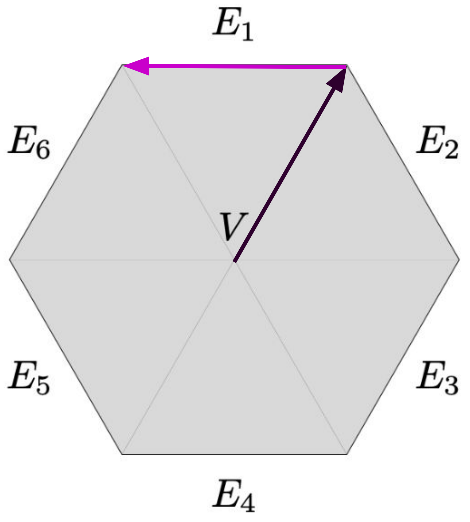
1. Starting from a given vertex, traverse the mesh and return a `std::vector` containing all of the edges that are opposite that vertex. In the diagram below, $E_1, \ldots, E_6$ are the edges opposite V. **Hint:** Start with the vertex's halfedge, then perform a do-while loop until we return to the original halfedge.

```
std::vector<EdgeIter> getOppositeEdges(VertexIter v)
{
    std::vector<EdgeIter> edges;
    HalfedgeIter h = v->halfedge();
    HalfedgeIter start_h = h;
    do {
        edges.push_back(h->next()->edge());
        h = h->next()->next()->twin();
    }  while (h != start_h);
    return edges;
}
```



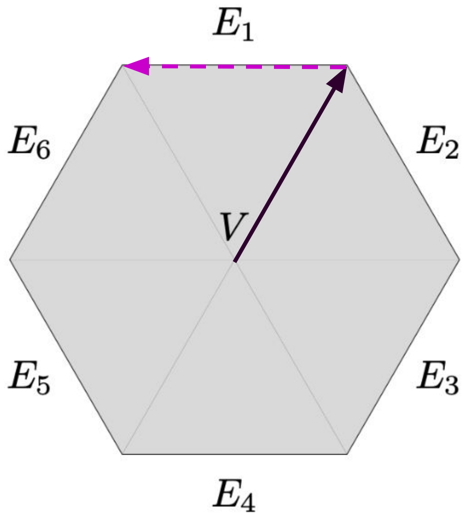
1. Starting from a given vertex, traverse the mesh and return a `std::vector` containing all of the edges that are opposite that vertex. In the diagram below, $E_1, \ldots, E_6$ are the edges opposite V. **Hint:** Start with the vertex's halfedge, then perform a do-while loop until we return to the original halfedge.

```
std::vector<EdgeIter> getOppositeEdges(VertexIter v)
{
    std::vector<EdgeIter> edges;
    HalfedgeIter h = v->halfedge();
    HalfedgeIter start_h = h;
    do {
        edges.push_back(h->next()->edge());
        h = h->next()->next()->twin();
    } while (h != start_h);
    return edges;
}
```



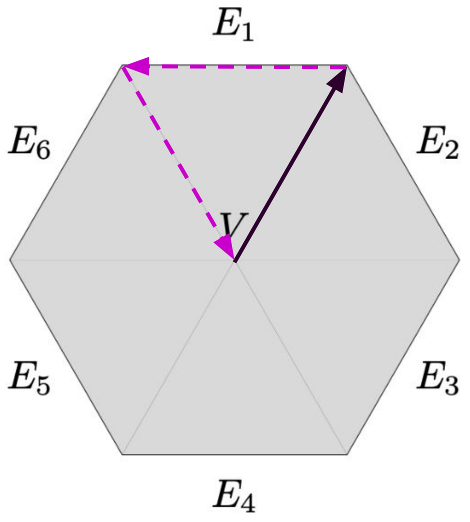
1. Starting from a given vertex, traverse the mesh and return a `std::vector` containing all of the edges that are opposite that vertex. In the diagram below, $E_1, \ldots, E_6$ are the edges opposite V. **Hint:** Start with the vertex's halfedge, then perform a do-while loop until we return to the original halfedge.

```
std::vector<EdgeIter> getOppositeEdges(VertexIter v)
{
    std::vector<EdgeIter> edges;
    HalfedgeIter h = v->halfedge();
    HalfedgeIter start_h = h;
    do {
        edges.push_back(h->next()->edge());
        h = h->next()->next()->twin();
    } while (h != start_h);
    return edges;
}
```



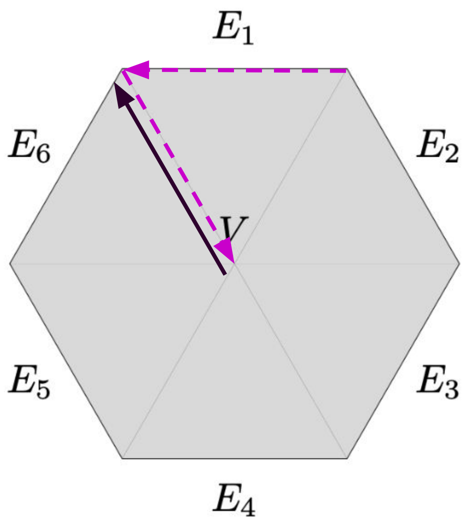
1. Starting from a given vertex, traverse the mesh and return a `std::vector` containing all of the edges that are opposite that vertex. In the diagram below, $E_1, \ldots, E_6$ are the edges opposite V. **Hint:** Start with the vertex's halfedge, then perform a do-while loop until we return to the original halfedge.

```
std::vector<EdgeIter> getOppositeEdges(VertexIter v)
{
    std::vector<EdgeIter> edges;
    HalfedgeIter h = v->halfedge();
    HalfedgeIter start_h = h;
    do {
        edges.push_back(h->next()->edge());
        h = h->next()->next()->twin();
    } while (h != start_h);
    return edges;
}
```



1. Starting from a given vertex, traverse the mesh and return a `std::vector` containing all of the edges that are opposite that vertex. In the diagram below, $E_1, \ldots, E_6$ are the edges opposite V. **Hint:** Start with the vertex's halfedge, then perform a do-while loop until we return to the original halfedge.

```
std::vector<EdgeIter> getOppositeEdges(VertexIter v)
{
    std::vector<EdgeIter> edges;
    HalfedgeIter h = v->halfedge();
    HalfedgeIter start_h = h;
    do {
        edges.push_back(h->next()->edge());
        h = h->next()->next()->twin();
    } while (h != start_h);
    return edges;
}
```

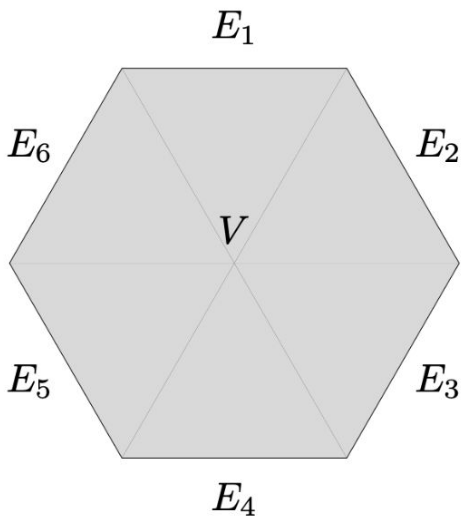


2. Let's write a function that locally attenuates noise in a mesh. Given a vertex v at position x, let

$$L(v) = \frac{1}{n} \sum_{v_j \in N(v)} x_j - x,$$

where $N(v)$ is the set of neighboring vertices of vertex v, x_j is the position of neighboring vertex v_j, and n is the number of neighboring vertices.

Apply $L(v)$ to slightly move v from position x to a new position x', making the mesh slightly smoother around v: $x' = x + kL(v)$, where k is a weight. We call this a `diffuse` operation on v's position.



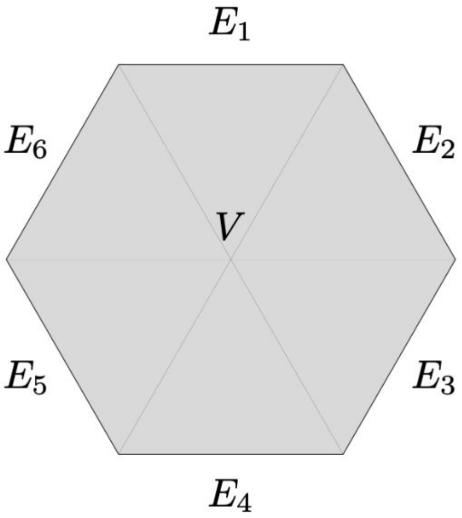
2. Let's write a function that locally attenuates noise in a mesh. Given a vertex v at position x, let

$$L(v) = \frac{1}{n} \sum_{v_j \in N(v)} x_j - x,$$

where $N(v)$ is the set of neighboring vertices of vertex v, x_j is the position of neighboring vertex v_j, and n is the number of neighboring vertices.

Apply $L(v)$ to slightly move v from position x to a new position x', making the mesh slightly smoother around v: $x' = x + kL(v)$, where k is a weight. We call this a `diffuse` operation on v's position.

```
void diffuse(VertexIter v, float k)
{
  Vector3D L(0, 0, 0);
  HalfedgeIter h = v->halfedge();
  HalfedgeIter start_h = h;
  int n = 0;
  do {
      VertexIter v_j = h->next()->vertex();
      Vector3D dir = v_j->position() - v->position();
      L = L + dir;
      n++;
      h = h->twin()->next()
  }  while (h != start_h);
  v->position() = v->position() + k * L / n;
}
```



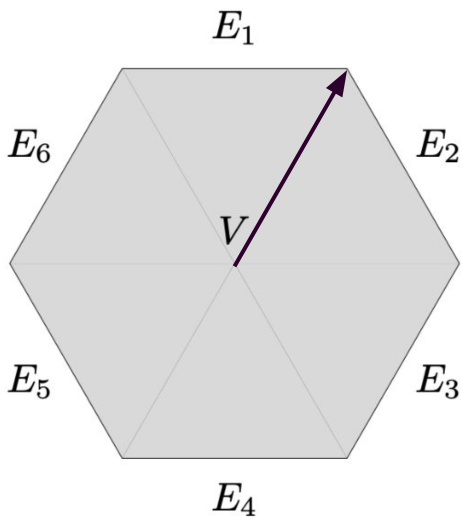
2. Let's write a function that locally attenuates noise in a mesh. Given a vertex v at position x, let

$$L(v) = \frac{1}{n} \sum_{v_j \in N(v)} x_j - x,$$

where $N(v)$ is the set of neighboring vertices of vertex v, x_j is the position of neighboring vertex v_j, and n is the number of neighboring vertices.

Apply $L(v)$ to slightly move v from position x to a new position x', making the mesh slightly smoother around v: $x' = x + kL(v)$, where k is a weight. We call this a `diffuse` operation on v's position.

```
void diffuse(VertexIter v, float k)
{
  Vector3D L(0, 0, 0);
  HalfedgeIter h = v->halfedge();
  HalfedgeIter start_h = h;
  int n = 0;
  do {
       VertexIter v_j = h->next()->vertex();
       Vector3D dir = v_j->position() - v->position();
       L = L + dir;
       n++;
       h = h->twin()->next()
  }  while (h != start_h);
  v->position() = v->position() + k * L / n;
}
```



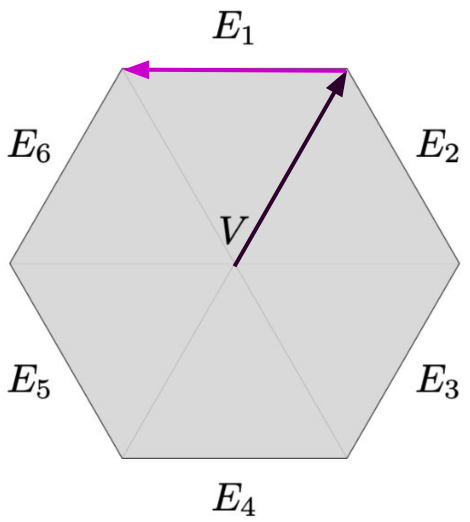
2. Let's write a function that locally attenuates noise in a mesh. Given a vertex v at position x, let

$$L(v) = \frac{1}{n} \sum_{v_j \in N(v)} x_j - x,$$

where $N(v)$ is the set of neighboring vertices of vertex v, x_j is the position of neighboring vertex v_j, and n is the number of neighboring vertices.

Apply $L(v)$ to slightly move v from position x to a new position x', making the mesh slightly smoother around v: $x' = x + kL(v)$, where k is a weight. We call this a `diffuse` operation on v's position.

```
void diffuse(VertexIter v, float k)
{
  Vector3D L(0, 0, 0);
  HalfedgeIter h = v->halfedge();
  HalfedgeIter start_h = h;
  int n = 0;
  do {
       VertexIter v_j = h->next()->vertex();
       Vector3D dir = v_j->position() - v->position();
       L = L + dir;
       n++;
       h = h->twin()->next()
  }  while (h != start_h);
  v->position() = v->position() + k * L / n;
}
```



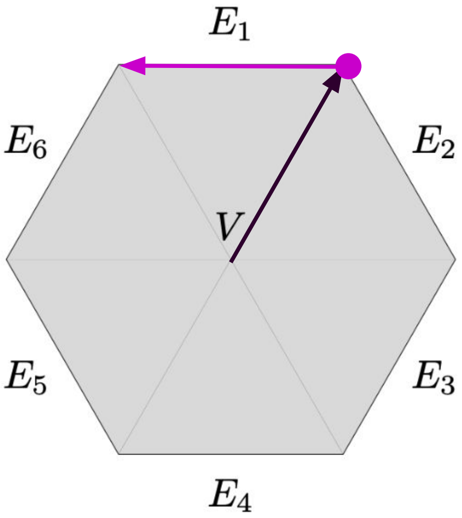
2. Let's write a function that locally attenuates noise in a mesh. Given a vertex v at position x, let

$$L(v) = \frac{1}{n} \sum_{v_j \in N(v)} x_j - x,$$

where $N(v)$ is the set of neighboring vertices of vertex v, x_j is the position of neighboring vertex v_j, and n is the number of neighboring vertices.

Apply $L(v)$ to slightly move v from position x to a new position x', making the mesh slightly smoother around v: $x' = x + kL(v)$, where k is a weight. We call this a `diffuse` operation on v's position.

```
void diffuse(VertexIter v, float k)
{
  Vector3D L(0, 0, 0);
  HalfedgeIter h = v->halfedge();
  HalfedgeIter start_h = h;
  int n = 0;
  do {
      VertexIter v_j = h->next()->vertex();
      Vector3D dir = v_j->position() - v->position();
      L = L + dir;
      n++;
      h = h->twin()->next()
  }  while (h != start_h);
  v->position() = v->position() + k * L / n;
}
```



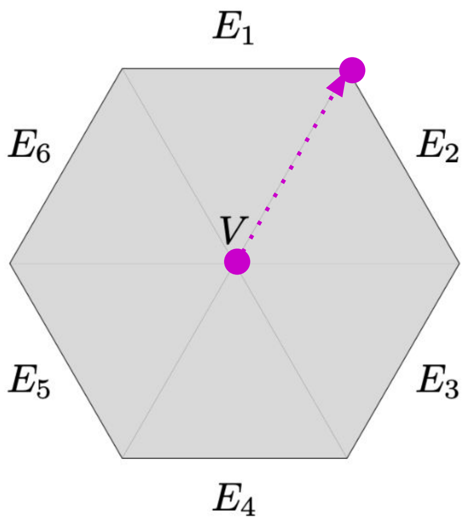
2. Let's write a function that locally attenuates noise in a mesh. Given a vertex v at position x, let

$$L(v) = \frac{1}{n} \sum_{v_j \in N(v)} x_j - x,$$

where $N(v)$ is the set of neighboring vertices of vertex v, x_j is the position of neighboring vertex v_j, and n is the number of neighboring vertices.

Apply $L(v)$ to slightly move v from position x to a new position x', making the mesh slightly smoother around v: $x' = x + kL(v)$, where k is a weight. We call this a `diffuse` operation on v's position.

```
void diffuse(VertexIter v, float k)
{
  Vector3D L(0, 0, 0);
  HalfedgeIter h = v->halfedge();
  HalfedgeIter start_h = h;
  int n = 0;
  do {
       VertexIter v_j = h->next()->vertex();
       Vector3D dir = v_j->position() - v->position();
       L = L + dir;
       n++;
       h = h->twin()->next()
  }  while (h != start_h);
  v->position() = v->position() + k * L / n;
}
```



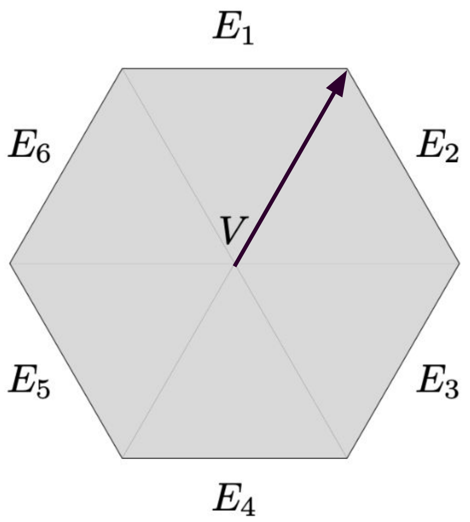
2. Let's write a function that locally attenuates noise in a mesh. Given a vertex v at position x, let

$$L(v) = \frac{1}{n} \sum_{v_j \in N(v)} x_j - x,$$

where $N(v)$ is the set of neighboring vertices of vertex v, x_j is the position of neighboring vertex v_j, and n is the number of neighboring vertices.

Apply $L(v)$ to slightly move v from position x to a new position x', making the mesh slightly smoother around v: $x' = x + kL(v)$, where k is a weight. We call this a `diffuse` operation on v's position.

```
void diffuse(VertexIter v, float k)
{
  Vector3D L(0, 0, 0);
  HalfedgeIter h = v->halfedge();
  HalfedgeIter start_h = h;
  int n = 0;
  do {
       VertexIter v_j = h->next()->vertex();
       Vector3D dir = v_j->position() - v->position();
       L = L + dir;
       n++;
       h = h->twin()->next()
  }  while (h != start_h);
  v->position() = v->position() + k * L / n;
}
```



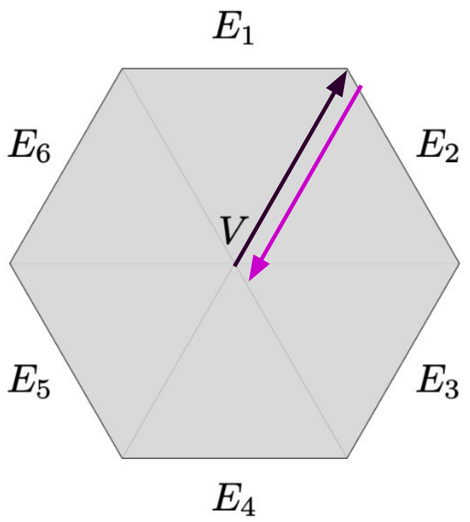
2. Let's write a function that locally attenuates noise in a mesh. Given a vertex v at position x, let

$$L(v) = \frac{1}{n} \sum_{v_j \in N(v)} x_j - x,$$

where $N(v)$ is the set of neighboring vertices of vertex v, x_j is the position of neighboring vertex v_j, and n is the number of neighboring vertices.

Apply $L(v)$ to slightly move v from position x to a new position x', making the mesh slightly smoother around v: $x' = x + kL(v)$, where k is a weight. We call this a `diffuse` operation on v's position.

```
void diffuse(VertexIter v, float k)
{
  Vector3D L(0, 0, 0);
  HalfedgeIter h = v->halfedge();
  HalfedgeIter start_h = h;
  int n = 0;
  do {
      VertexIter v_j = h->next()->vertex();
      Vector3D dir = v_j->position() - v->position();
      L = L + dir;
      n++;
      h = h->twin()->next()
  }  while (h != start_h);
  v->position() = v->position() + k * L / n;
}
```



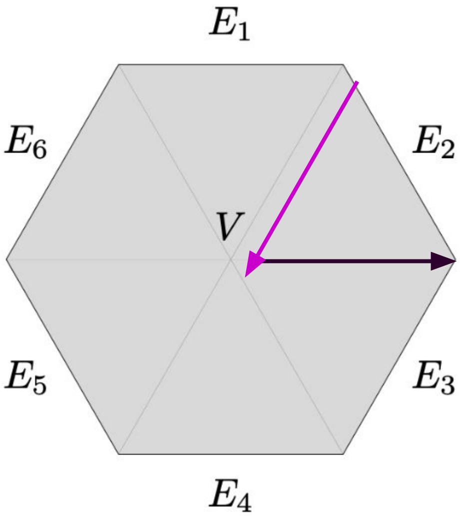
2. Let's write a function that locally attenuates noise in a mesh. Given a vertex v at position x, let

$$L(v) = \frac{1}{n} \sum_{v_j \in N(v)} x_j - x,$$

where $N(v)$ is the set of neighboring vertices of vertex v, x_j is the position of neighboring vertex v_j, and n is the number of neighboring vertices.

Apply $L(v)$ to slightly move v from position x to a new position x', making the mesh slightly smoother around v: $x' = x + kL(v)$, where k is a weight. We call this a `diffuse` operation on v's position.

```
void diffuse(VertexIter v, float k)
{
  Vector3D L(0, 0, 0);
  HalfedgeIter h = v->halfedge();
  HalfedgeIter start_h = h;
  int n = 0;
  do {
       VertexIter v_j = h->next()->vertex();
       Vector3D dir = v_j->position() - v->position();
       L = L + dir;
       n++;
       h = h->twin()->next()
  }  while (h != start_h);
  v->position() = v->position() + k * L / n;
}
```



Worksheet Question 1.3

3. Fill in the blanks so that the resulting procedure collapses the edge connecting vertices V_0 and V_2.

(i) e_{11}`->next()` = ____________________

(ii) f_3`->halfedge()` = ____________________

(iii) V_0`->halfedge()` = ____________________

(iv) e_3`->face()` = ____________________

(v) e_{12}`->vertex()` = ____________________

(vi) V_1`->halfedge()` = ____________________

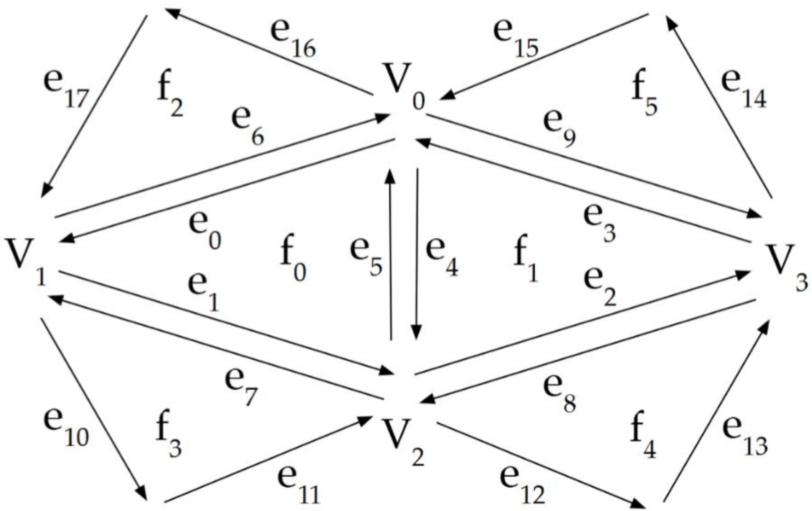
(vii) ____________________

(viii) ____________________

(ix) ____________________

(x) ____________________

(xi) ____________________



(xii) Delete vertex V_2

(xiii) Delete half-edges $e_1, e_2, e_4, e_5, e_7, e_8$

(xiv) Delete faces f_0, f_1

3. Fill in the blanks so that the resulting procedure collapses the edge connecting vertices V_0 and V_2.

(i) e_{11}`->next()` = ____________________

(ii) f_3`->halfedge()` = ____________________

(iii) V_0`->halfedge()` = ____________________

(iv) e_3`->face()` = ____________________

(v) e_{12}`->vertex()` = ____________________

(vi) V_1`->halfedge()` = ____________________

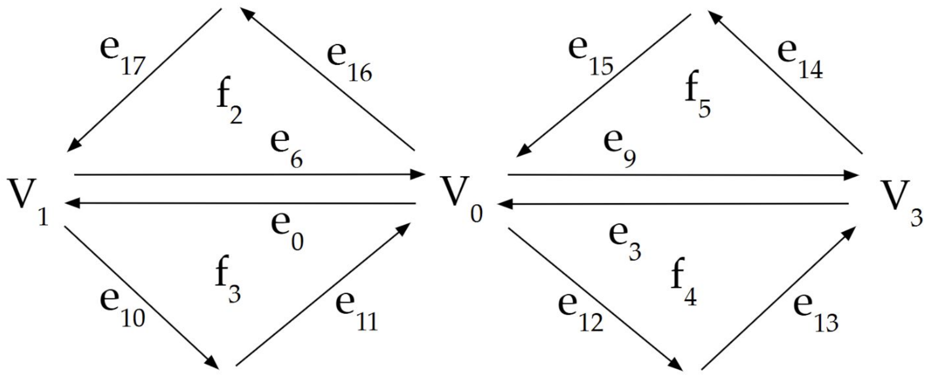
(vii) ____________________

(viii) ____________________

(ix) ____________________

(x) ____________________

(xi) ____________________



(xii) Delete vertex V_2

(xiii) Delete half-edges $e_1, e_2, e_4, e_5, e_7, e_8$

(xiv) Delete faces f_0, f_1

3. Fill in the blanks so that the resulting procedure collapses the edge connecting vertices V_0 and V_2.

(i) `e11->next() =` ___ e_0 ___

(ii) `f3->halfedge() =` ______________

(iii) `V0->halfedge() =` ______________

(iv) `e3->face() =` ______________

(v) `e12->vertex() =` ______________

(vi) `V1->halfedge() =` ______________

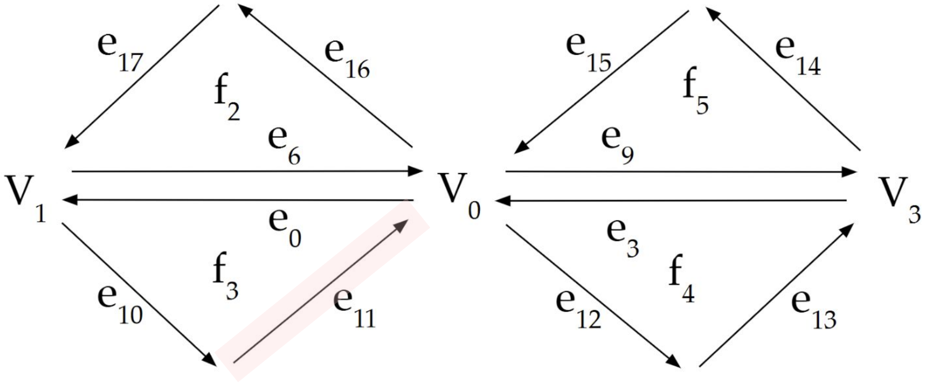
(vii) ______________

(viii) ______________

(ix) ______________

(x) ______________

(xi) ______________



(xii) Delete vertex V_2

(xiii) Delete half-edges $e_1, e_2, e_4, e_5, e_7, e_8$

(xiv) Delete faces f_0, f_1

3. Fill in the blanks so that the resulting procedure collapses the edge connecting vertices V_0 and V_2.

(i) e_{11}`->next()` = e_0

(ii) f_3`->halfedge()` = any of e_0, e_{10}, e_{11}

(iii) V_0`->halfedge()` = __________

(iv) e_3`->face()` = __________

(v) e_{12}`->vertex()` = __________

(vi) V_1`->halfedge()` = __________

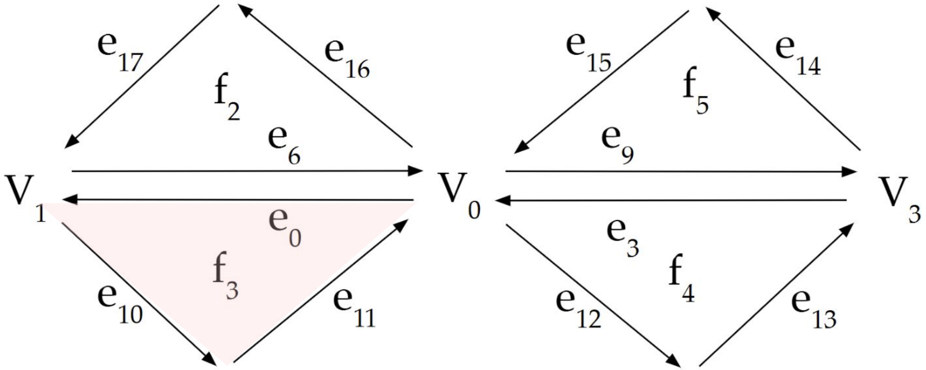
(vii) __________

(viii) __________

(ix) __________

(x) __________

(xi) __________



(xii) Delete vertex V_2

(xiii) Delete half-edges $e_1, e_2, e_4, e_5, e_7, e_8$

(xiv) Delete faces f_0, f_1

3. Fill in the blanks so that the resulting procedure collapses the edge connecting vertices V_0 and V_2.

(i) e_{11}`->next()` = e_0

(ii) f_3`->halfedge()` = any of e_0, e_{10}, e_{11}

(iii) V_0`->halfedge()` = any of e_0, e_9, e_{12}, e_{16}

(iv) e_3`->face()` = ______

(v) e_{12}`->vertex()` = ______

(vi) V_1`->halfedge()` = ______

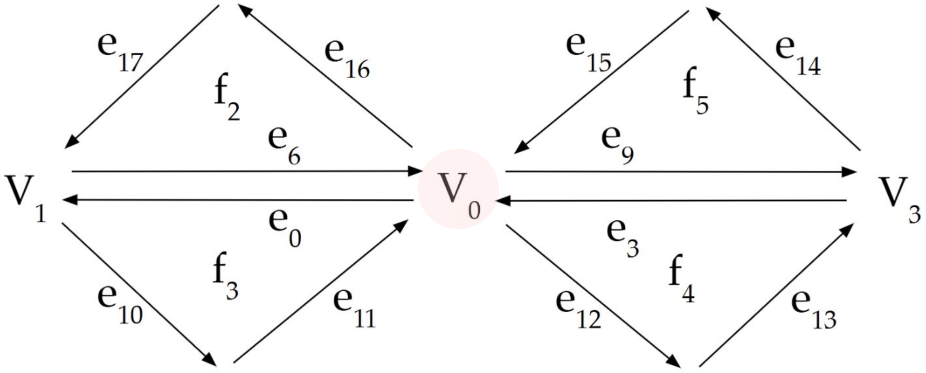
(vii) ______

(viii) ______

(ix) ______

(x) ______

(xi) ______



(xii) Delete vertex V_2

(xiii) Delete half-edges $e_1, e_2, e_4, e_5, e_7, e_8$

(xiv) Delete faces f_0, f_1

3. Fill in the blanks so that the resulting procedure collapses the edge connecting vertices V_0 and V_2.

(i) `e11->next()` = e_0

(ii) `f3->halfedge()` = any of e_0, e_{10}, e_{11}

(iii) `V0->halfedge()` = any of e_0, e_9, e_{12}, e_{16}

(iv) `e3->face()` = f_4, e_3->next() = e_{12}

(v) `e12->vertex()` = ______

(vi) `V1->halfedge()` = ______

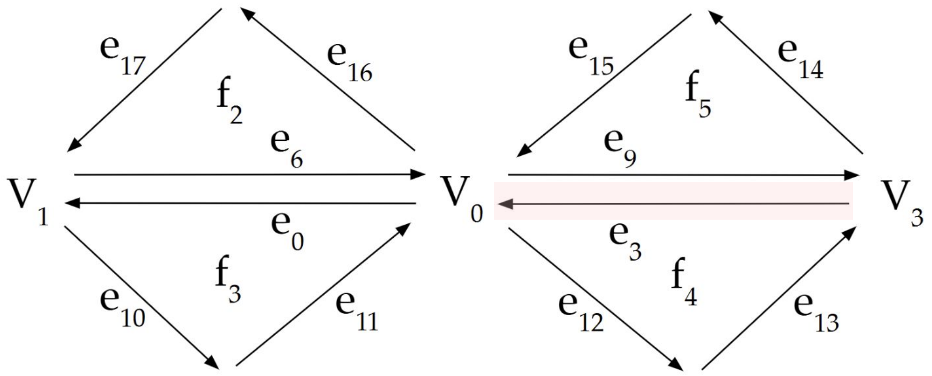
(vii) ______

(viii) ______

(ix) ______

(x) ______

(xi) ______



(xii) Delete vertex V_2

(xiii) Delete half-edges $e_1, e_2, e_4, e_5, e_7, e_8$

(xiv) Delete faces f_0, f_1

3. Fill in the blanks so that the resulting procedure collapses the edge connecting vertices V_0 and V_2.

(i) `e11->next()` = e_0

(ii) `f3->halfedge()` = any of e_0, e_{10}, e_{11}

(iii) `V0->halfedge()` = any of e_0, e_9, e_{12}, e_{16}

(iv) `e3->face()` = f_4, e_3->next() = e_{12}

(v) `e12->vertex()` = V_0

(vi) `V1->halfedge()` = ________

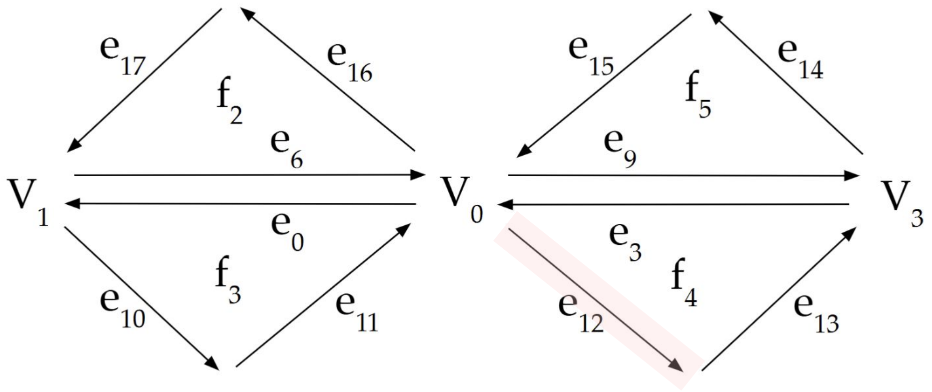
(vii) ________

(viii) ________

(ix) ________

(x) ________

(xi) ________



(xii) Delete vertex V_2

(xiii) Delete half-edges $e_1, e_2, e_4, e_5, e_7, e_8$

(xiv) Delete faces f_0, f_1

3. Fill in the blanks so that the resulting procedure collapses the edge connecting vertices V_0 and V_2.

(i) e_{11}`->next()` = e_0

(ii) f_3`->halfedge()` = any of e_0, e_{10}, e_{11}

(iii) V_0`->halfedge()` = any of e_0, e_9, e_{12}, e_{16}

(iv) e_3`->face()` = f_4, e_3->next() = e_{12}

(v) e_{12}`->vertex()` = V_0

(vi) V_1`->halfedge()` = either of e_6, e_{10}

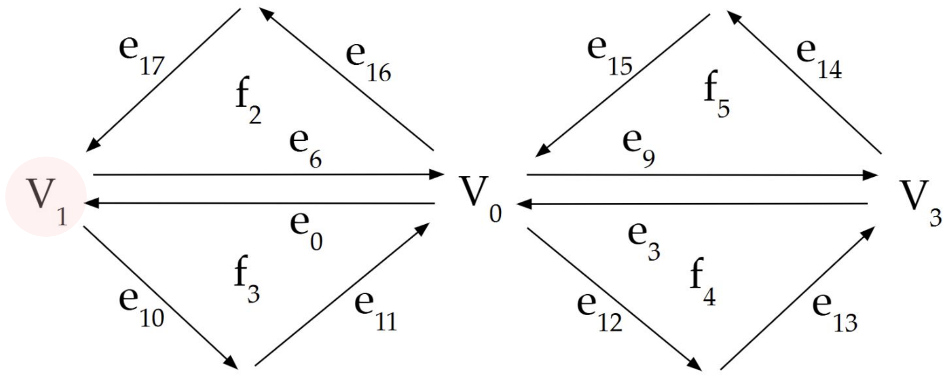
(vii) ________

(viii) ________

(ix) ________

(x) ________

(xi) ________



(xii) Delete vertex V_2

(xiii) Delete half-edges $e_1, e_2, e_4, e_5, e_7, e_8$

(xiv) Delete faces f_0, f_1

3. Fill in the blanks so that the resulting procedure collapses the edge connecting vertices V_0 and V_2.

(i) `e11->next() =` e_0

(ii) `f3->halfedge() =` any of e_0, e_{10}, e_{11}

(iii) `V0->halfedge() =` any of e_0, e_9, e_{12}, e_{16}

(iv) `e3->face() =` f_4, e_3->next() = e_{12}

(v) `e12->vertex() =` V_0

(vi) `V1->halfedge() =` either of e_6, e_{10}

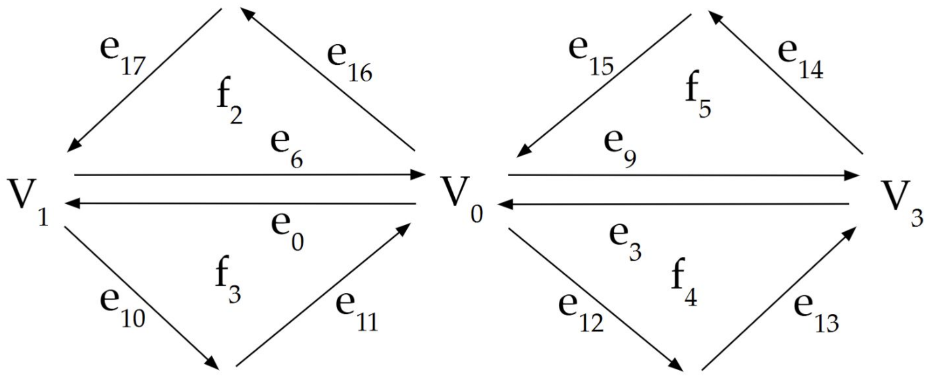
(vii) e_{13}->next() = e_3

(viii) f_4->halfedge() = any of e_3, e_{12}, e_{13}

(ix) e_0->face() = f_3

(x) e_0->next() = e_{10}

(xi) V_3->halfedge() = either of e_3, e_{14}

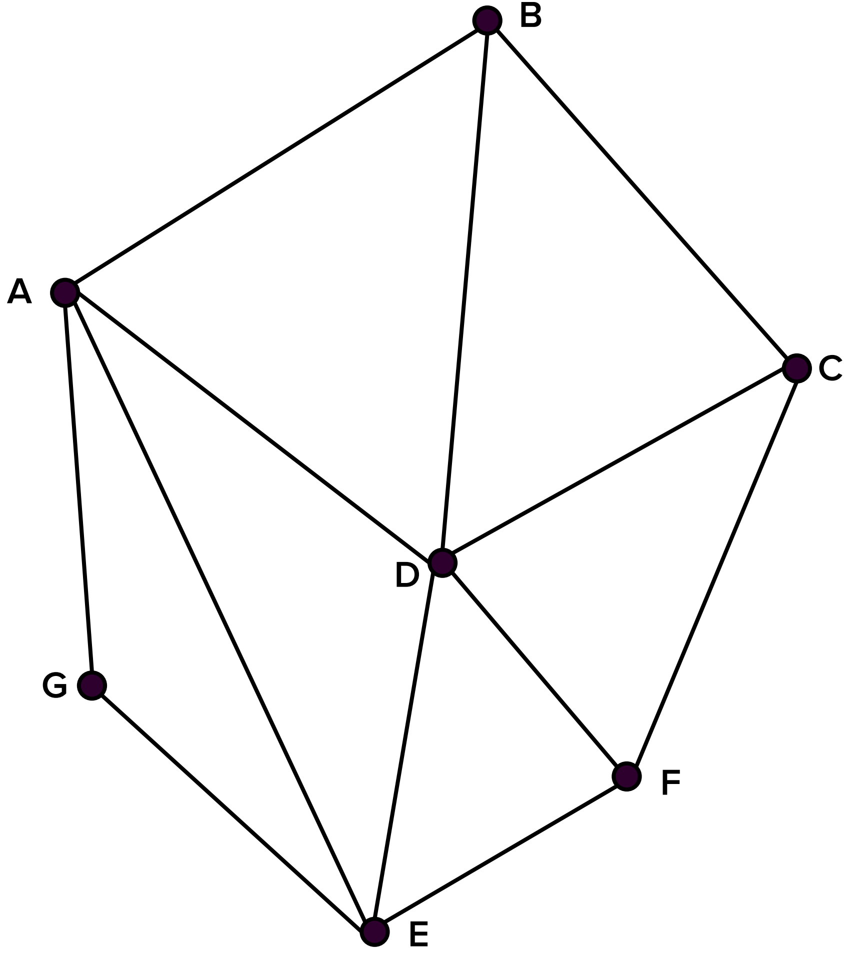


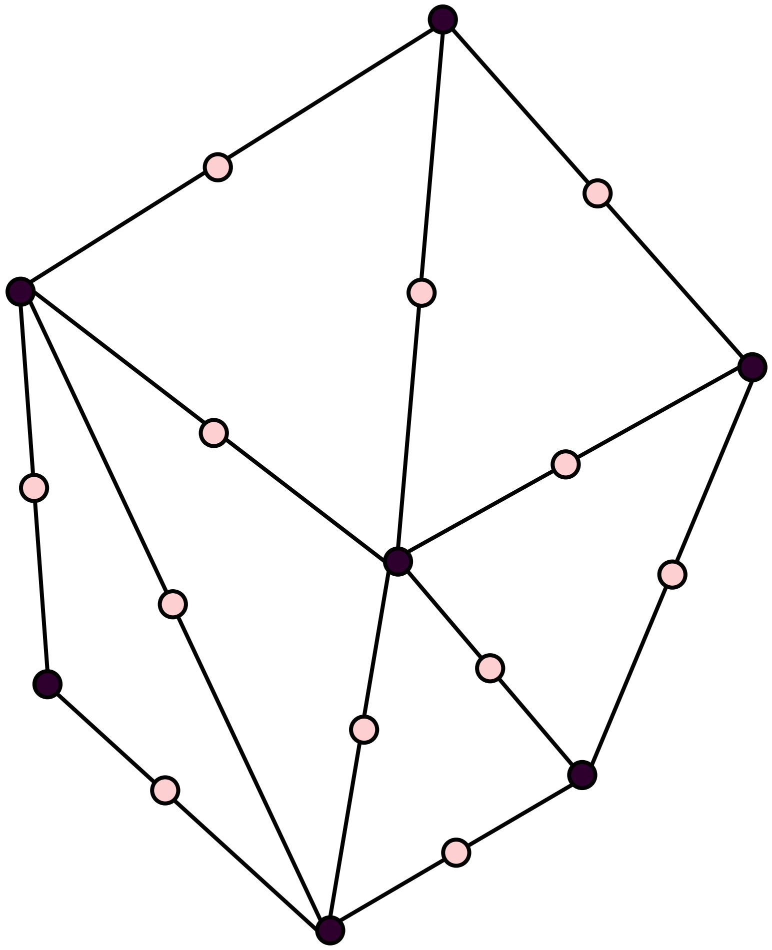
(xii) Delete vertex V_2

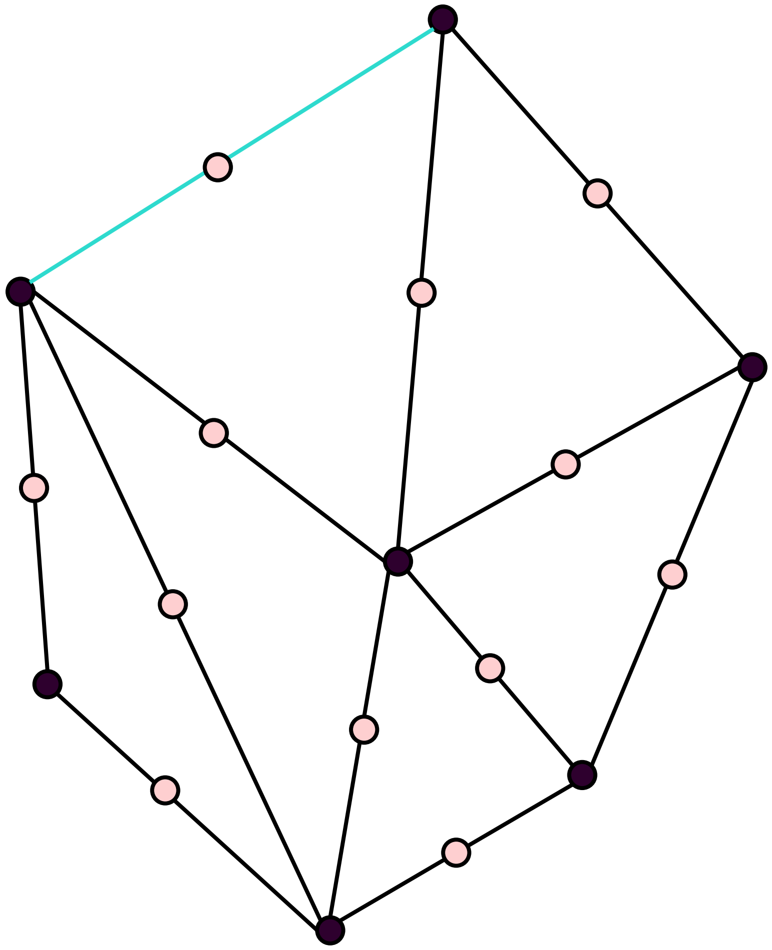
(xiii) Delete half-edges $e_1, e_2, e_4, e_5, e_7, e_8$

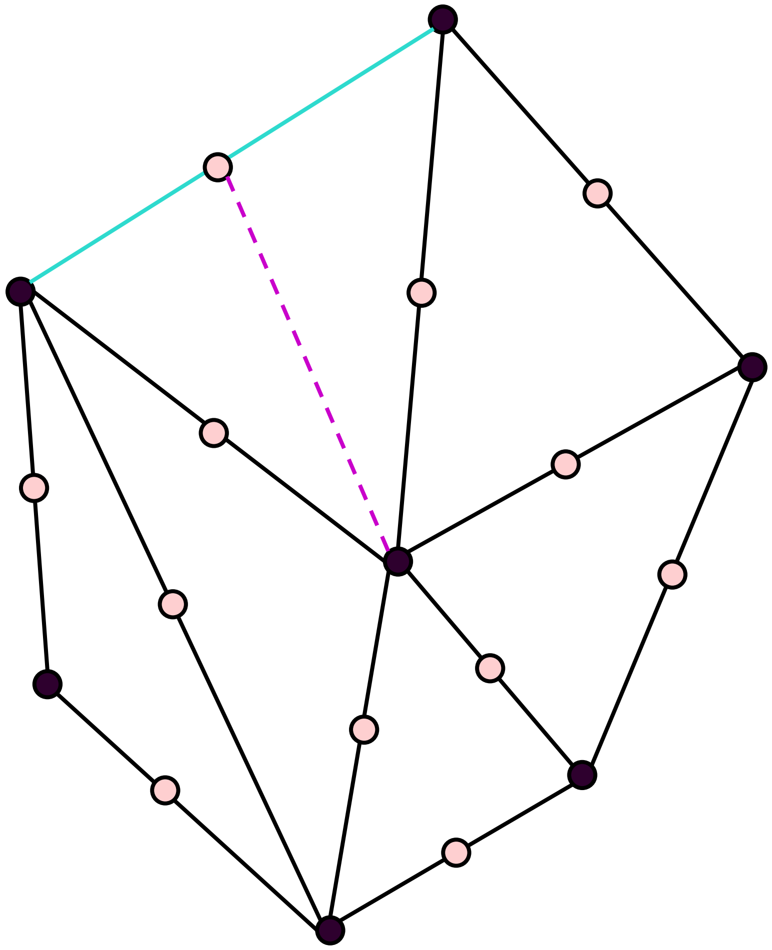
(xiv) Delete faces f_0, f_1

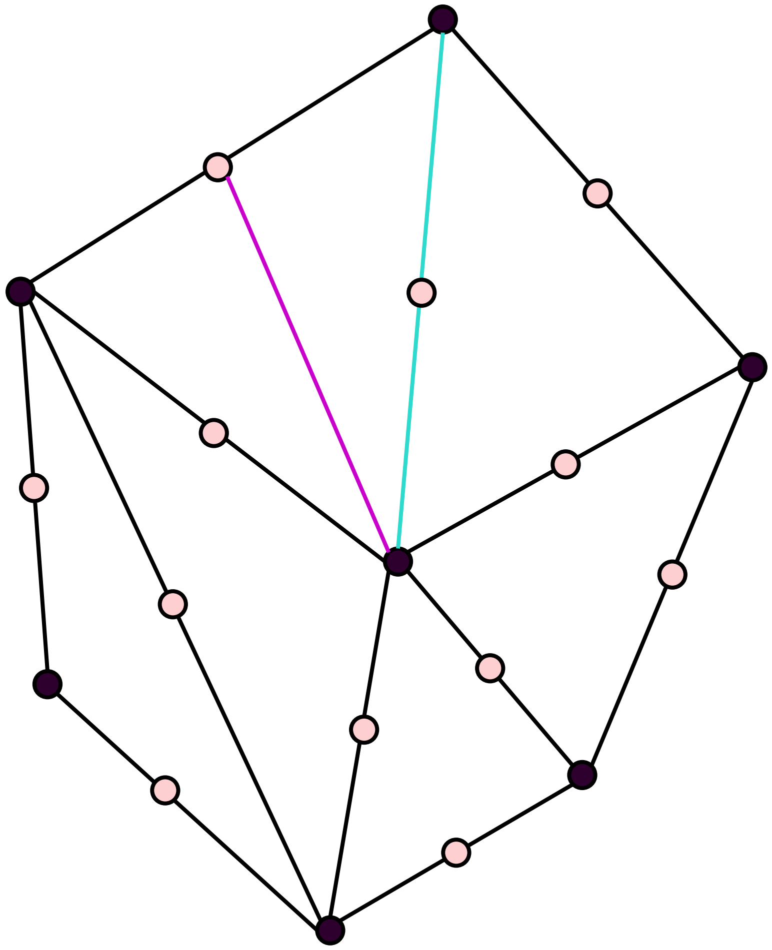
Worksheet Question 2

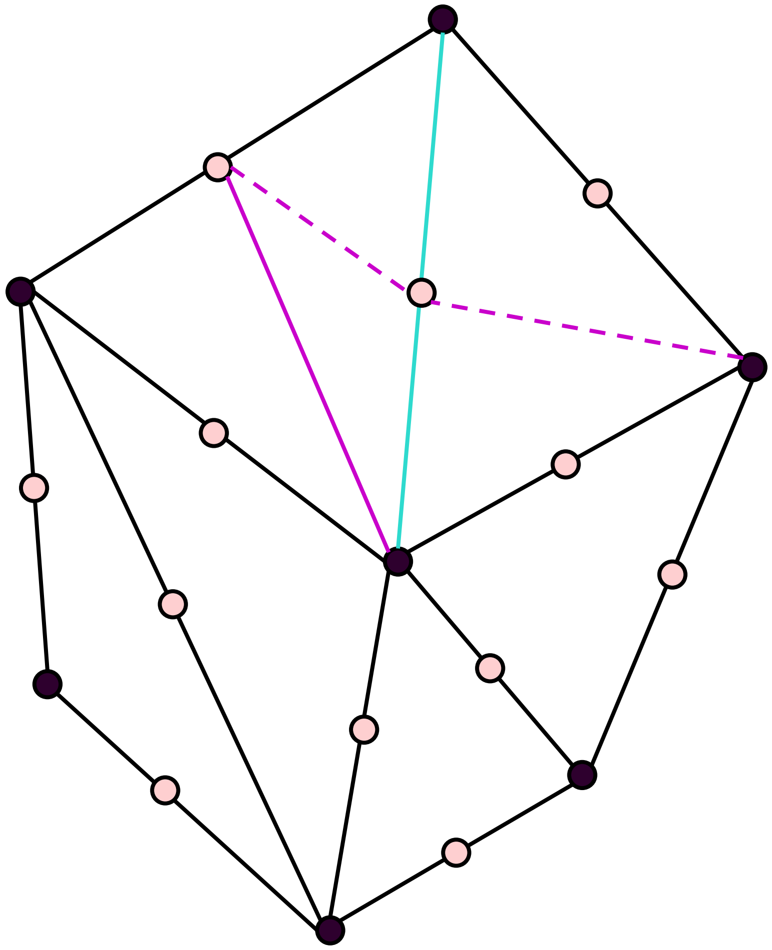

B
A
C
D
G
F
E

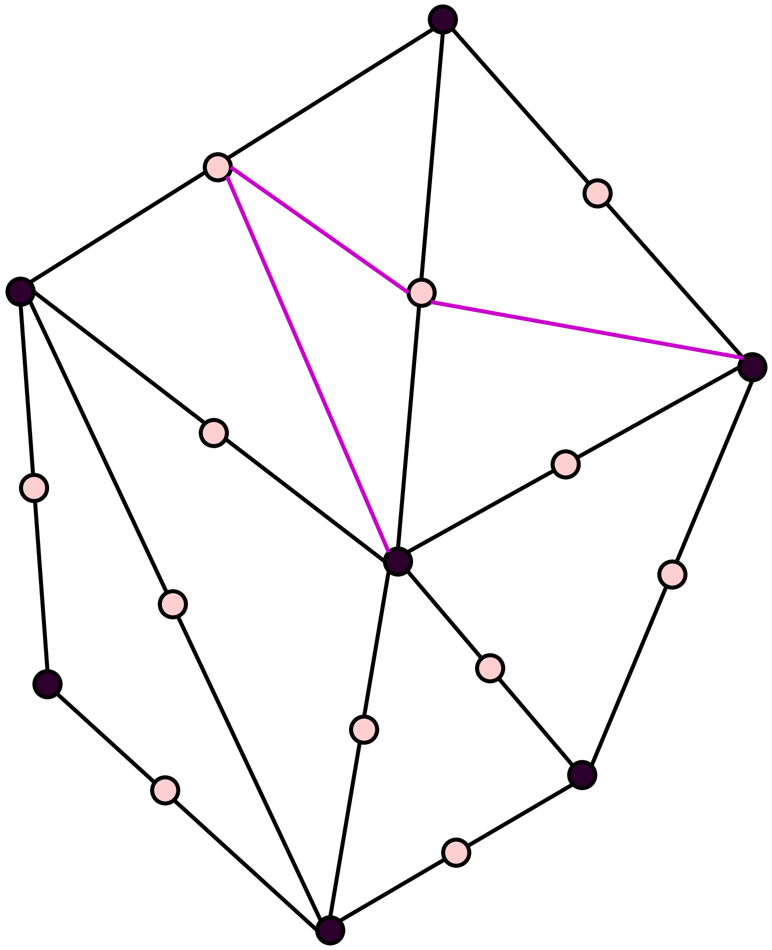






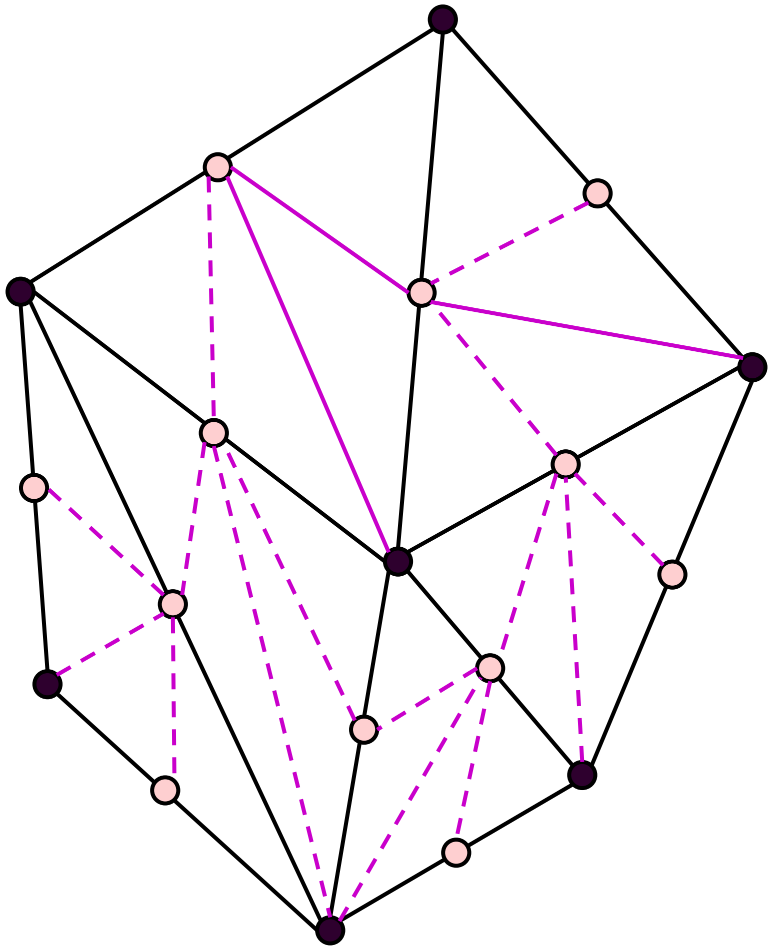






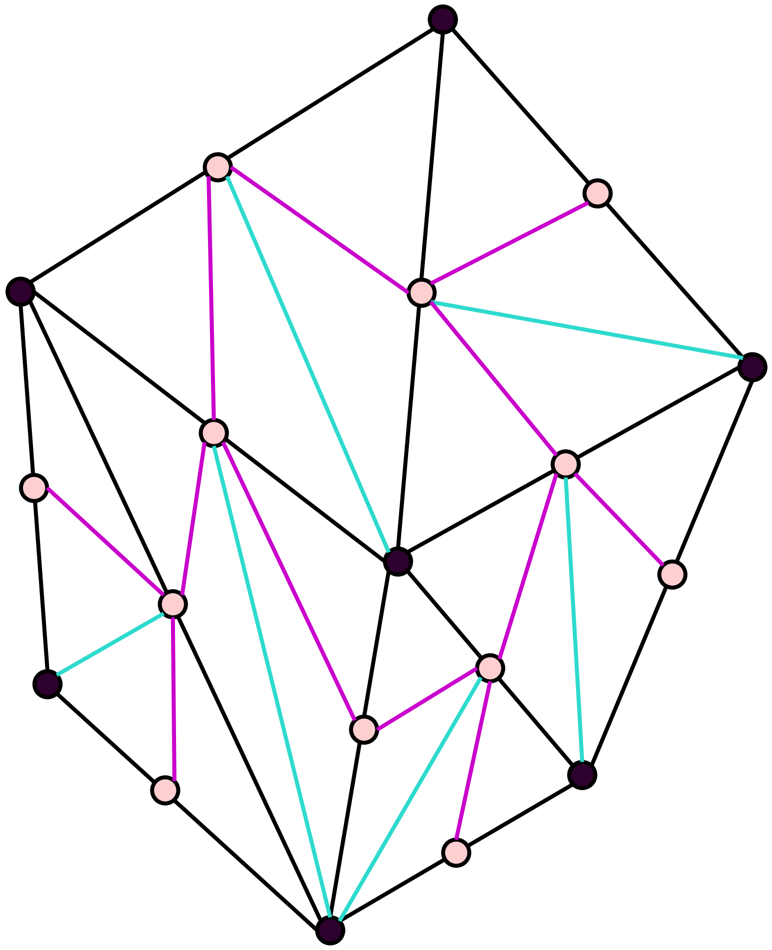
This result depends on the order that edges are processed in!

Do you see why?



This result depends on the order that edges are processed in!

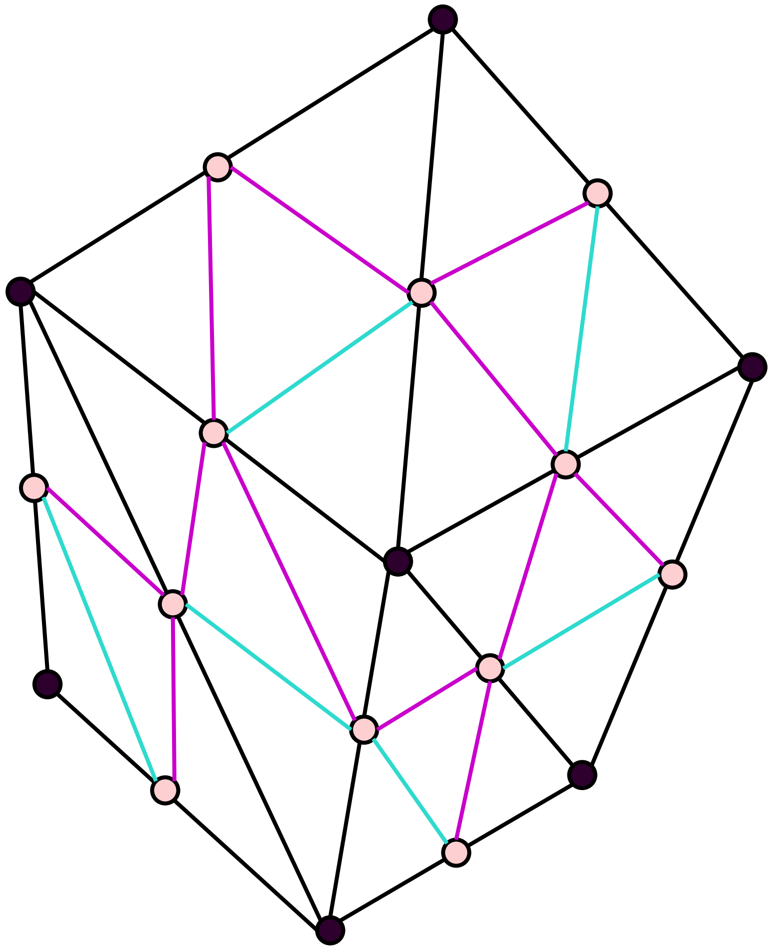
Do you see why?



For any new edge that connects *new* vertex to *old* vertex...

This result depends on the order that edges are processed in!

Do you see why?

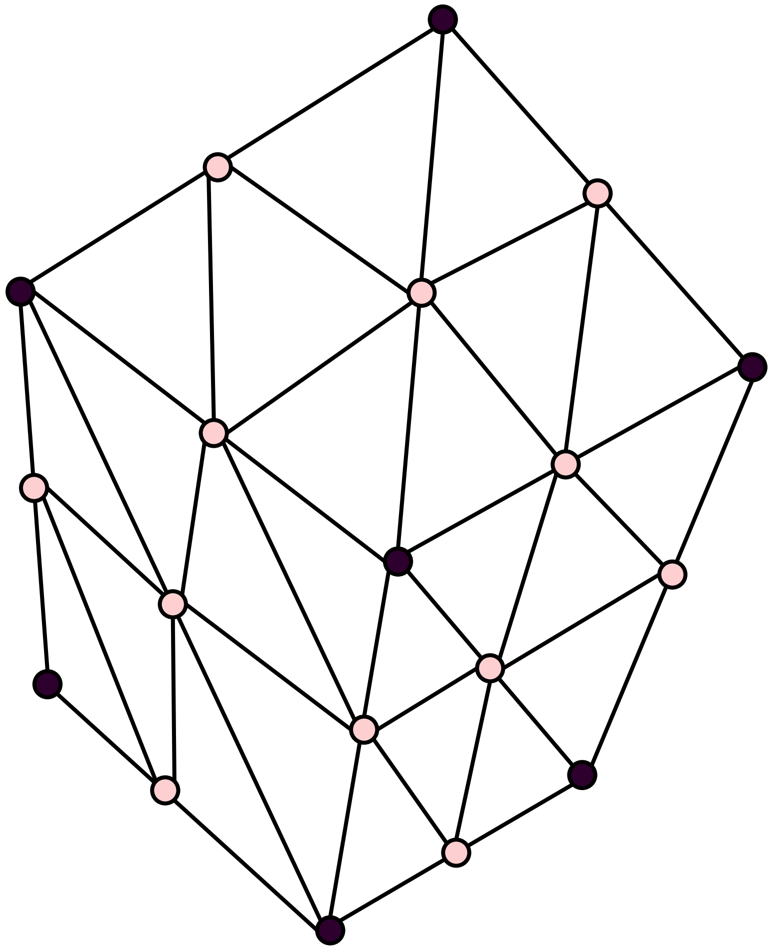


For any new edge that connects *new* vertex to *old* vertex...

Edge flip!

This result depends on the order that edges are processed in!

Do you see why?



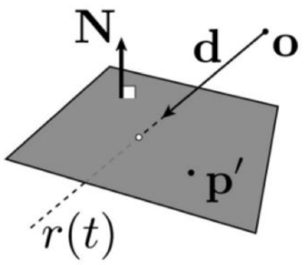
For any new edge that connects *new* vertex to *old* vertex...

Edge flip!

This completes the *face splitting* procedure.

Worksheet Question 3

1. As a warm-up, let's re-derive the equation for a ray intersecting an arbitrary plane. Recall that a plane can be defined as the set of all points $\mathbf{p}$ satisfying



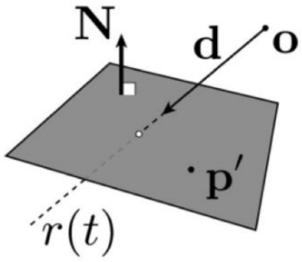
$$(\mathbf{p} - \mathbf{p}') \cdot \mathbf{N} = 0,$$

where $\mathbf{p}'$ is any point on the plane and $\mathbf{N}$ is the plane's normal vector. Set $\mathbf{p}$ equal to $\mathbf{r}(t)$ and solve for t.

1. As a warm-up, let's re-derive the equation for a ray intersecting an arbitrary plane. Recall that a plane can be defined as the set of all points $\mathbf{p}$ satisfying

$$(\mathbf{p} - \mathbf{p}') \cdot \mathbf{N} = 0,$$

where $\mathbf{p}'$ is any point on the plane and $\mathbf{N}$ is the plane's normal vector. Set $\mathbf{p}$ equal to $\mathbf{r}(t)$ and solve for t.



Solution:

$(\mathbf{p} - \mathbf{p}') \cdot \mathbf{N} = 0$

$(\mathbf{o} + t\mathbf{d} - \mathbf{p}') \cdot \mathbf{N} = 0$

$(\mathbf{o} - \mathbf{p}') \cdot \mathbf{N} + t\mathbf{d} \cdot \mathbf{N} = 0$

$t\mathbf{d} \cdot \mathbf{N} = (\mathbf{p}' - \mathbf{o}) \cdot \mathbf{N}$

$t = \frac{(\mathbf{p}' - \mathbf{o}) \cdot \mathbf{N}}{\mathbf{d} \cdot \mathbf{N}}$

2. What does it mean if we get a value of $t < 0$?

3. What does it mean if $\mathbf{d} \cdot \mathbf{N} = 0$?

2. What does it mean if we get a value of $t < 0$?

Solution: This means that the intersection point with the plane is behind the ray's origin. Since the ray only moves forward in the direction of $\mathbf{d}$ for positive values of t, $t < 0$ indicates that this value of t is not a valid intersection with the ray.

Note that this is true for any ray-surface intersection problem, not just with planes.

3. What does it mean if $\mathbf{d} \cdot \mathbf{N} = 0$?

2. What does it mean if we get a value of $t < 0$?

Solution: This means that the intersection point with the plane is behind the ray's origin. Since the ray only moves forward in the direction of $\mathbf{d}$ for positive values of t, $t < 0$ indicates that this value of t is not a valid intersection with the ray.

Note that this is true for any ray-surface intersection problem, not just with planes.

3. What does it mean if $\mathbf{d} \cdot \mathbf{N} = 0$?

Solution: When $\mathbf{d} \cdot \mathbf{N} = 0$, the ray's direction vector is perpendicular to the plane's normal vector $\mathbf{N}$. Therefore, the ray is parallel to the plane and will either intersect for all values of t or not intersect at all.

To see which is the case, substitute $\mathbf{r}(t) = \mathbf{o} + t\mathbf{d}$ into the plane equation $(\mathbf{p} - \mathbf{p}') \cdot \mathbf{N} = 0$:

$$(\mathbf{o} + t\mathbf{d} - \mathbf{p}') \cdot \mathbf{N} = 0 \quad \Longrightarrow \quad (\mathbf{o} - \mathbf{p}') \cdot \mathbf{N} + t(\mathbf{d} \cdot \mathbf{N}) = 0.$$

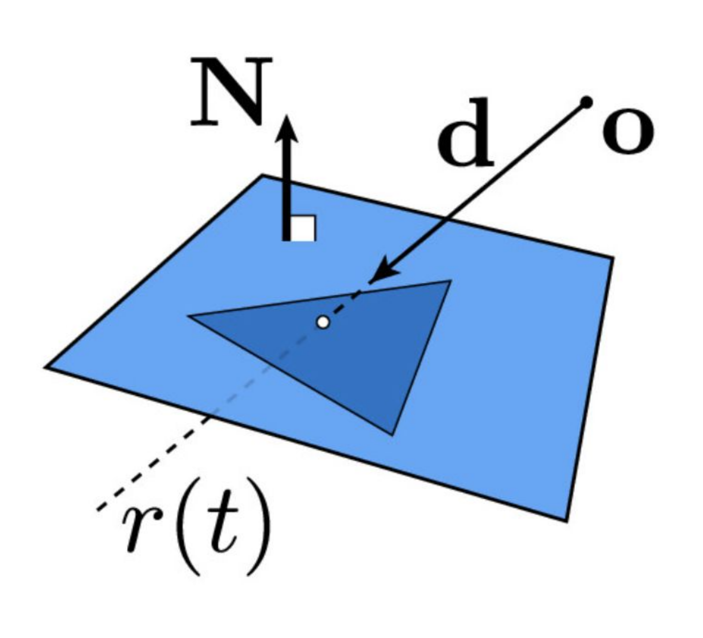
But since $\mathbf{d} \cdot \mathbf{N} = 0$, we get

$$(\mathbf{o} - \mathbf{p}') \cdot \mathbf{N} = 0.$$

If $(\mathbf{o} - \mathbf{p}') \cdot \mathbf{N} = 0$, the entire line (extended ray) lies in the plane, yielding infinite intersections.
If $(\mathbf{o} - \mathbf{p}') \cdot \mathbf{N} \neq 0$, the ray never intersects the plane (zero intersections).

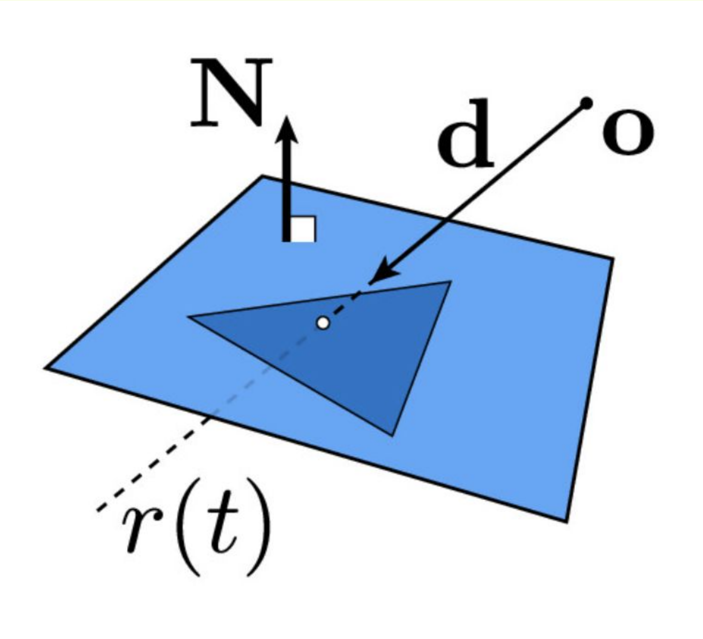
Bonus Question!

How would you check if an intersection is inside a triangle?



Bonus Question!

How would you check if an intersection is inside a triangle?



Triangle is in a plane

- Ray-plane intersection
- Test if hit point is inside triangle (Assignment 1!)

Many ways to optimize…

2 Ray-Surface Intersection

4. Given the following implicit representation of an ellipsoid and the definition of a ray, compute where (and at what parameter value(s) of t) the ray intersects the ellipsoid:

$$f(x, y, z) = \frac{(x-2)^2}{4} + (y-2)^2 + \frac{z^2}{4} - 1$$

$$\mathbf{r}(t) = (0,\ 0,\ 0) + t\,(1,\ 1,\ 0)$$

Start by substituting the ray $\mathbf{r}(t)$ into the function $f(x, y, z)$ to obtain $f(\mathbf{o} + t\,\mathbf{d})$.

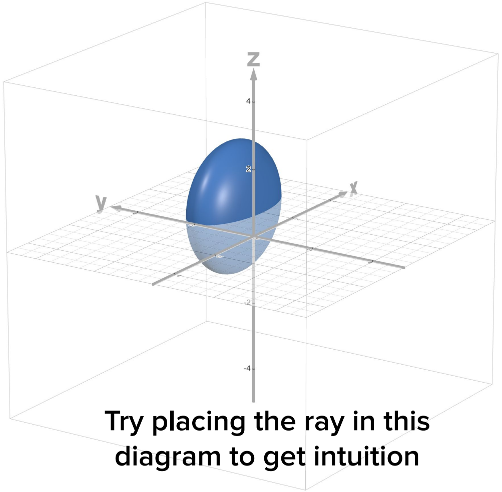
2 Ray-Surface Intersection

4. Given the following implicit representation of an ellipsoid and the definition of a ray, compute where (and at what parameter value(s) of t) the ray intersects the ellipsoid:

$$f(x, y, z) = \frac{(x-2)^2}{4} + (y-2)^2 + \frac{z^2}{4} - 1$$

$$\mathbf{r}(t) = (0,\ 0,\ 0) + t\,(1,\ 1,\ 0)$$

Start by substituting the ray $\mathbf{r}(t)$ into the function $f(x, y, z)$ to obtain $f(\mathbf{o} + t\,\mathbf{d})$.



Try placing the ray in this diagram to get intuition

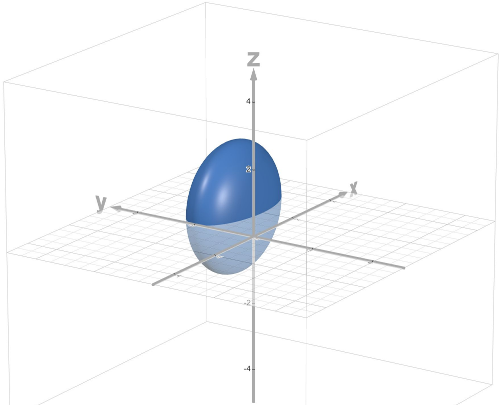
2 Ray-Surface Intersection

4. Given the following implicit representation of an ellipsoid and the definition of a ray, compute where (and at what parameter value(s) of t) the ray intersects the ellipsoid:

$$f(x, y, z) = \frac{(x-2)^2}{4} + (y-2)^2 + \frac{z^2}{4} - 1$$

$$\mathbf{r}(t) = (0,\ 0,\ 0) + t\,(1,\ 1,\ 0)$$

Start by substituting the ray $\mathbf{r}(t)$ into the function $f(x, y, z)$ to obtain $f(\mathbf{o} + t\,\mathbf{d})$.



Try placing the ray in this diagram to get intuition

Solution: Set $f(\mathbf{o} + t\mathbf{d}) = 0$ and solve for t. Then plug your value(s) of t back into the ray equation to find the corresponding point(s) of intersection. Finally, identify where the ray first hits the ellipsoid (i.e., the smallest positive t).

2 Ray-Surface Intersection

$$f(x, y, z) = \frac{(x-2)^2}{4} + (y-2)^2 + \frac{z^2}{4} - 1$$

$$\mathbf{r}(t) = (0,\ 0,\ 0) + t\,(1,\ 1,\ 0)$$

Solution: Set $f(\mathbf{o} + t\mathbf{d}) = 0$ and solve for t. Then plug your value(s) of t back into the ray equation to find the corresponding point(s) of intersection. Finally, identify where the ray first hits the ellipsoid (i.e., the smallest positive t).

2 Ray-Surface Intersection

$$f(x, y, z) = \frac{(x-2)^2}{4} + (y-2)^2 + \frac{z^2}{4} - 1$$

$$\mathbf{r}(t) = (0,\,0,\,0) + t\,(1,\,1,\,0)$$

Solution: Set $f(\mathbf{o} + t\mathbf{d}) = 0$ and solve for t. Then plug your value(s) of t back into the ray equation to find the corresponding point(s) of intersection. Finally, identify where the ray first hits the ellipsoid (i.e., the smallest positive t).

To get $f(\mathbf{o} + t\mathbf{d}) = 0$, we substitute $x = 0 + t \cdot 1 = t$, $y = 0 + t \cdot 1 = t$, and $z = 0 + t \cdot 0 = 0$. This gives us:

$$\frac{(x-2)^2}{4} + (y-2)^2 + \frac{z^2}{4} - 1 = 0$$

$$\frac{(t-2)^2}{4} + (t-2)^2 - 1 = 0$$

$$\frac{5}{4}(t-2)^2 = 1$$

$$(t-2)^2 = \frac{4}{5} \implies t = 2 \pm \sqrt{\frac{4}{5}}$$

2 Ray-Surface Intersection

$$f(x, y, z) = \frac{(x-2)^2}{4} + (y-2)^2 + \frac{z^2}{4} - 1$$

$$\mathbf{r}(t) = (0,\, 0,\, 0) + t\,(1,\, 1,\, 0)$$

Solution: Set $f(\mathbf{o} + t\mathbf{d}) = 0$ and solve for t. Then plug your value(s) of t back into the ray equation to find the corresponding point(s) of intersection. Finally, identify where the ray first hits the ellipsoid (i.e., the smallest positive t).

Both t values are positive (since $\sqrt{\frac{4}{5}} < 2$), meaning they are both valid intersections. The first t value is the first intersection. Plugging in $t = 2 - \sqrt{\frac{4}{5}}$, we see that the ray first intersects the ellipsoid at $(0,0,0) + (2 - \sqrt{\frac{4}{5}})\,(1,1,0) = \boxed{(2 - \sqrt{\frac{4}{5}},\, 2 - \sqrt{\frac{4}{5}},\, 0)}$.

$$t = 2 \pm \sqrt{\frac{4}{5}}$$

Let's Take Attendance.

- Be sure to select Week 5 and input your TA's pre-exam secret word 😉
- Any feedback? Let us know!