Fortune's Algorithm for Computing the Voronoi Diagram

Outline

- Math Review
- Overview of the Algorithm
- Implementation

Math Review

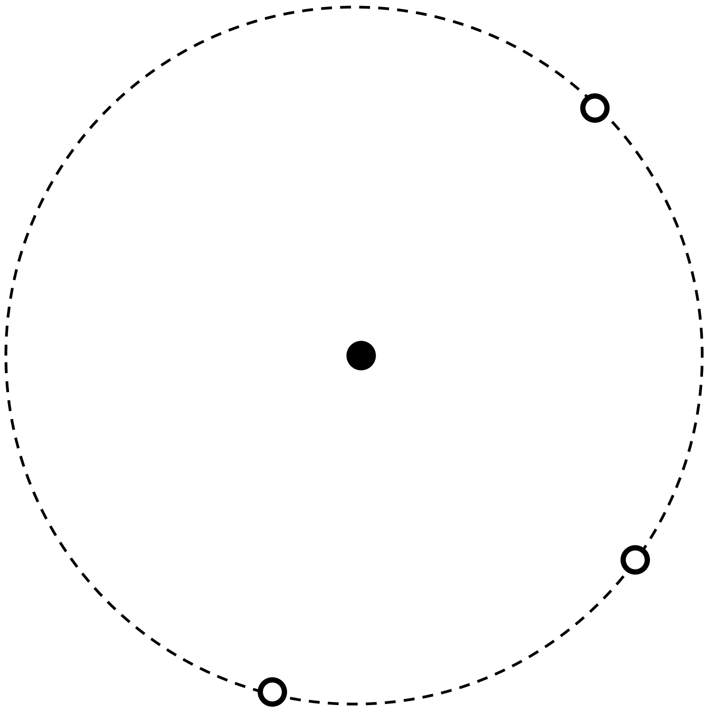
Circumcircles:

Q: Given three points in 2D, how do we compute the center (and radius) of the circumcircle?

Math Review

Circumcircles:

Q: Given three points in 2D, how do we compute the center (and radius) of the circumcircle?

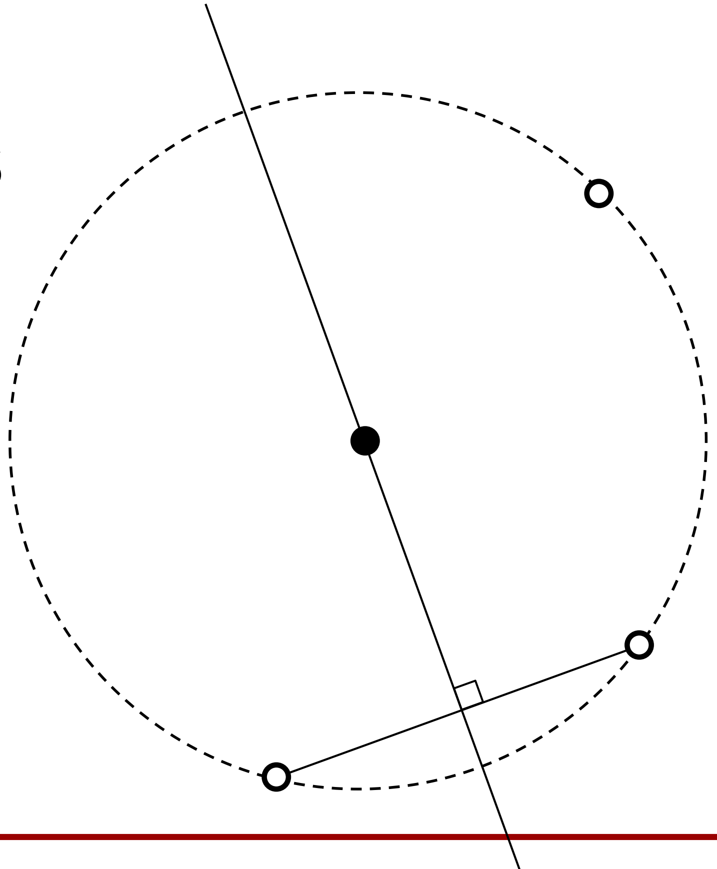


Math Review

Circumcircles:

A: Pick two of the points and draw the perpendicular bisector.

The bisector must pass through the center of the circumcircle.

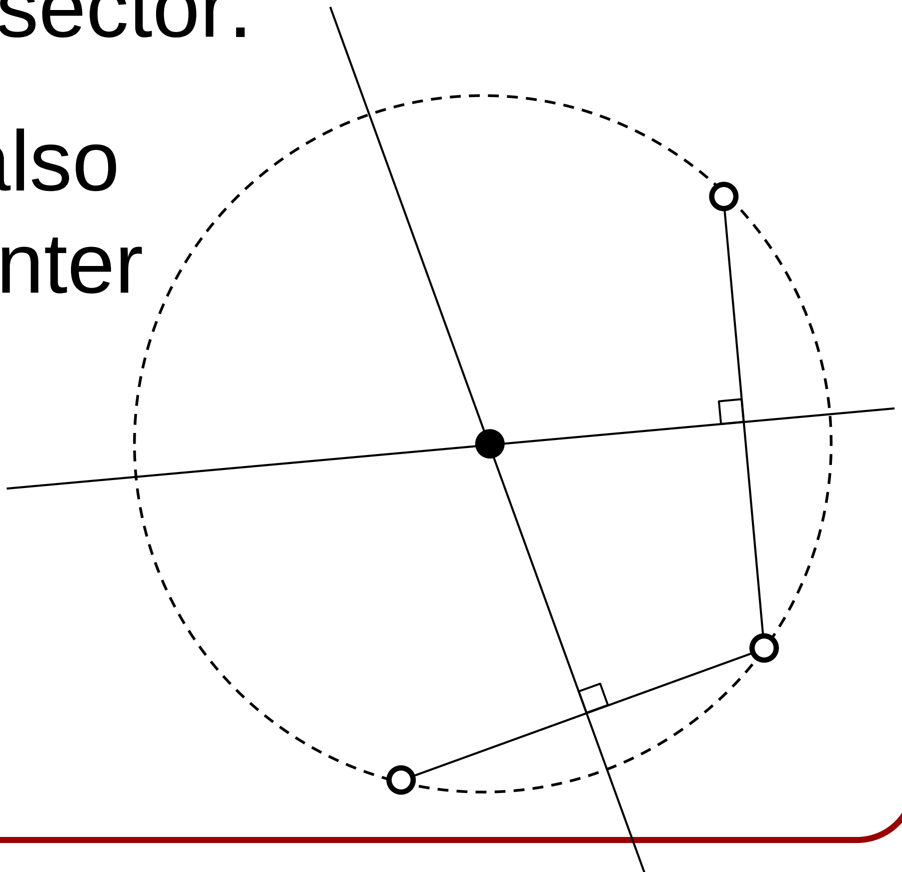


Math Review

<u>Circumcircles</u>:

A: Pick another two of the points and draw the perpendicular bisector.

This bisector must also pass through the center of the circumcircle.



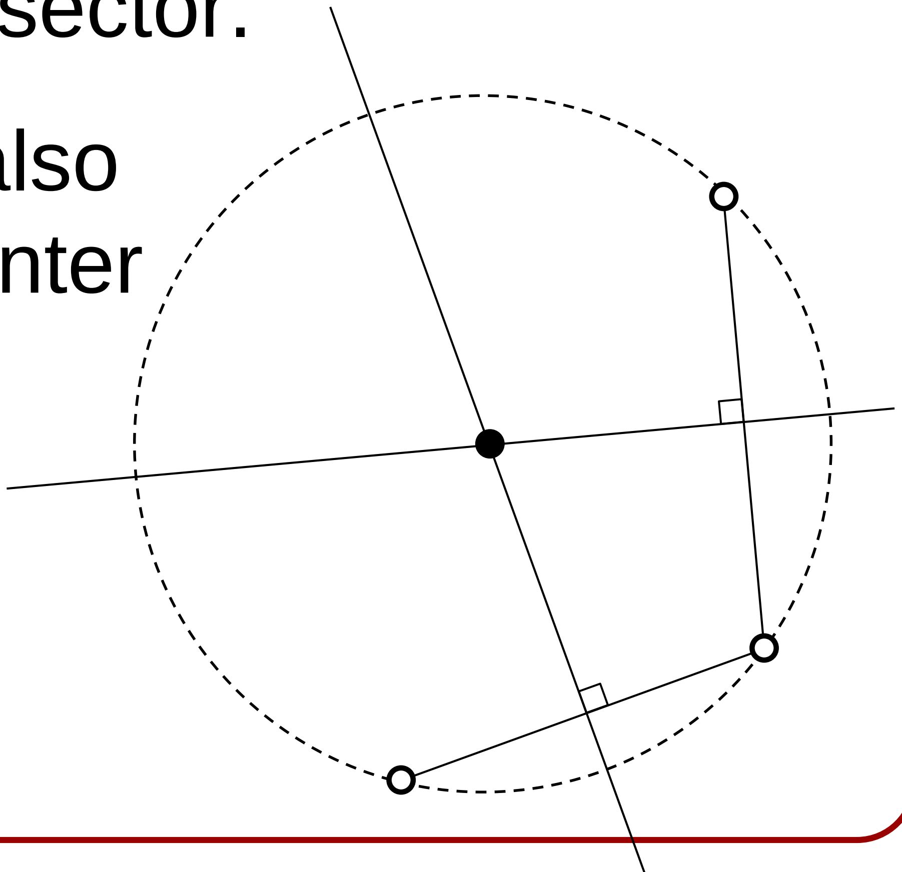
Math Review

Circumcircles:

A: Pick another two of the points and draw the perpendicular bisector.

This bisector must also pass through the center of the circumcircle.

⇒ The intersection of the bisectors is the center.



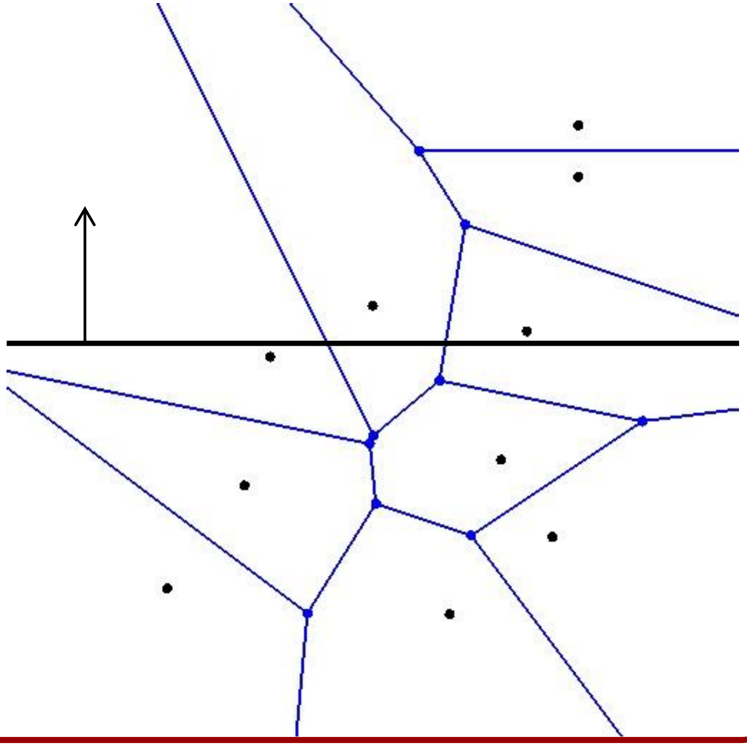
Outline

- Math Review
- Overview of the Algorithm
- Implementation

Overview of the Algorithm

<u>Goal</u>:

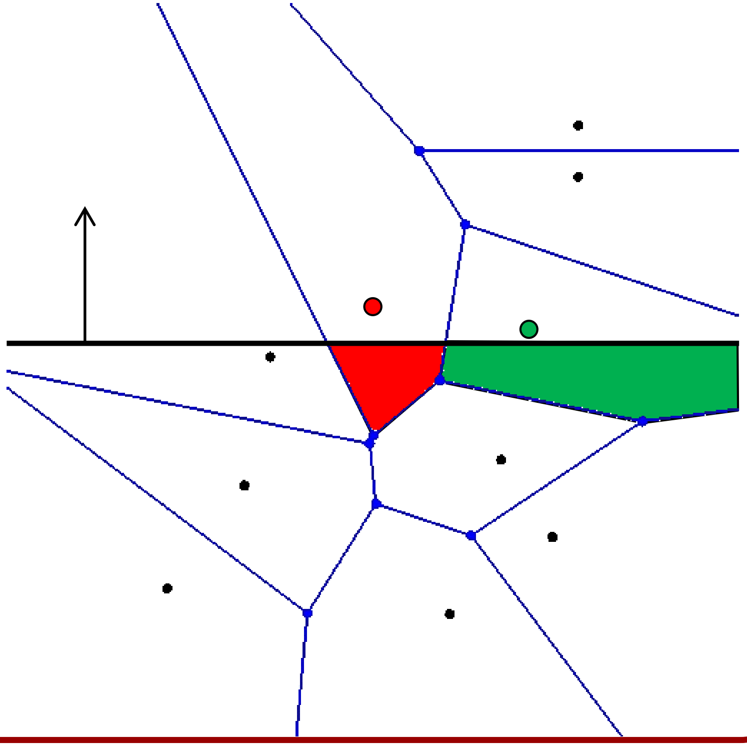
Given a set of points (sites), the goal is to use a sweep-line algorithm to construct the Voronoi diagram.



Overview of the Algorithm

<u>Challenge</u>:

Regions that have already been swept may be closer to sites that are in front of the sweep line, so we cannot know how the Voronoi Diagram looks there.



Overview of the Algorithm

Suppose that:

- The sweep-line is at position $y = y_l$
- We've seen a site at position $s = (x_s, y_s)$, with $y_s < y_l$

The set of points $p = (x, y)$ closer to the site than to the sweep-line satisfies:

$$\|p - s\|^2 \leq (y - y_l)^2$$

$$\Updownarrow$$

$$(x - x_s)^2 + (y - y_s)^2 \leq (y - y_l)^2$$

$$\Updownarrow$$

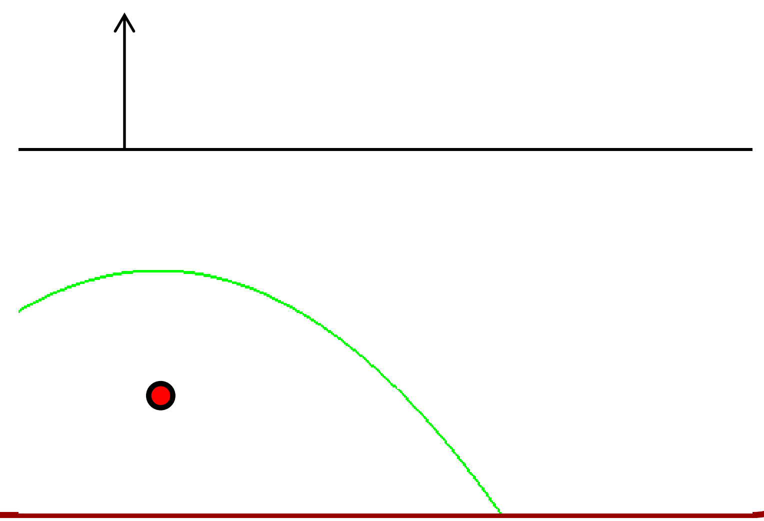
$$y \leq \frac{x^2 - 2x \cdot x_s + x_s^2 + y_s^2 - y_l^2}{2y_s - 2y_l}$$

Overview of the Algorithm

Given a sweep-line $y = y_l$ and site $s = (x_s, y_s)$, the set of points $p = (x, y)$ closer to the site than to the sweep-line $y = y_l$ satisfies:

$$\boxed{y \leq \frac{x^2 - 2x \cdot x_s + x_s^2 + y_s^2 - y_l^2}{2y_s - 2y_l}}$$

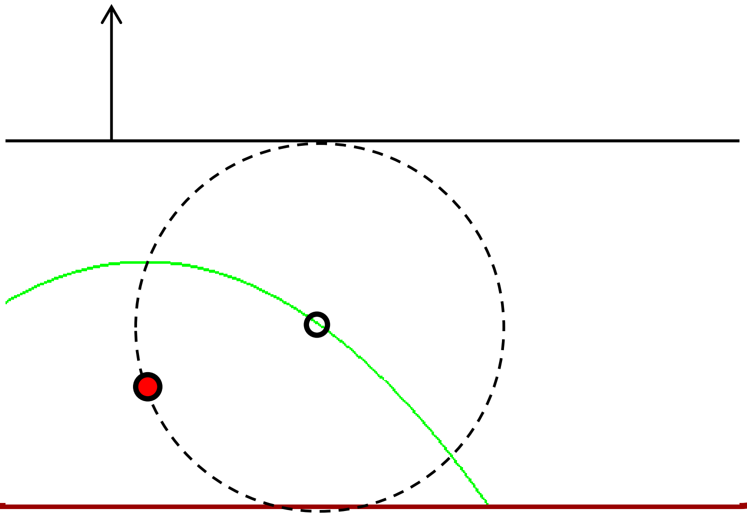
This is the equation of a parabola that expands as we advance the sweep-line.



Overview of the Algorithm

$$y \leq \frac{x^2 - 2x \cdot x_s + x_s^2 + y_s^2 - y_l^2}{2y_s - 2y_l}$$

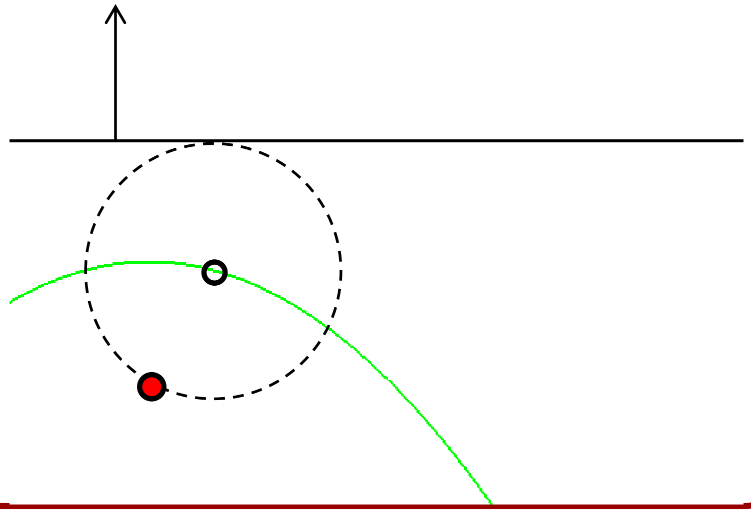
Points on the parabola are equidistant to the site and the sweep-line.



Overview of the Algorithm

$$y \leq \frac{x^2 - 2x \cdot x_s + x_s^2 + y_s^2 - y_l^2}{2y_s - 2y_l}$$

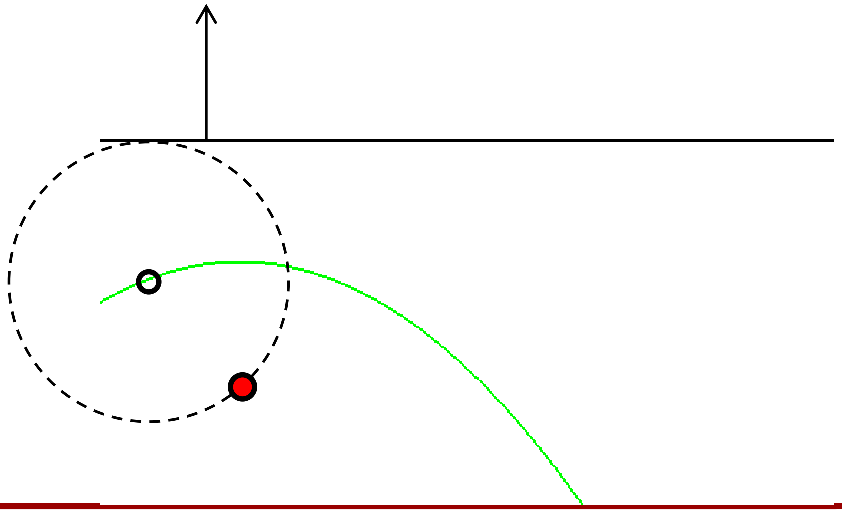
Points on the parabola are equidistant to the site and the sweep-line.



Overview of the Algorithm

$$y \leq \frac{x^2 - 2x \cdot x_s + x_s^2 + y_s^2 - y_l^2}{2y_s - 2y_l}$$

Points on the parabola are equidistant to the site and the sweep-line.

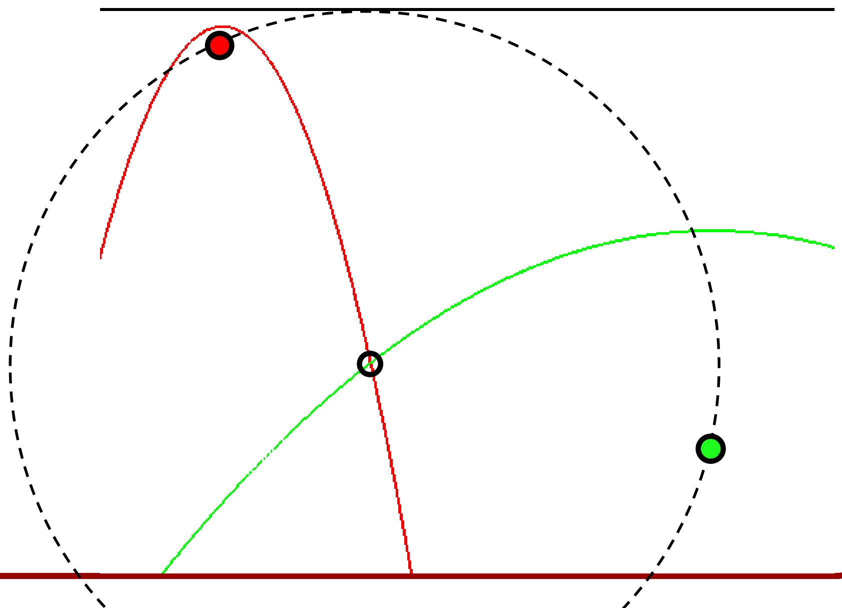


Overview of the Algorithm

$$y \leq \frac{x^2 - 2x \cdot x_s + x_s^2 + y_s^2 - y_l^2}{2y_s - 2y_l}$$

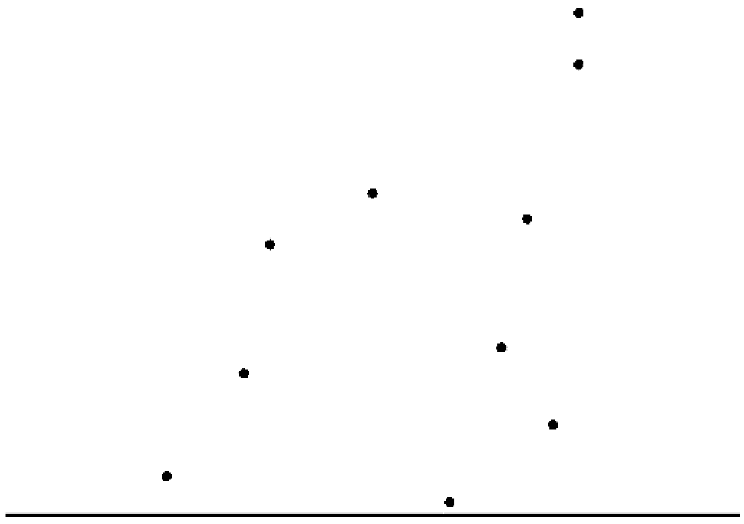
Points on the intersection of two such parabolas are equidistant to the two sites.

⇒ They could be on the Voronoi Diagram.



Overview of the Algorithm

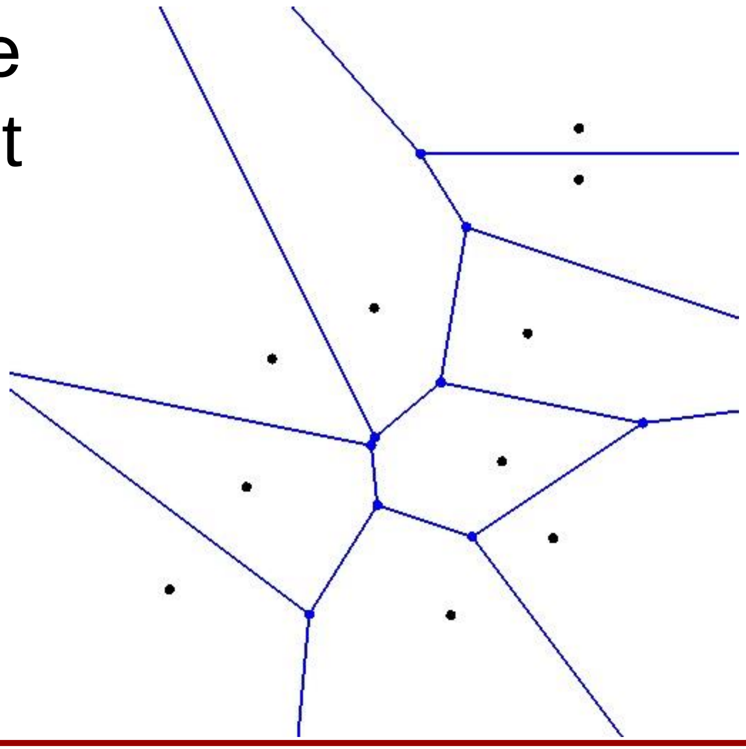
When advancing the sweep-line, we can associate a parabola with each seen site. We know we can finalize the Voronoi Diagram behind these parabolas (the *beach-line*).



Overview of the Algorithm

When advancing the sweep-line, we can associate a parabola with each seen site. We know we can finalize the Voronoi Diagram behind these parabolas (the *beach-line*).

Fortune's Algorithm tracks the beach-line as it evolves until it has passed through all of the event points.

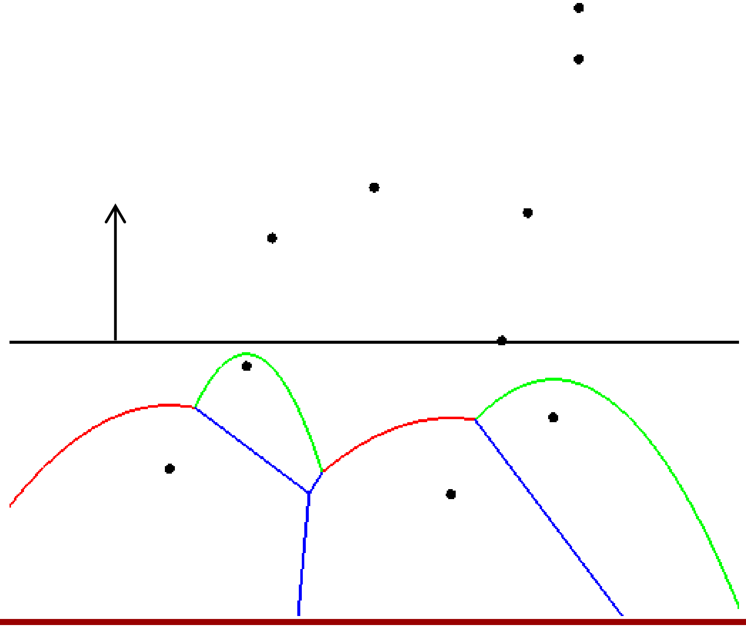


Overview of the Algorithm

As the sweep-line advances, the beach-line evolves in one of two ways:

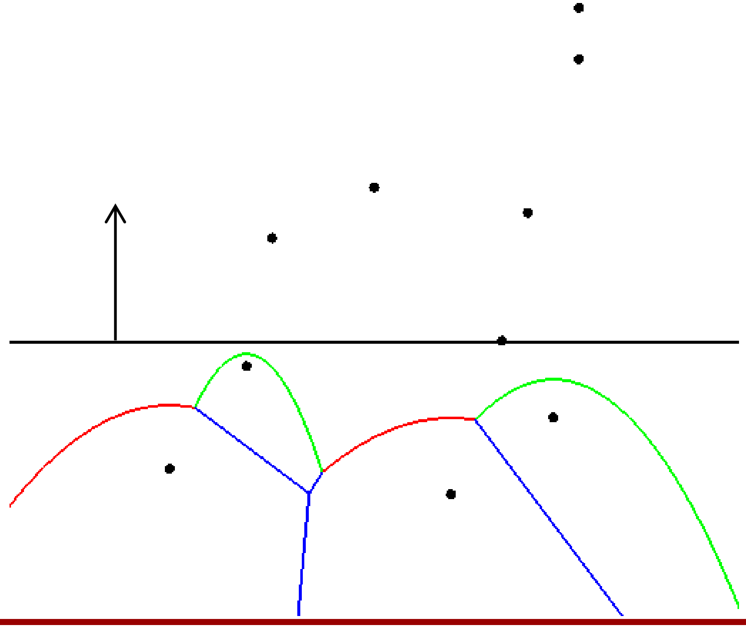
- Discrete:
  The topology of the beach-front changes
- Continuous:
  The geometry of an individual arc changes

We only need to track the discrete events.



Overview of the Algorithm

What are the events that change the topology of the beach-line?

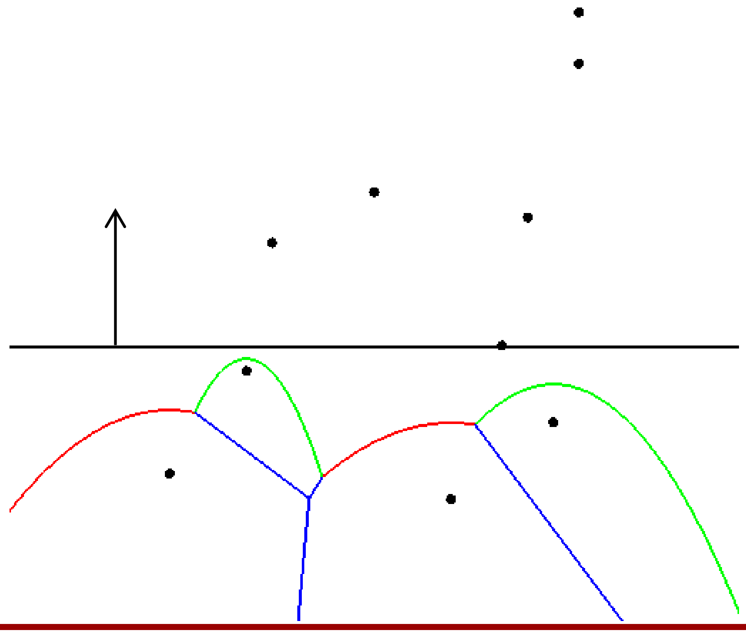


Overview of the Algorithm

What are the events that change the topology of the beach-line?

- The sweep-line passes across a site

  ⇒ A new parabola is introduced, splitting an old parabola in two

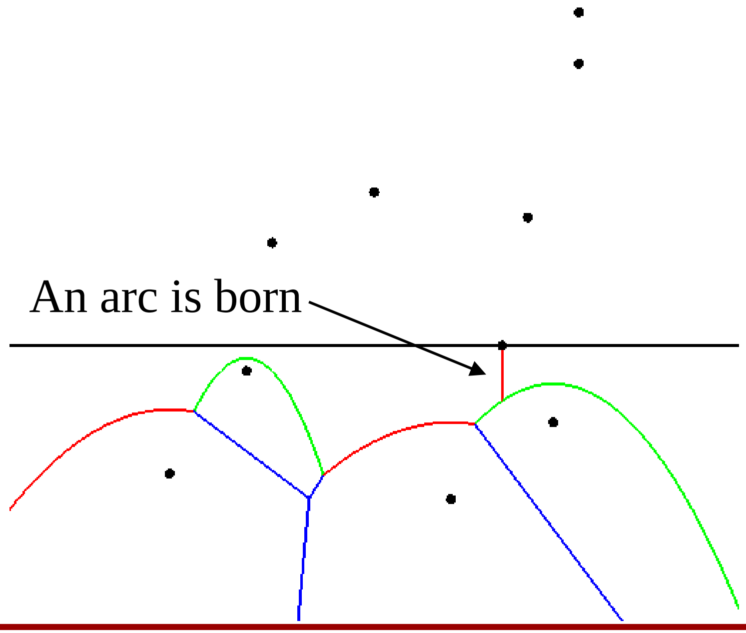


Overview of the Algorithm

What are the events that change the topology of the beach-line?

- The sweep-line passes across a site

  ⇒ A new parabola is introduced, splitting an old parabola in two

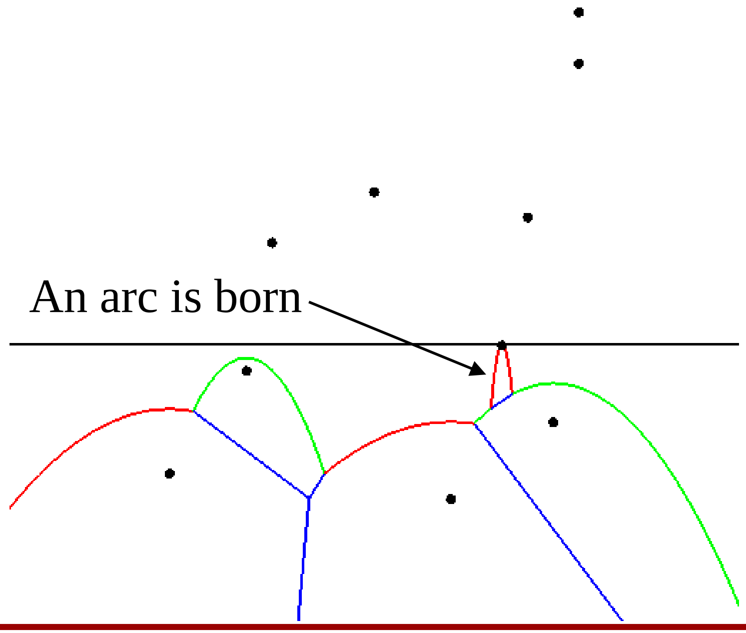


Overview of the Algorithm

What are the events that change the topology of the beach-line?

- The sweep-line passes across a site

  ⇒ A new parabola is introduced, splitting an old parabola in two

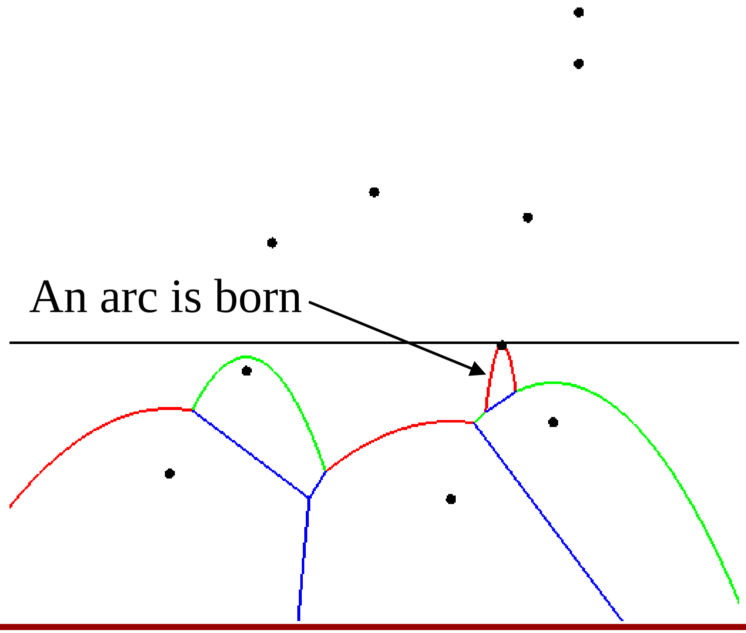


Overview of the Algorithm

What are the events that change the topology of the beach-line?

- The sweep-line passes across a site

  ⇒ A new parabola is introduced, splitting an old parabola in two

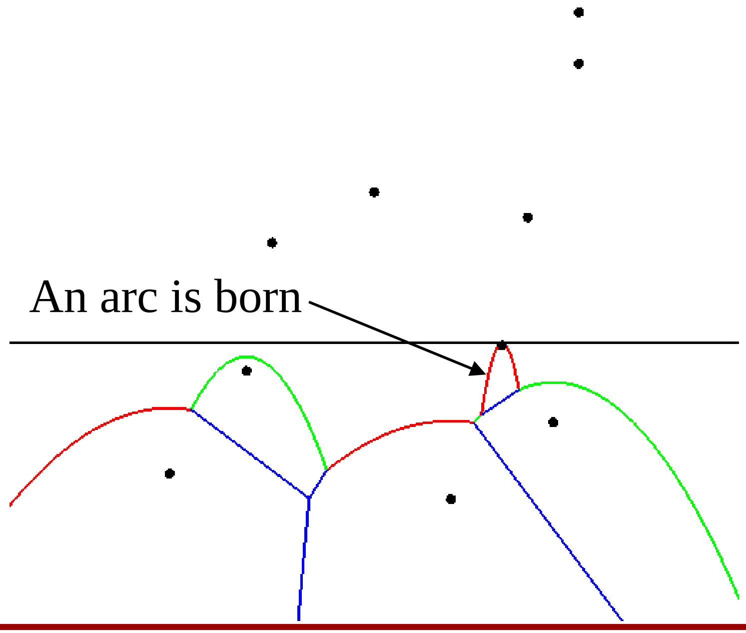


Overview of the Algorithm

What are the events that change the topology of the beach-line?

- The sweep-line passes across a site

  ⇒ A new parabola is introduced, splitting an old parabola in two



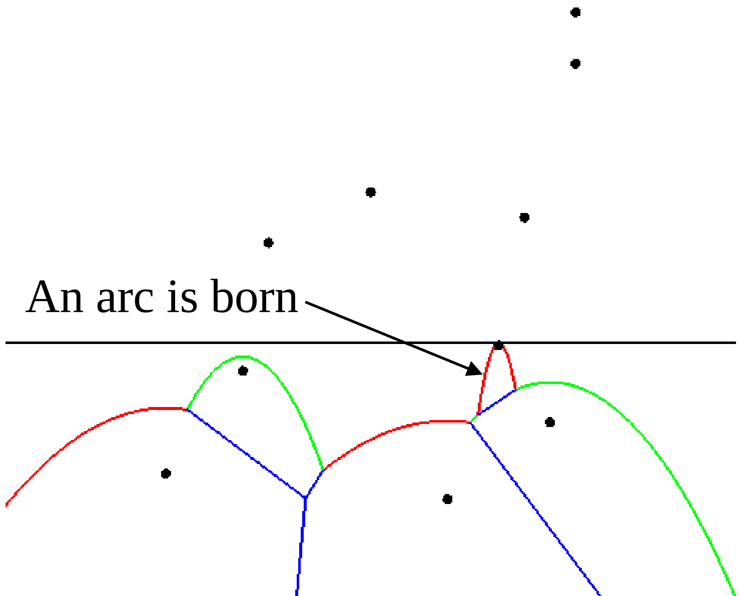
Overview of the Algorithm

What are the events that change the topology of the beach-line?

- The sweep-line passes across a site

  ⇒ A new parabola is introduced, splitting an old parabola in two

This corresponds to a new face in the Voronoi Diagram.



Overview of the Algorithm

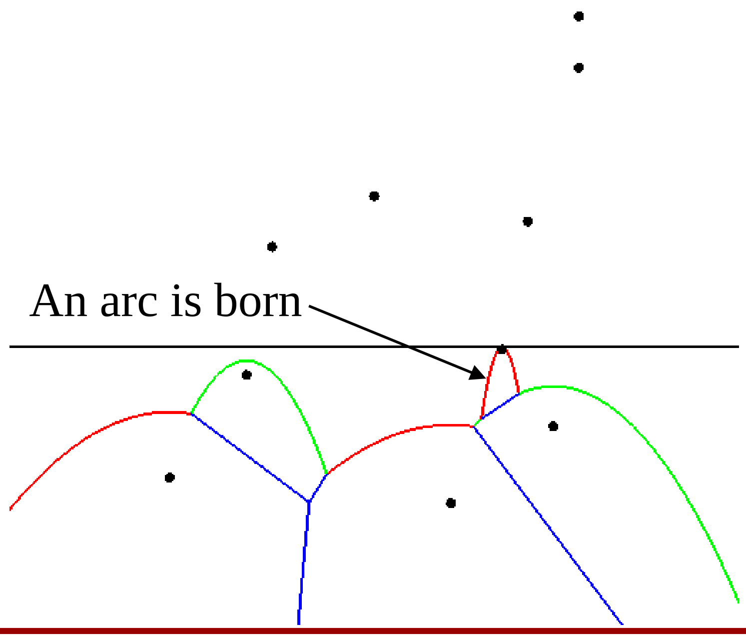
What are the events that change the topology of the beach-line?

- The sweep-line passes across a site

  ⇒ A new parabola is introduced, splitting an old parabola in two

This corresponds to a new face in the Voronoi Diagram.

This occurs when the sweep-line passes through the site.

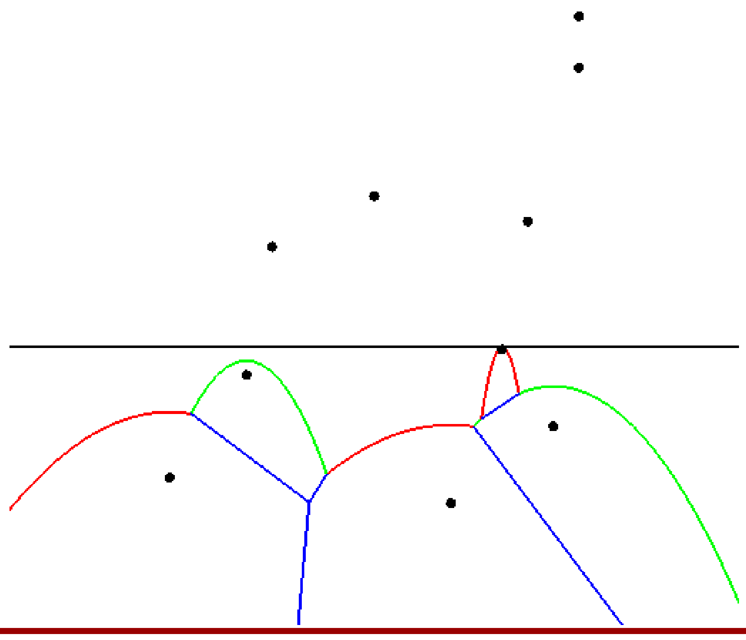


Overview of the Algorithm

What are the events that change the topology of the beach-line?

- One parabolic arc overtakes another

  ⇒ A parabolic arc is removed

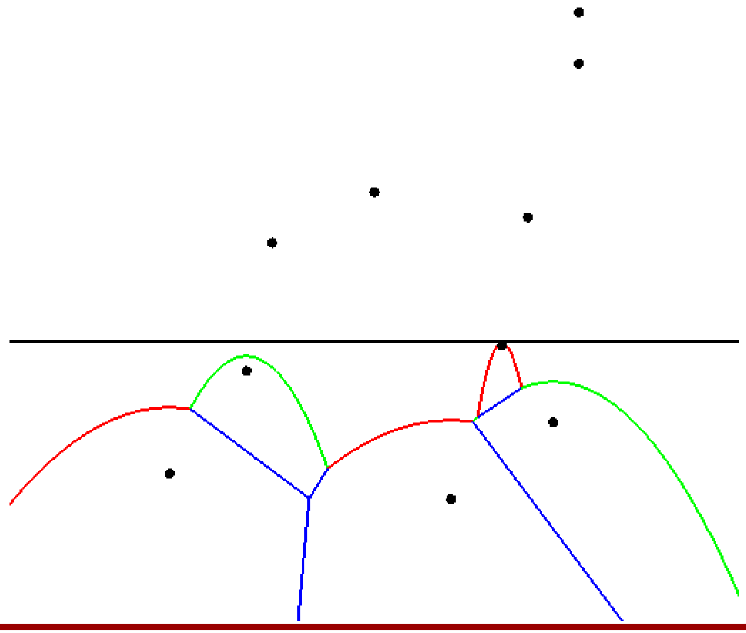


Overview of the Algorithm

What are the events that change the topology of the beach-line?

- One parabolic arc overtakes another

  ⇒ A parabolic arc is removed

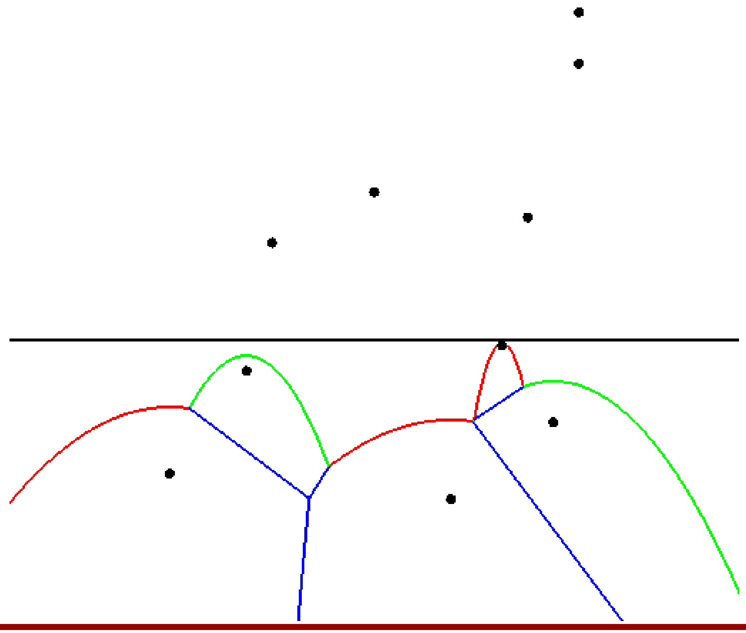


Overview of the Algorithm

What are the events that change the topology of the beach-line?

- One parabolic arc overtakes another

  ⇒ A parabolic arc is removed

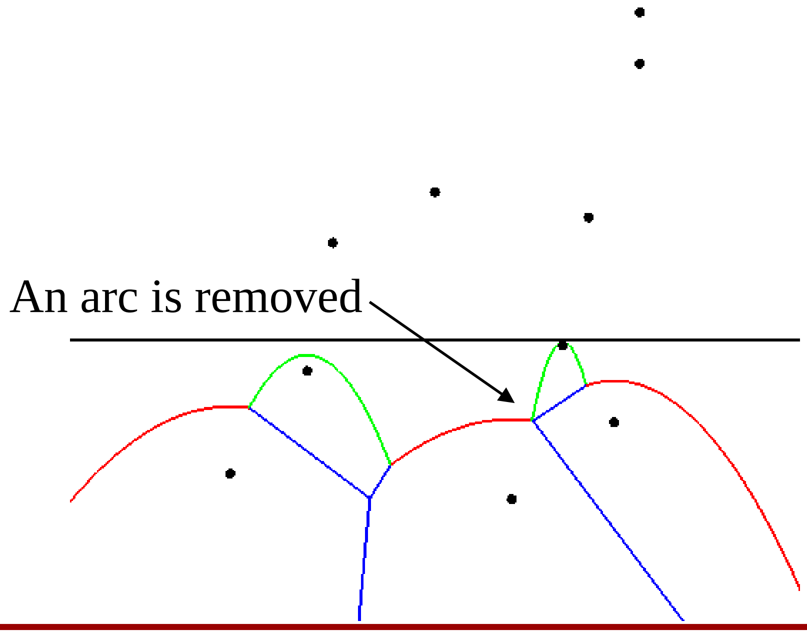


Overview of the Algorithm

What are the events that change the topology of the beach-line?

- One parabolic arc overtakes another

  ⇒ A parabolic arc is removed

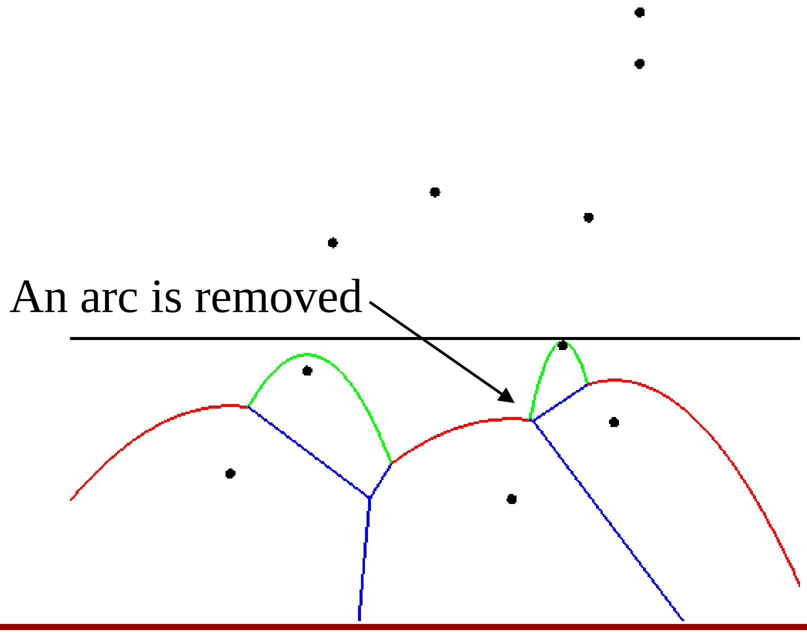


Overview of the Algorithm

What are the events that change the topology of the beach-line?

- One parabolic arc overtakes another

  ⇒ A parabolic arc is removed



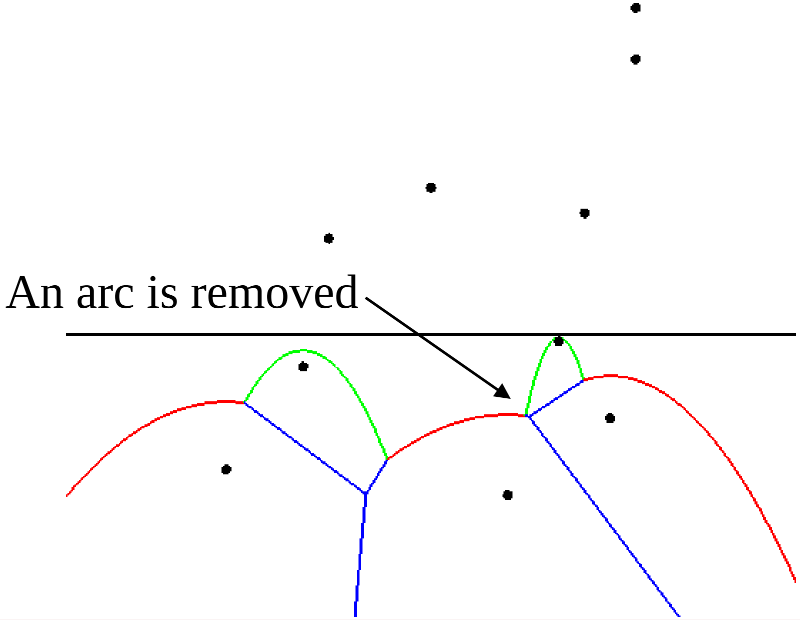
Overview of the Algorithm

What are the events that change the topology of the beach-line?

- One parabolic arc overtakes another

  ⇒ A parabolic arc is removed

This corresponds to a new vertex in the Voronoi Diagram.



Overview of the Algorithm

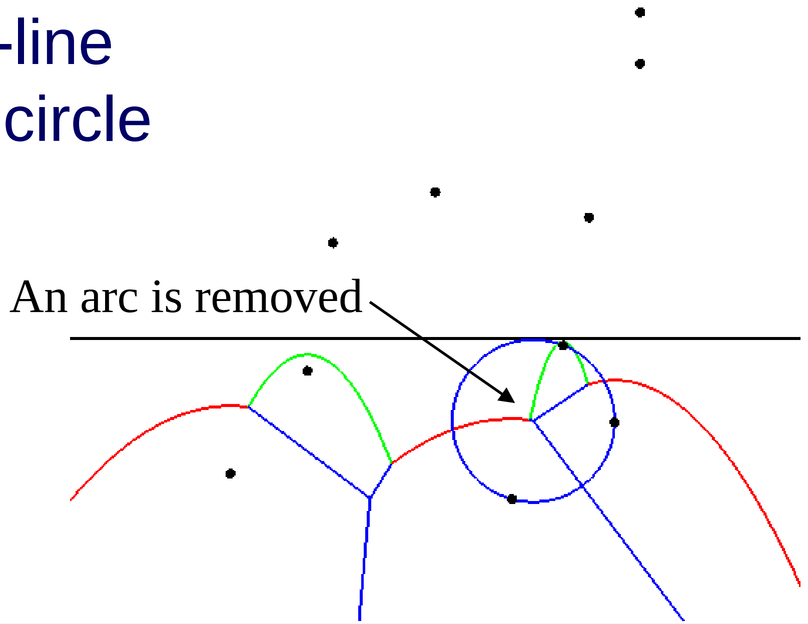
What are the events that change the topology of the beach-line?

- One parabolic arc overtakes another

  ⇒ A parabolic arc is removed

This corresponds to a new vertex in the Voronoi Diagram.

This occurs when the sweep-line crosses the top of the circumcircle through the three sites



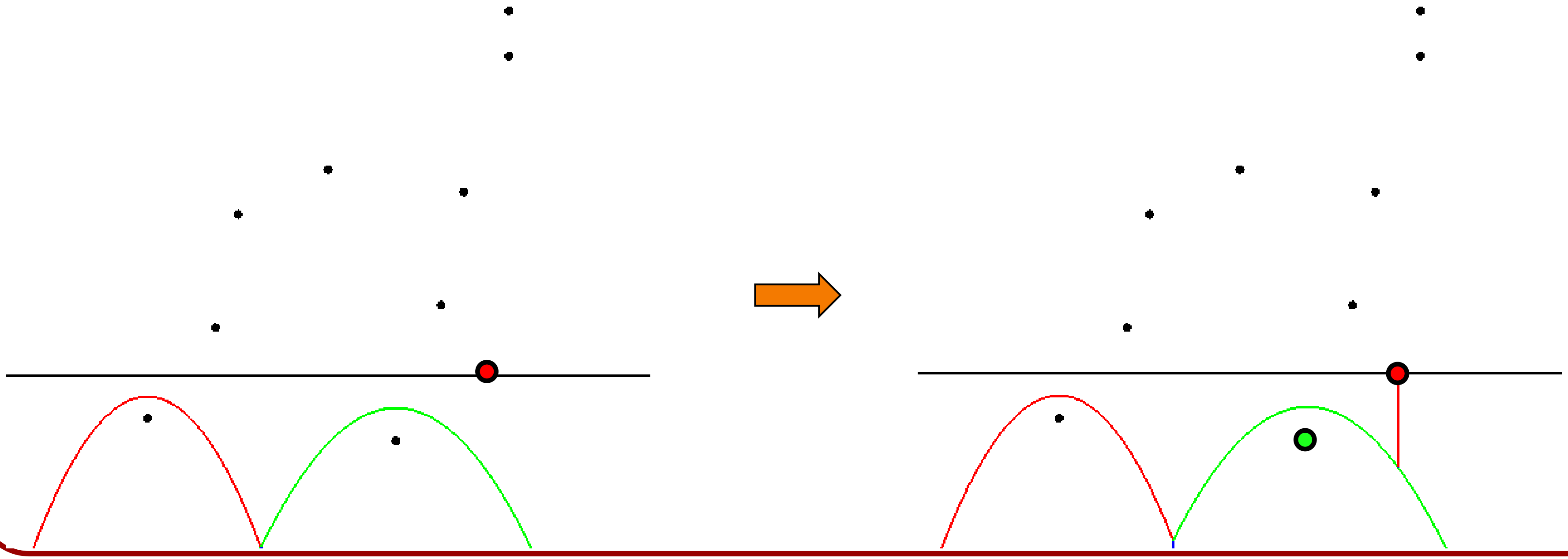
Outline of the Algorithm

- Create the faces of Voronoi diagrams (sites)
- Initialize the event-list of arc insertions/deletions with the sites
- Initialize an empty beach-line
- Iterate through the event list
  - If the next event is an insertion:
    - » …
  - If the next event is a deletion:
    - » …
- Compute the Delaunay Triangulation

Outline of the Algorithm

Insertion:

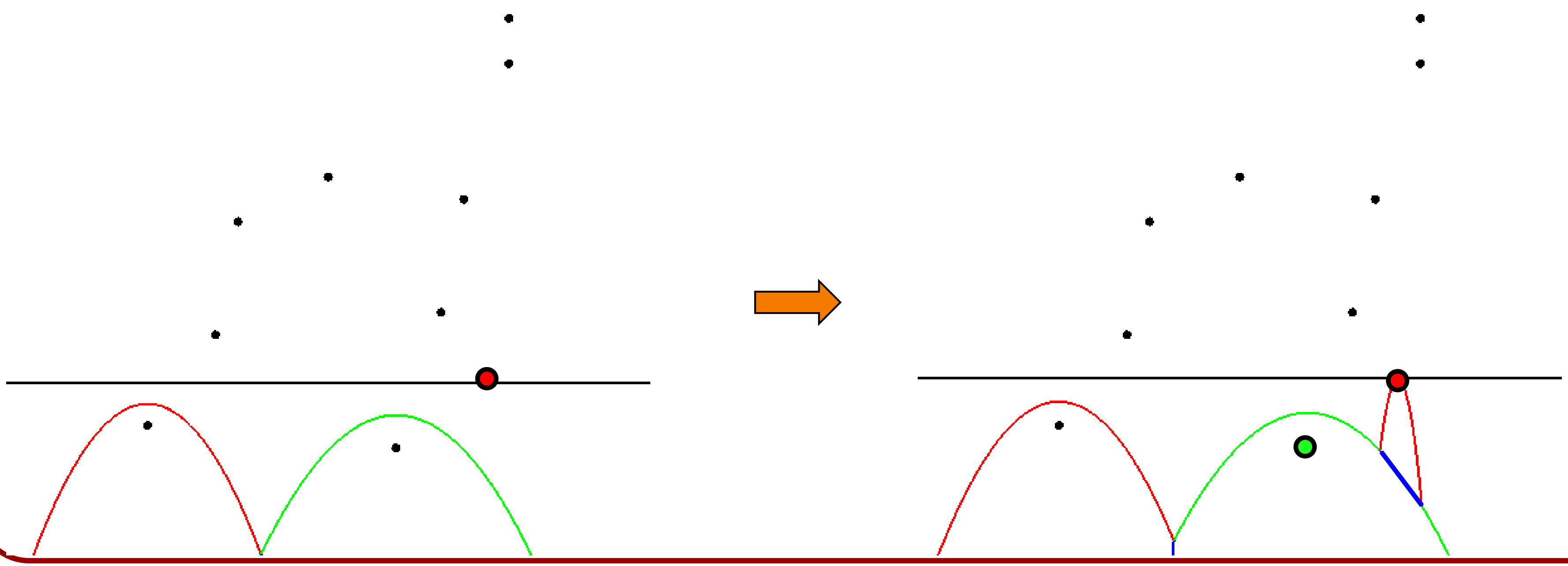
- Add an arc to the beach-line, splitting an old arc in two



Outline of the Algorithm

Insertion:

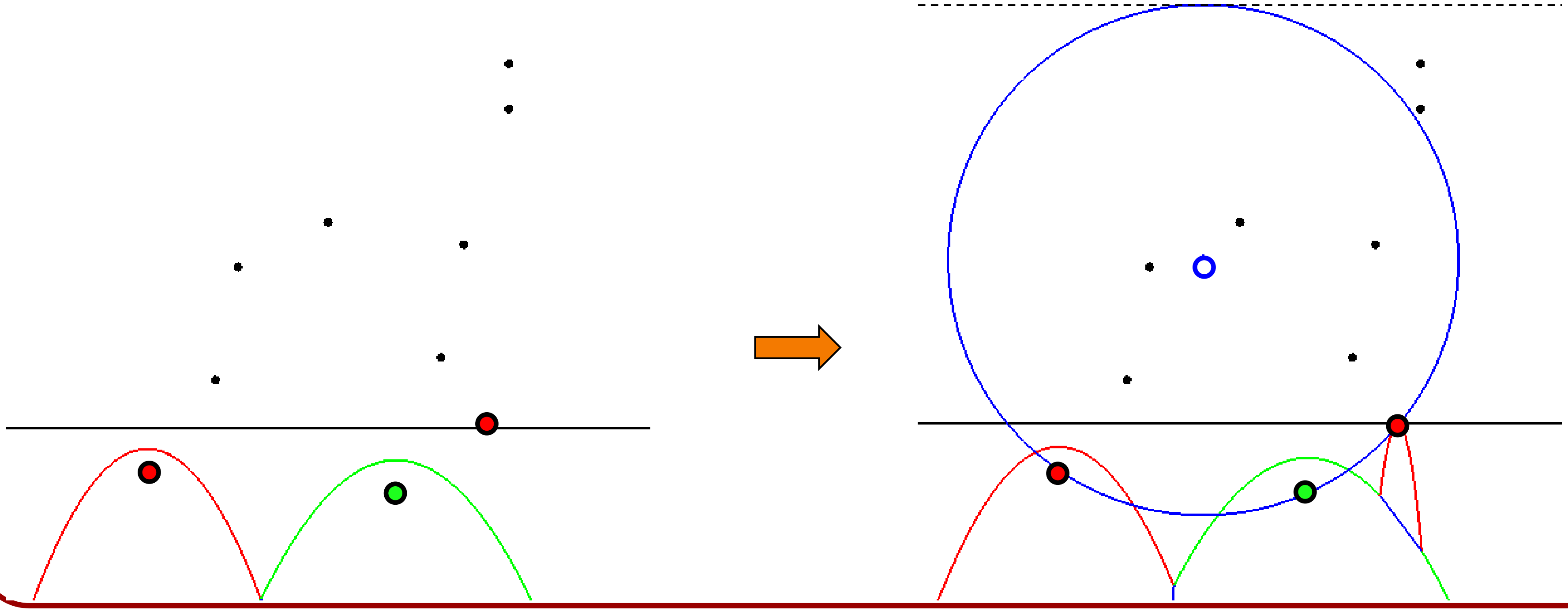
- Add an arc to the beach-line, splitting an old arc in two
- Add a Voronoi edge to the diagram



Outline of the Algorithm

Insertion:

- Add an arc to the beach-line, splitting an old arc in two
- Add a Voronoi edge to the diagram
- Check for potential deletion events with neighbors of the new arc and add to the event-list

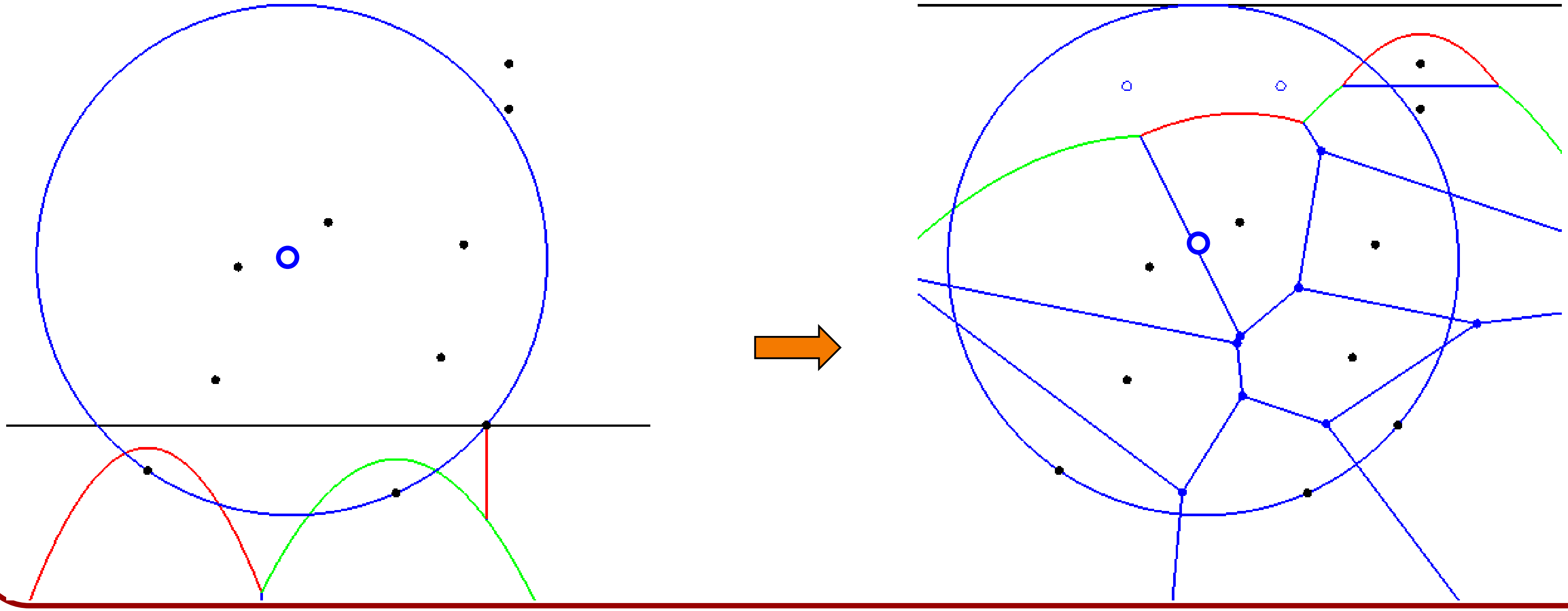


Outline of the Algorithm

Deletion:

- Check if the deletion event is valid

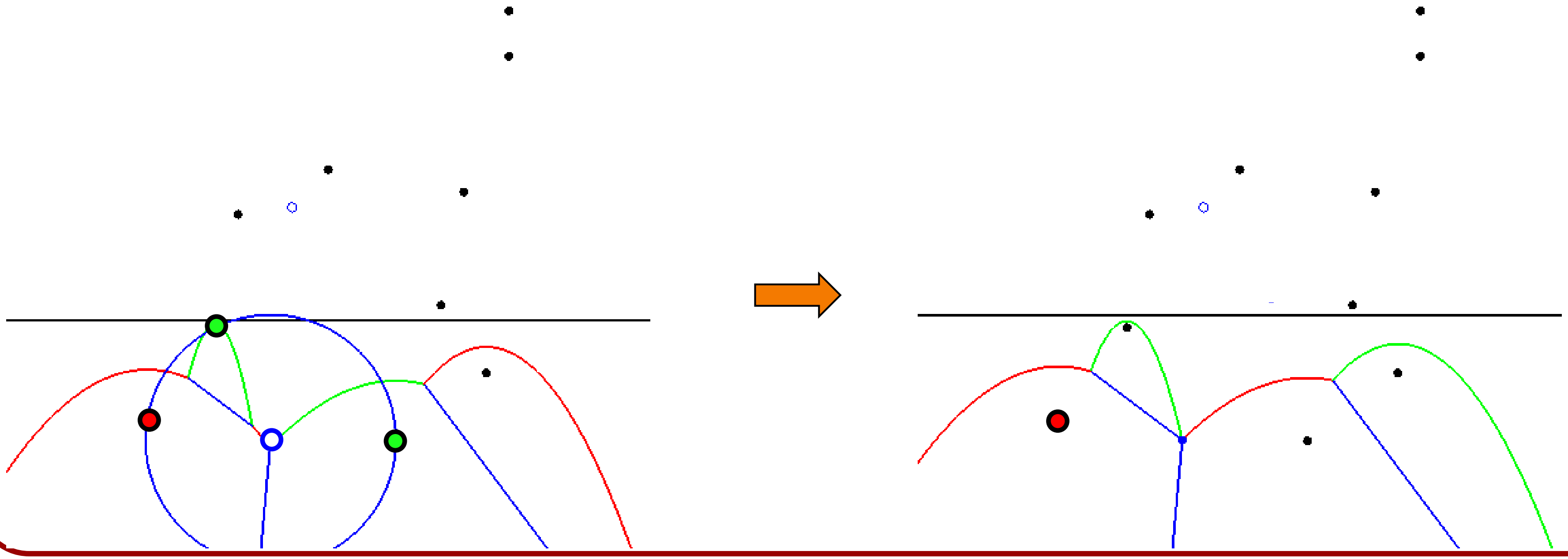
  If the circumcenter is behind the beach-line, that means there is some other site closer to the circumcenter than the original three sites that generated it.



Outline of the Algorithm

Deletion (if active):

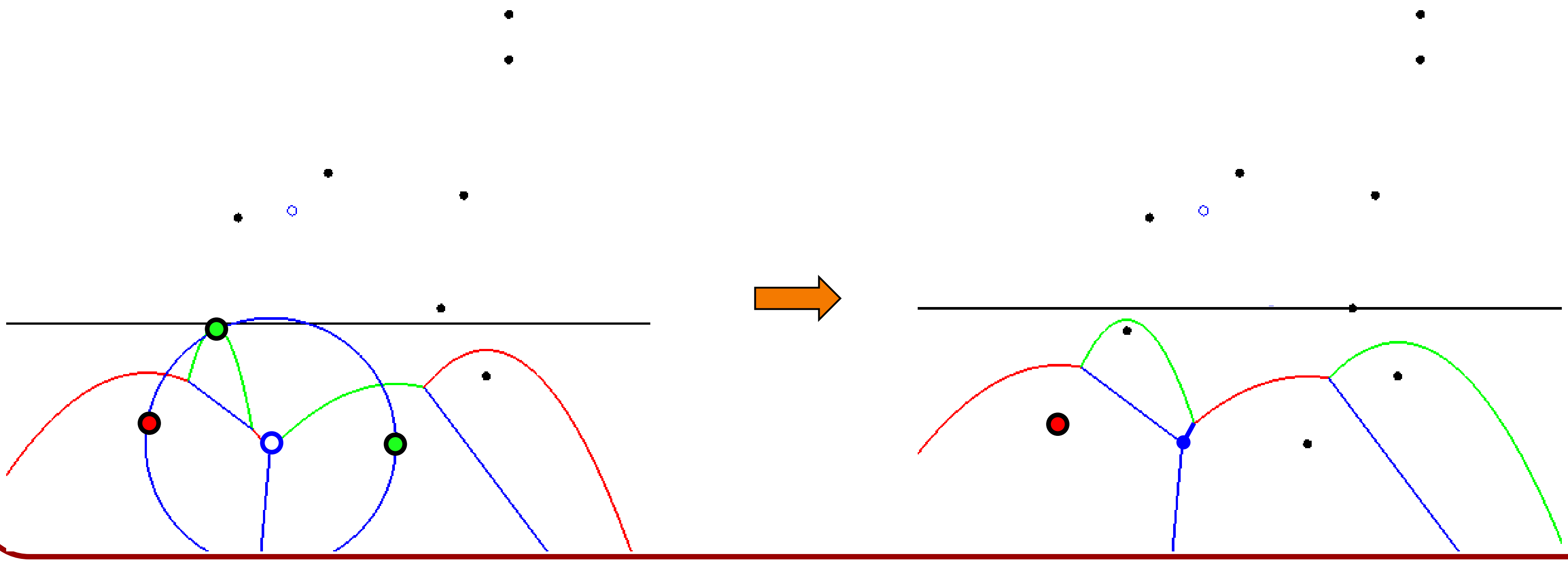
- Remove an arc from the beach-line



Outline of the Algorithm

<u>Deletion (if active)</u>:

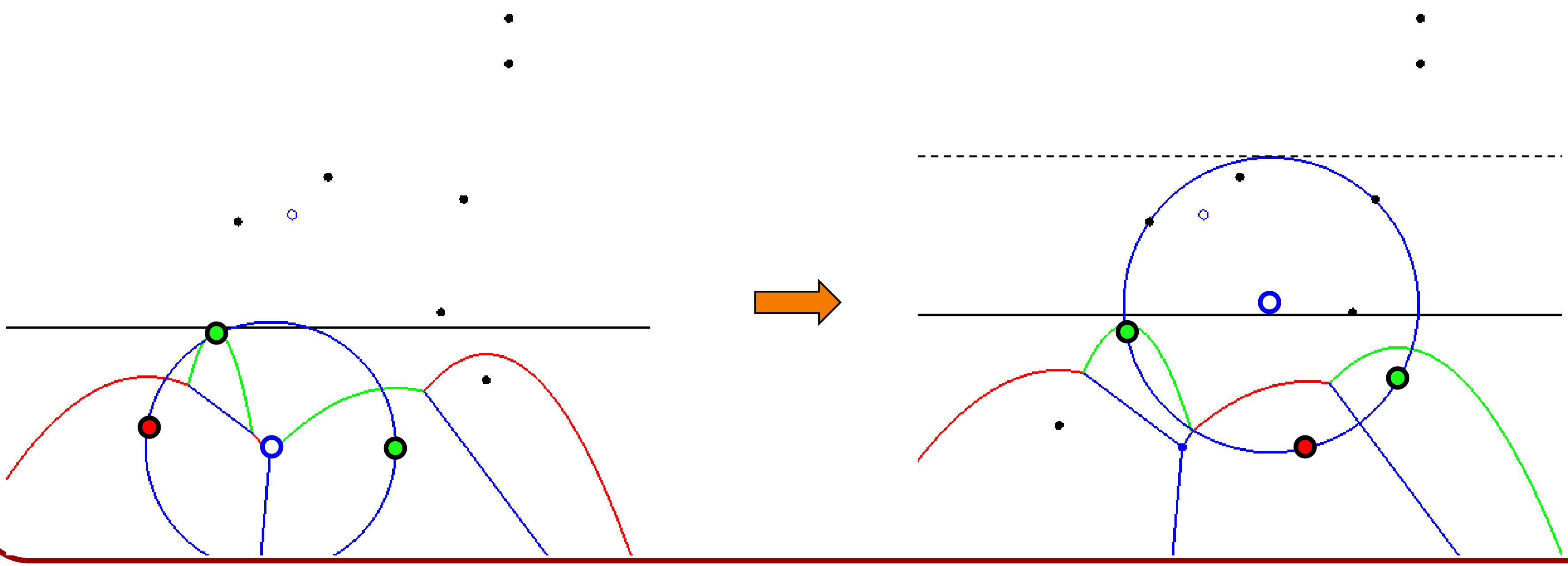
- Remove an arc from the beach-line
- Add a Voronoi vertex and edge into the diagram



Outline of the Algorithm

Deletion (if active):

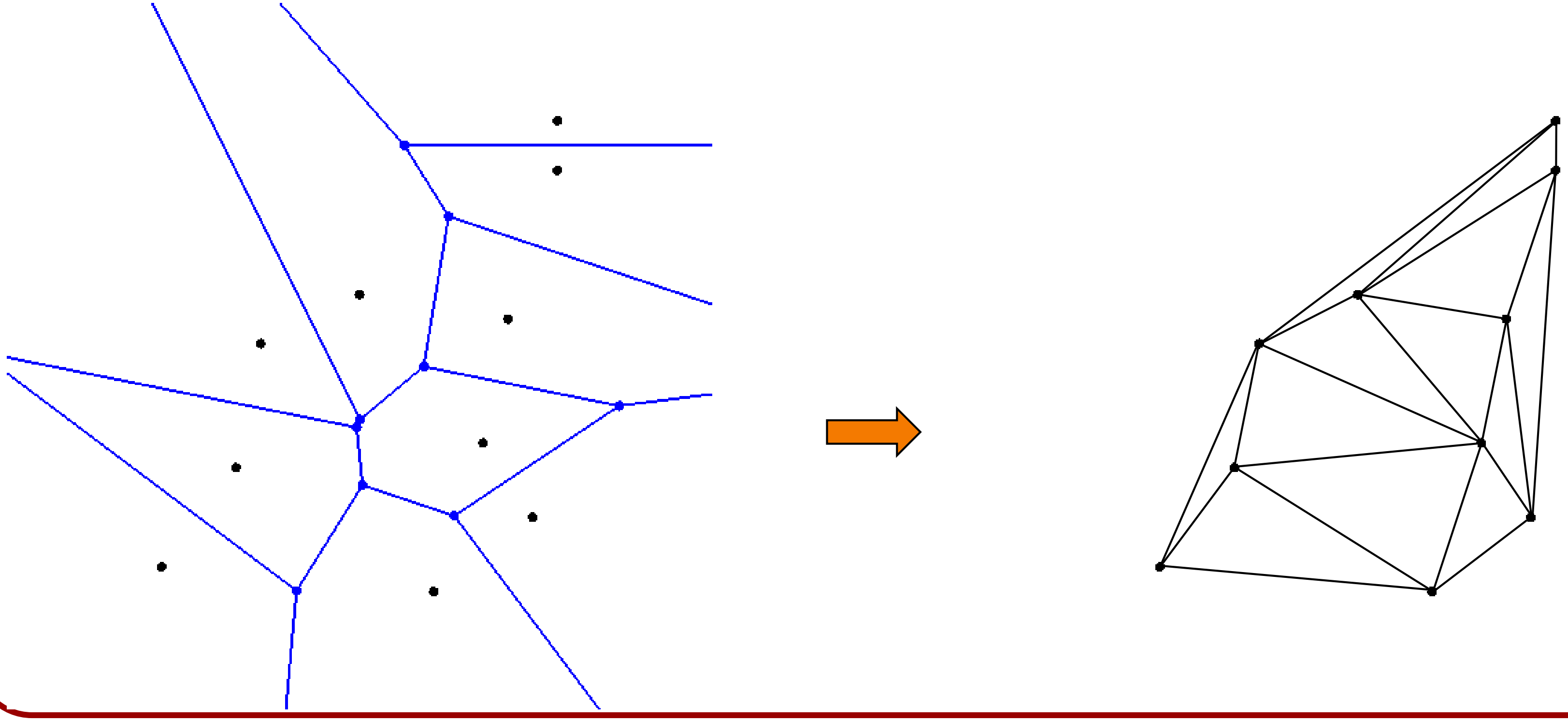
- Remove an arc from the beach-line
- Add a Voronoi vertex and edge into the diagram
- Check for potential deletion events with new neighbors of the arc and add to the event-list



Outline of the Algorithm

Computing the Delaunay Triangulation:

- Using duality, connect sites if the associated Voronoi faces share an edge.



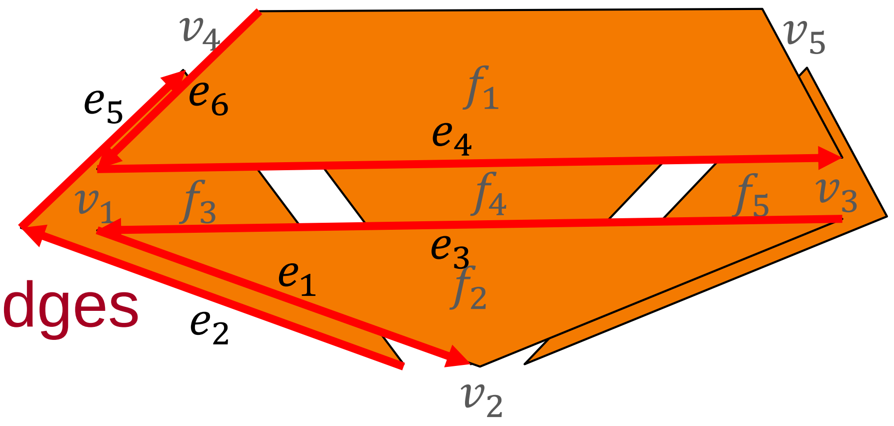
Outline

- Math Review
- Overview of the Algorithm
- Implementation
  - Half-edge data structure
  - Event-list representation
  - Beach-line representation
  - Handling the events

Half-Edge Data Structure

Recall:

- **Vertex Entry**:
  - » Vertex data
  - » Outgoing half-edge
- **Face Entry**:
  - » Face data
  - » Incident half-edge
- **Half-Edge Entry**:
  - » Half-edge data
  - » Previous/next half-edges
  - » Opposite half-edge
  - » Starting vertex
  - » Incident face



Half-Edge Data Structure

Useful to have a generic (template) class for half-edge data-structures and then instantiate with content type.

- Vertices will store data of type VData,
- Half-Edges will store data of type HEData,
- Faces will store data of type FData.

Half-Edge Data Structure

Pre-declare the objects since they will reference each other.

```
template< typename VData , typename HEData , typename FData >
struct HalfEdge
{
	struct V;		// The vertex
	struct HE;		// The half-edge
	struct F;		// The face
```

Half-Edge Data Structure

Declaring functions acting on vertices, half-edges, and faces

```
typedef std::function< void (         V * ) >       VFunctor;
typedef std::function< void ( const  V * ) >  ConstVFunctor;
typedef std::function< void (        HE * ) >      HEFunctor;
typedef std::function< void ( const HE * ) > ConstHEFunctor;
typedef std::function< void (         F * ) >       FFunctor;
typedef std::function< void ( const  F * ) >  ConstFFunctor;
```

Half-Edge Data Structure

Declaring the vertex structure

```
struct V
{
    HE *halfEdge;
    VData data;

    V( VData d=VData() );
    void processHalfEdges(          HEFunctor f );
    void processHalfEdges( ConstHEFunctor f ) const;
    void processFaces(          FFunctor f );
    void processFaces( ConstFFunctor f ) const;
};
```

Half-Edge Data Structure

Declaring the half-edge structure

```
struct HE
{
    HE *opposite , *previous , *next;
    V *startVertex;
    F *face;
    HEData data;

    HE( HEData d=HEData() );
};
```

Half-Edge Data Structure

Declaring the face structure

```
struct F
{
    HE *halfEdge;
    FData data;

    F( FData d=FData() );
    void processHalfEdges(         HEFunctor f );
    void processHalfEdges( ConstHEFunctor f ) const;
    void processVertices (          VFunctor f );
    void processVertices ( ConstVFunctor f ) const;
};
```

Half-Edge Data Structure

Defining the vertex constructor

```
template< typename VData , typename HEData , typename FData >
HalfEdge< VData , HEData , FData >::V::V( VData d )
    :
    data(d) , halfEdge(NULL)
    { }
```

Half-Edge Data Structure

Defining the vertex neighboring edge processor

```
template< typename VData , typename HEData , typename FData >
void HalfEdge< VData , HEData , FData >::V::processHalfEdges
( HalfEdge< VData , HEData , FData >::HEFunctor f )
{
    for( HE *he=halfEdge ; he ; he=he->opposite->next )
    {
        f( he );
        if( he->opposite->next==halfEdge ) break;
    }
}
```

Half-Edge Data Structure

Defining the vertex neighboring face processor

```
template< typename VData , typename HEData , typename FData >
void HalfEdge< VData , HEData , FData >::V::processFaces
( HalfEdge< VData , HEData , FData >::FFunctor f )
{
    processHalfEdges( [](HE *he ){ return f( he->face ); } );
}
```

Half-Edge Data Structure

Defining the half-edge constructor

```
template< typename VData , typename HEData , typename FData >
HalfEdge< VData , HEData , FData >::HE::HE( HEData d )
    :
    data(d) ,
    opposite(NULL) , previous(NULL) , next(NULL) ,
    startVertex(NULL) ,
    face(NULL)
    { }
```

Half-Edge Data Structure

Defining the face constructor

```
template< typename VData , typename HEData , typename FData >
HalfEdge< VData , HEData , FData >::F::F( FData d )
    :
    data(d) ,
    halfEdge(NULL)
    { }
```

Half-Edge Data Structure

Declaring vertex/edge/face data:

- Vertices should track the position of the circumcenter and its radius
- Half-edges don't need to track anything
- Faces should track the associated site and its index within the list of sites (for computing the Delaunay Triangulation)

```
struct VData{ Geometry::Point2d vertex ; double radius; };
struct HEData{ };
struct FData{ Geometry::Point2i site ; int index; };
```

Half-Edge Data Structure

For simplicity of notation, `typedef` half-edge elements to something concise.

```
typedef typename HalfEdge< VData , HEData , FData >::V Vertex;
typedef typename HalfEdge< VData , HEData , FData >::HE HalfEdge;
typedef typename HalfEdge< VData , HEData , FData >::F Face;
```

Outline

- Math Review
- Overview of the Algorithm
- Implementation
  - Half-edge data structure
  - Event-list representation
  - Beach-line representation
  - Handling the events

Event-List Representation

Challenges:

1. The event list is dynamic, with deletion events introduced while sweeping
2. The event list must support two different types of events – insertions and deletions

Event-List Representation

Approach:

1. Use a balanced binary tree (e.g. `std::set`) to represent the event-list
2. Have each event know its own type

Event-List Representation

An EventList object is a balanced binary tree containing Event objects

```
struct Event;
typedef std::function< bool ( const Event & , const Event & ) > EventComparator;
struct EventList : public std::set< Event , EventComparator >
{
    ...
};
```

Event-List Representation

An Event object is either an Insertion or Deletion object

```
struct Event
{
    struct Insertion { ... };
    struct Deletion { ... }
    enum Type { INSERTION , DELETION };
    Type type;
    union { Insertion insertion ; Deletion deletion; }
    static bool Compare( const Event &e1 , const Event &e2 );
};
```

Event-List Representation

An Insertion object corresponds to a Voronoi face

```
struct Event
{
    struct Insertion
    {
        Face *face;
        double eventTime( void ) const;
    };
};
```

Event-List Representation

A **Deletion** object corresponds to a circumcircle of three sites (i.e. Voronoi faces)

```
struct Event
{
    struct Deletion
    {
        Face *face1 , *face2 , *face3;
        Geometry::Point2d center( void ) const;
        double radius( void ) const;
        double eventTime( void ) const;
    };
};
```

Event-List Representation

An *Insertion* event happens when the sweep-line passes through the site

```
double Event::Insertion::eventTime( void ) const
{
    return face->data.site[1];
}
```

Event-List Representation

A **Deletion** event happens when the sweep-line passes through the top of the circumcircle

```
double Event::Deletion::eventTime( void ) const
{
	return center()[1] + radius();
}
```

Event-List Representation

Events can be ordered by their event-time

```
bool Event::Compare( const Event &e1 , const Event &e2 )
{
	double t1 , t2;
	if( e1.type==INSERTION ) t1 = e1.insertion.eventTime();
	else                     t1 = e1.deletion.eventTime();
	if( e2.type==INSERTION ) t2 = e2.insertion.eventTime();
	else                     t2 = e2.deletion.eventTime();
	return t1 < t2;
}
```

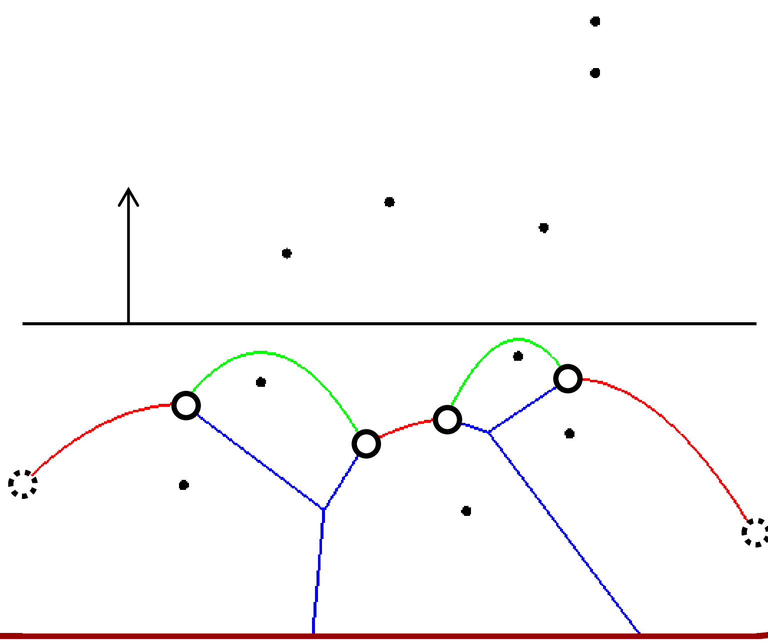
Outline

- Math Review
- Overview of the Algorithm
- Implementation
  - Half-edge data structure
  - Event-list representation
  - Beach-line representation
  - Handling the events

Beach-Line Representation

One way to represent a beach-line is by the end-points of the parabolic arcs:

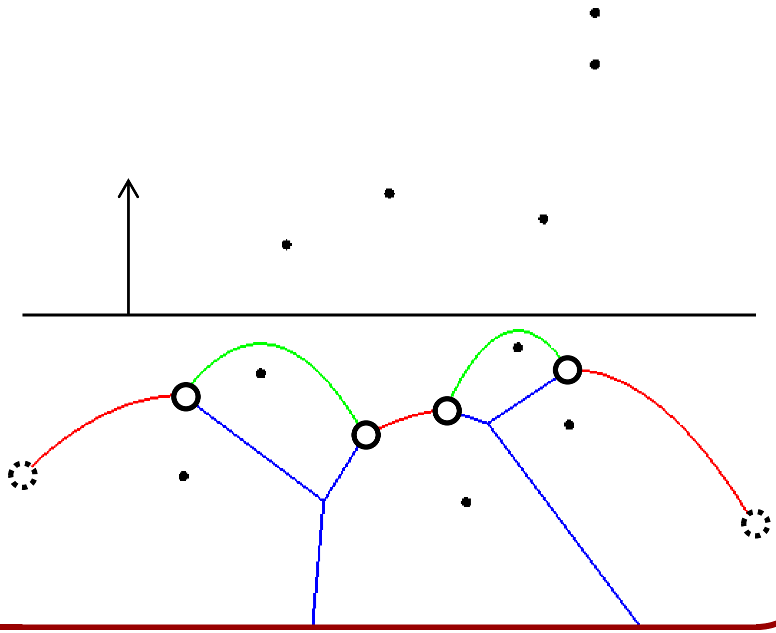
```
struct EndPoint
{
    static double SweepLineHeight;
    Face *leftFace , *rightFace;
    HalfEdge *left , *right;
    double x( void ) const;
};
```



Beach-Line Representation

To compute the x-coordinate of the end-point, need to construct parabolic fronts for the left and right sites and see where they intersect

```
double EndPoint::x( void ) const
{
    …
}
```



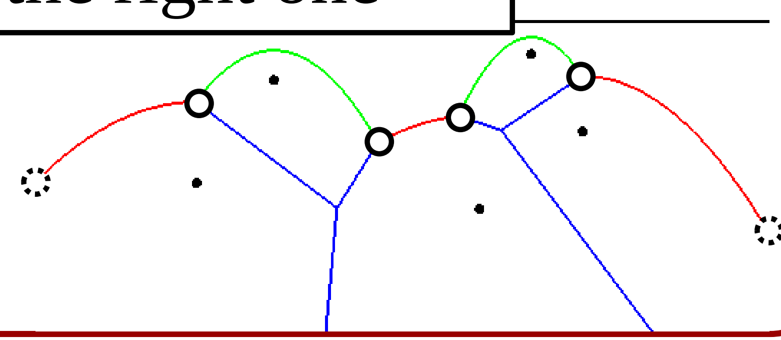
Beach-Line Representation

To compute the x-coordinate of the end-point, need to construct parabolic fronts for the left and right sites and see where they intersect

```
double End
{
    ...
}
```

Note:
The left/right face could be NULL if it bounds left/right-most arc in the beach-line

Warning:
The two parabolas will intersect at two points. Need to choose the right one



Beach-Line Representation

When an Insertion event occurs we will want to query the beach line using the x-position of the site. (The end-points of the arc will not be known at the time.)

```
struct XPosition
{
	double x;
};
```

Beach-Line Representation

As with the Event object, a beach-line element object will be either an EndPoint or a Position

```
struct BLElement
{
    struct EndPoint{ …. };
    struct XPosition{ … };
    enum Type{ END_POINT , POSITION };
    Type type;
    union { EndPoint endPoint ; Position position; }
    static bool Compare( const BLElement &e1 , const BLElement &e2 );
};
```

Beach-Line Representation

BLElements can be ordered by their positions

```
bool BLElement::Compare( const BLElement &e1 , const BLElement &e2 )
{
    double t1 , t2;
    if( e1.type==END_POINT ) t1 = e1.endPoint.x();
    else                     t1 = e1.position.x;
    if( e2.type==END_POINT ) t2 = e2.endPoint.x();
    else                     t2 = e2.position.x;
    return t1 < t2;
}
```

Beach-Line Representation

The beach-line itself can be represented as a balanced binary tree of BLElements

```
typedef std::function< bool ( const BLElement & , const BLElement & ) > BLElementComparator;
struct BeachLine : public std::set< BLElement , BLElementComparator >
{
    bool sanityCheck( void ) const;
    bool isActive( Geometry::Point2d p ) const;
};
```

Beach-Line Representation

Confirm that the beach line is consistent

```
bool BeachLine::sanityCheck( void ) const
{
    for( auto iter=begin() ; iter!=end() ; iter++ )
    {
        auto next = std::next( iter );
        if( iter!=end() )
            if( iter->endPoint.rightFace!=next->endPoint.leftFace ) return false;
        ...
    }
}
```

Beach-Line Representation

A 2D point is not active, if its y-coordinate is less than the height of the parabolic arc over its x-coordinate

```
bool BeachLine::isActive( Geometry::Point2d p ) const
{
    BLElement e;
    e.type = POSITION;
    e.position.x = p[0];
    auto upperIter = upper_bound( e );
    auto lowerIter = std::prev( upperIter );
    ...
}
```

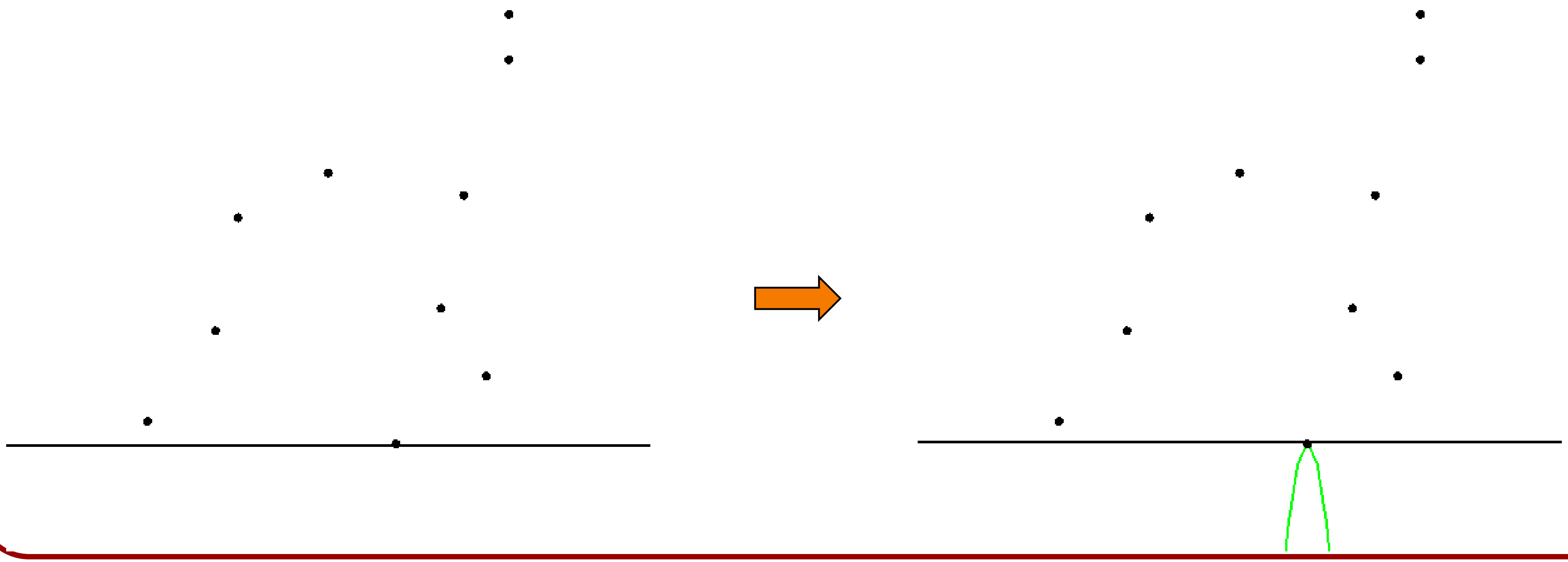
Outline

- Math Review
- Overview of the Algorithm
- Implementation
  - Half-edge data structure
  - Event-list representation
  - Beach-line representation
  - Handling the events

Handling the Events

Insertion (first event):

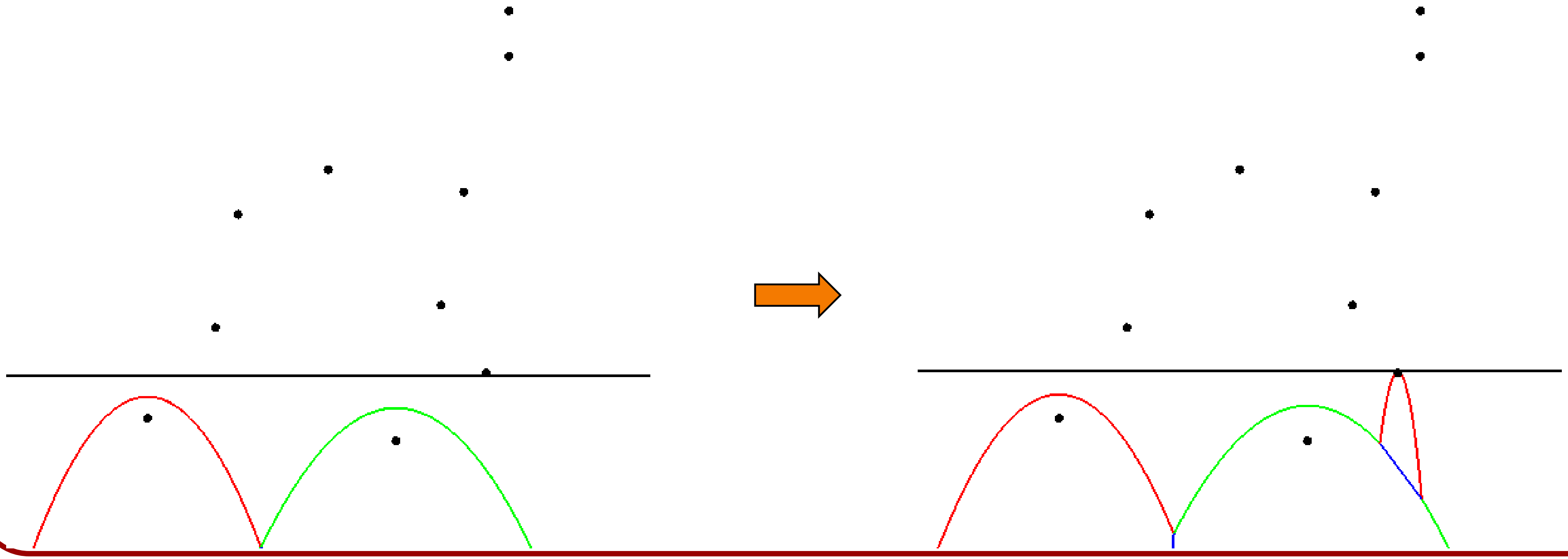
- Create two *EndPoint* objects and add them into the *BeachLine* object
  - » The left one should have a left *NULL* face and the right one should have a right *NULL* face



Handling the Events

Insertion (subsequent event):

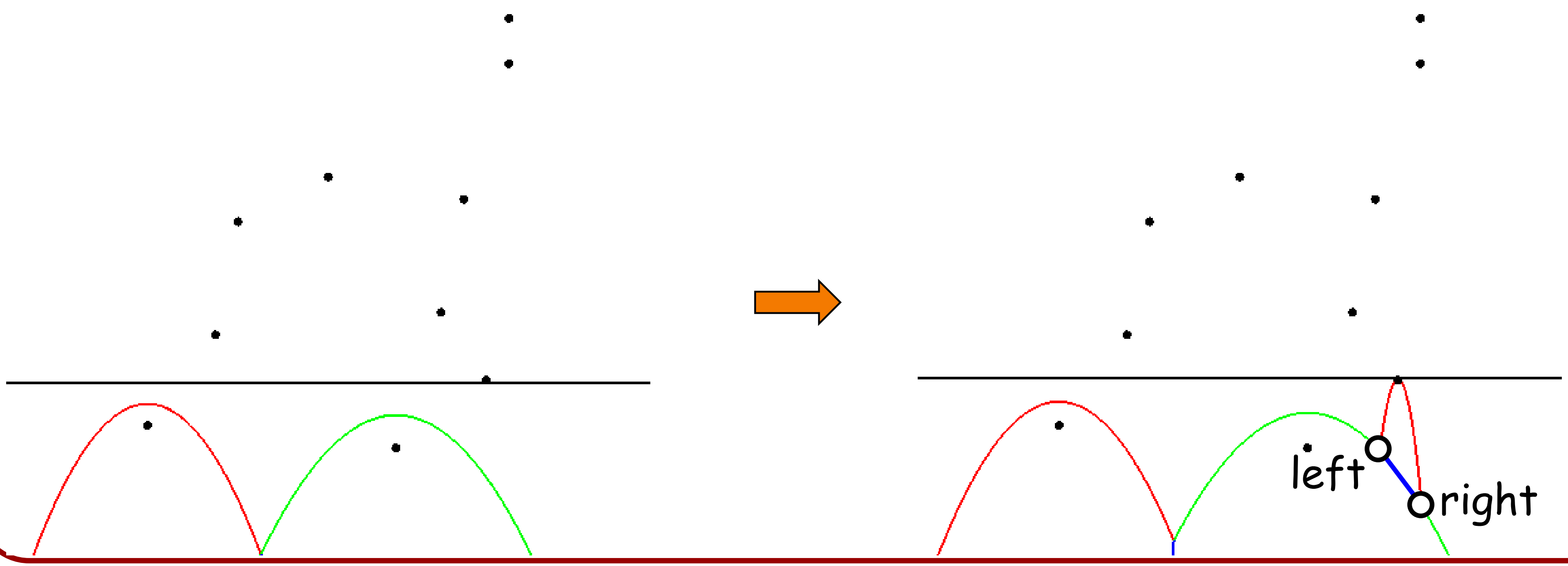
- Find the arc that will be split
  - » Use `std::set::upper_bound`



Handling the Events

Insertion (subsequent event):

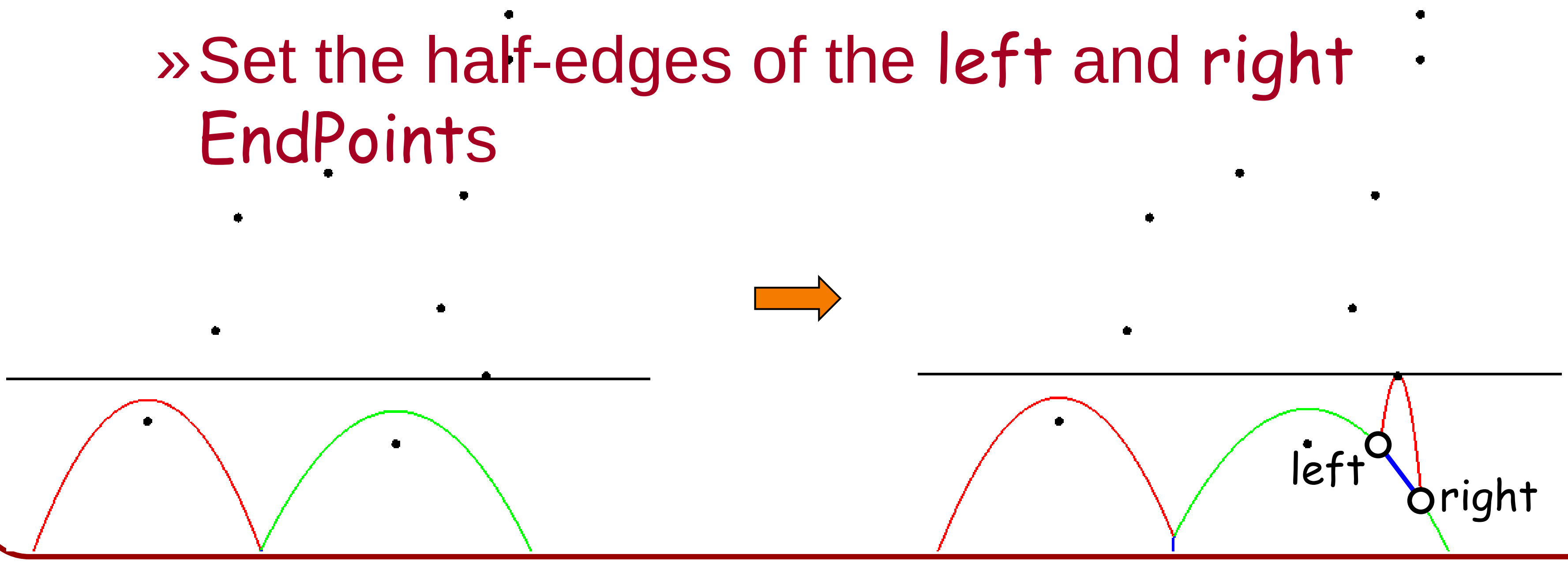
- Find the arc that will be split
  - » Use std::set::upper_bound
- Create the left and right EndPoint objects and add them into the beach-line



Handling the Events

Insertion (subsequent event):

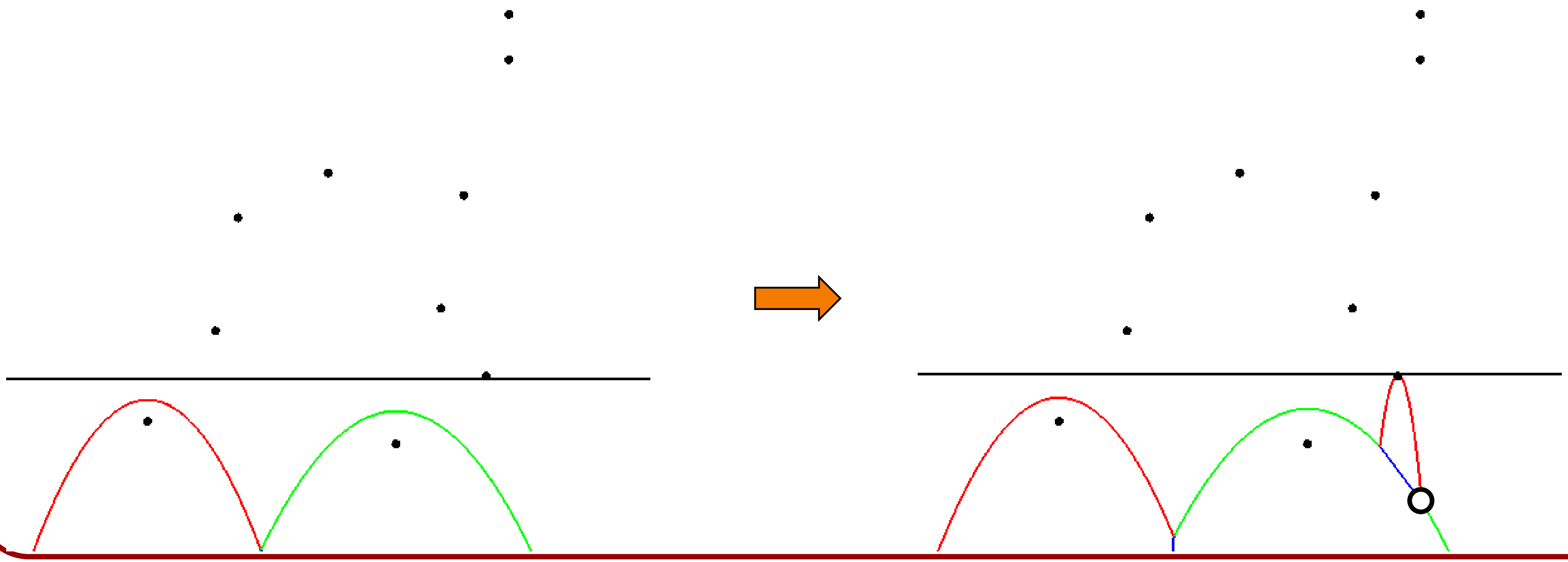
- Add a Voronoi edge to the diagram
  - » Create the half-edge and its opposite
  - » Make them point to each other
  - » Set the incident faces
  - » Set the half-edges of the *left* and *right* *EndPoints*



Handling the Events

Insertion (subsequent event):

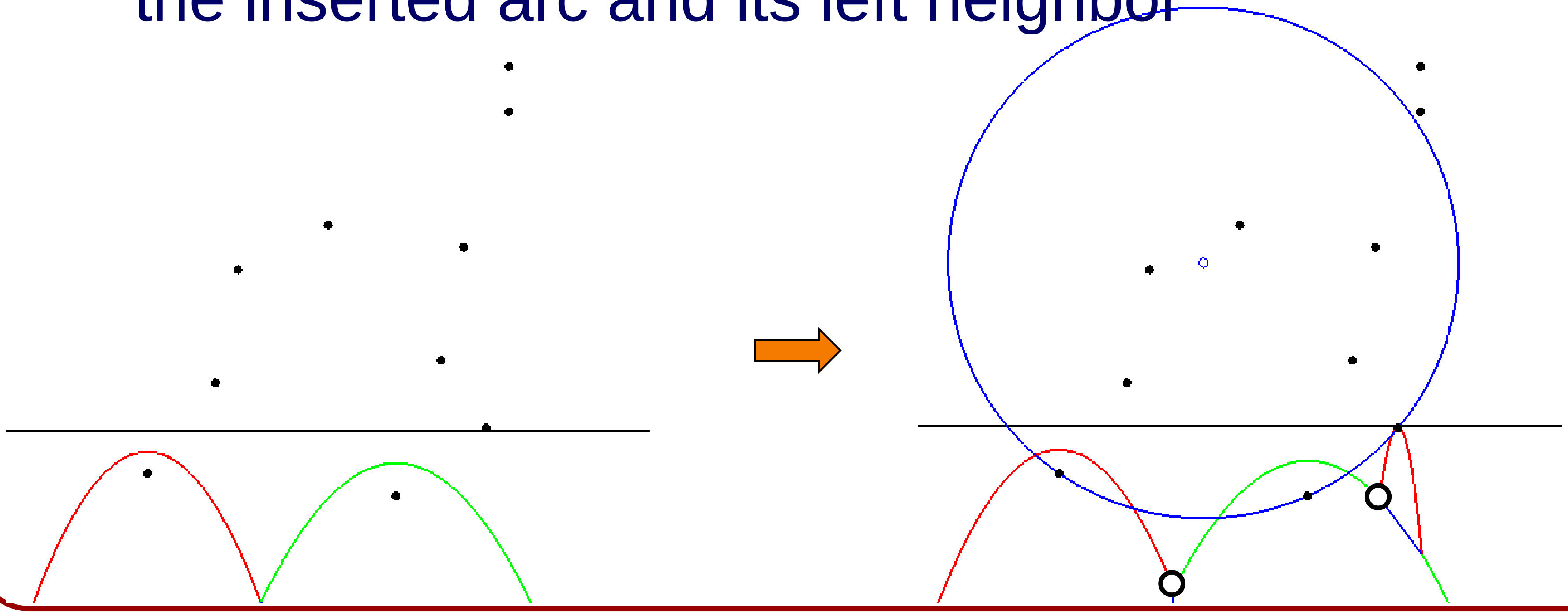
- Try creating a *Deletion* using the right end-point of the inserted arc and its right neighbor



Handling the Events

Insertion (subsequent event):

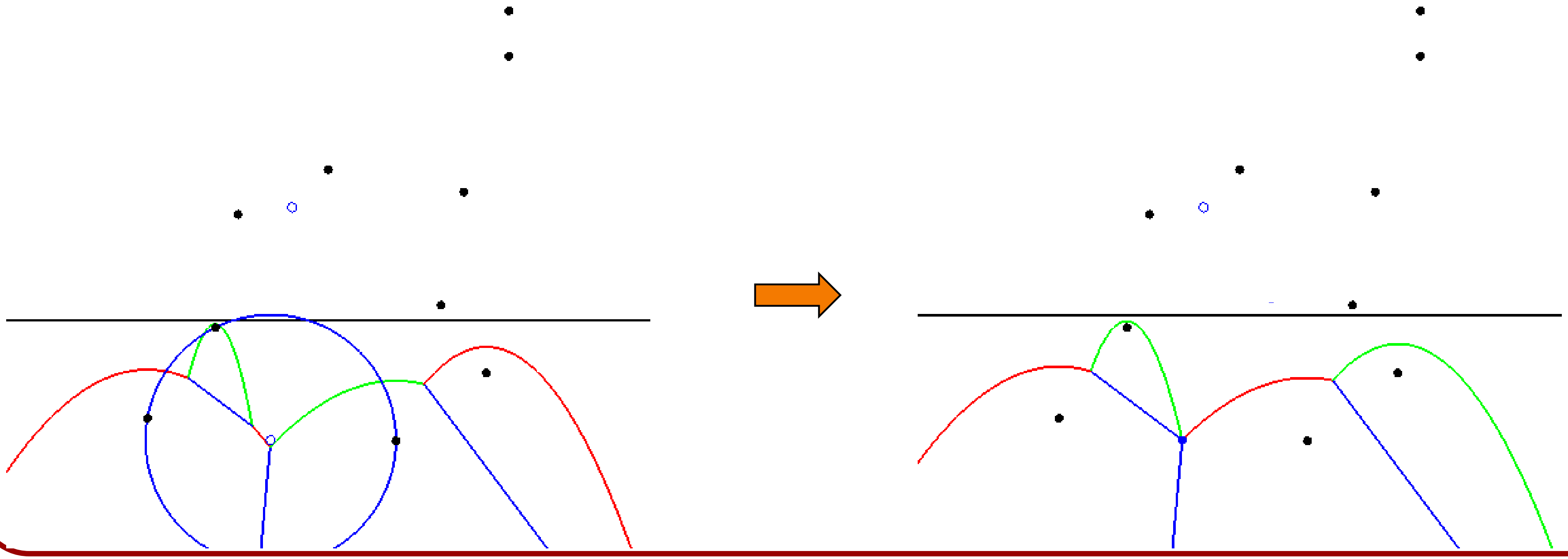
- Try creating a Deletion using the right end-point of the inserted arc and its right neighbor
- Try creating a Deletion using the left end-point of the inserted arc and its left neighbor



Handling the Events

Deletion:

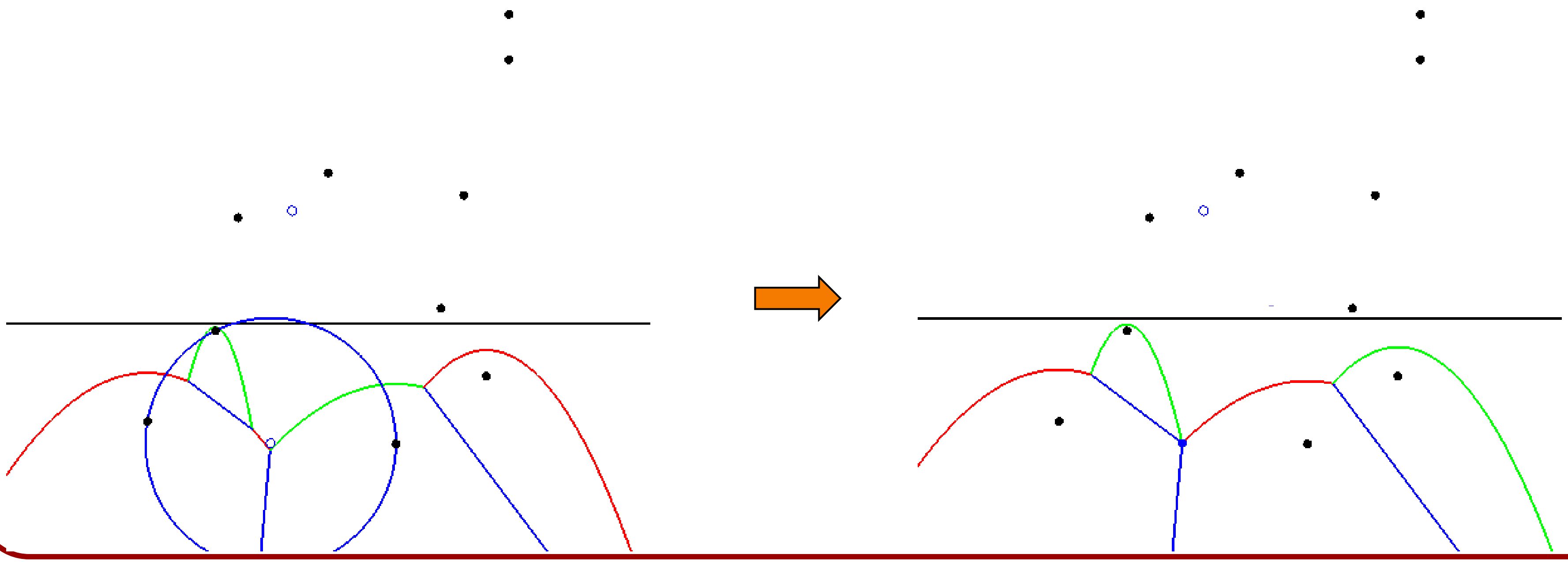
- Check if the circumcenter was just ahead of the beach-line



Handling the Events

Deletion:

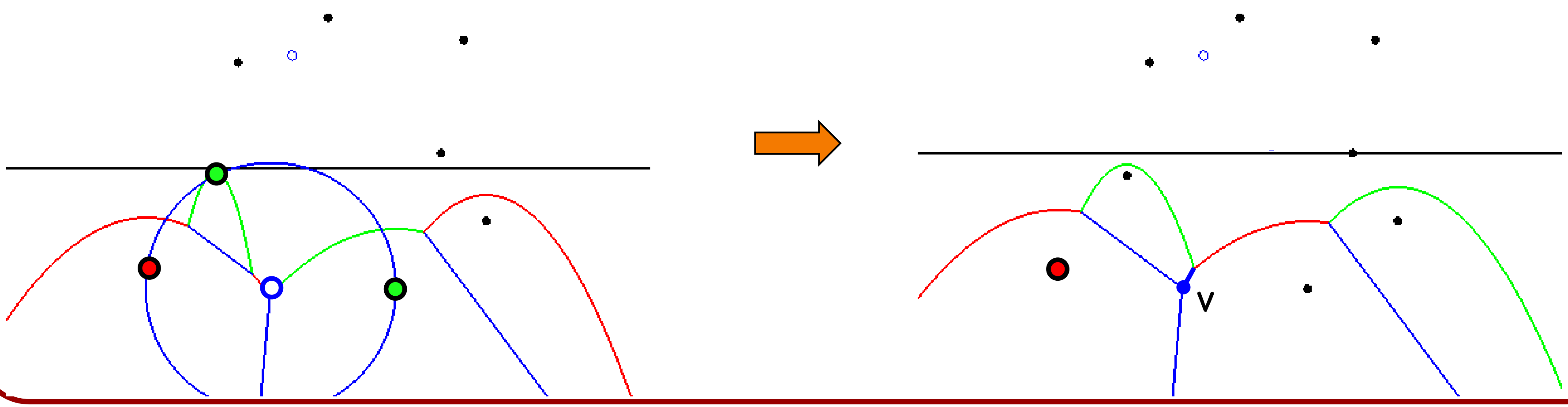
- Use the three sites to compute the three EndPoints that meet at the circumcenter
  - » Remove the two already in the beach-line and insert the third



Handling the Events

<u>Deletion</u>:

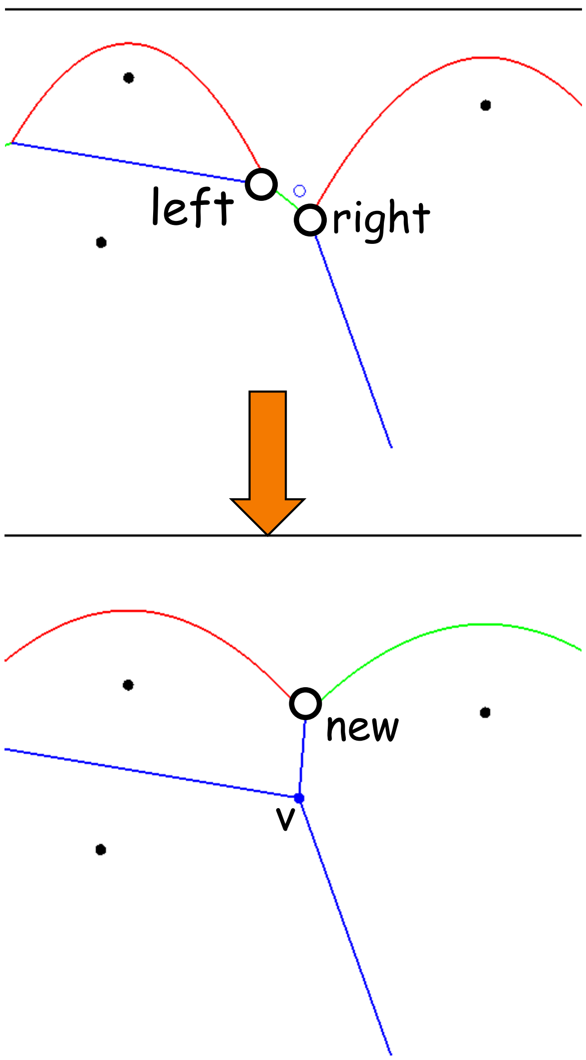
- Add a Voronoi vertex and edge into the diagram
  - » Create the new Voronoi vertex, *v*
  - » Create the half-edge and its opposite, *new.left* and *new.right*
  - » Link up everything



Handling the Events

Deletion:

» Link up everything



Handling the Events

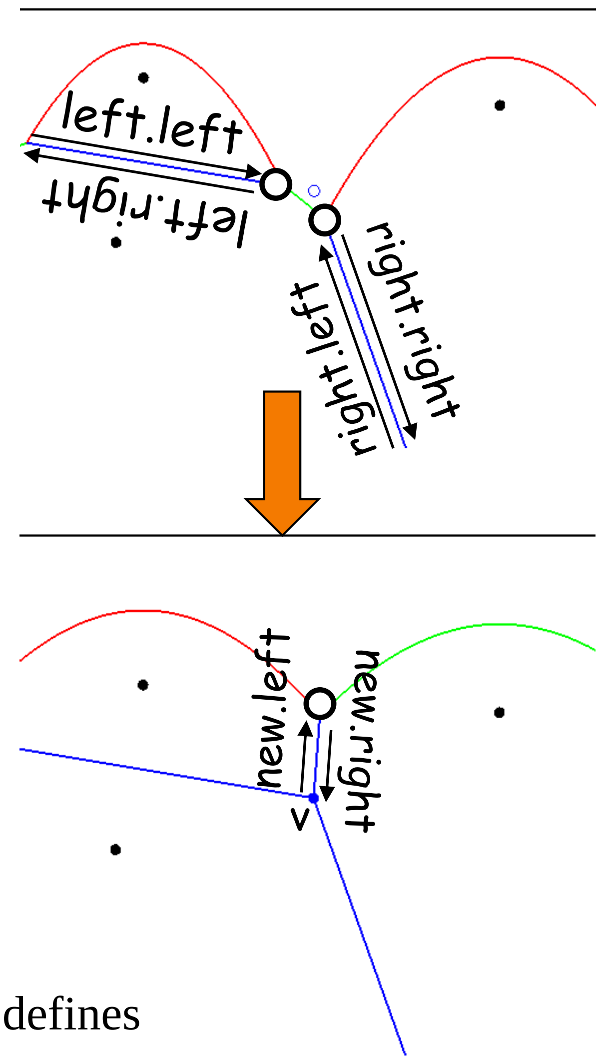
Deletion:

» Link up everything

– Two half-edges are associated with the old left end-point

– Two half-edges are associated with the old right end-point

– Two half-edges are associated with the new end-point



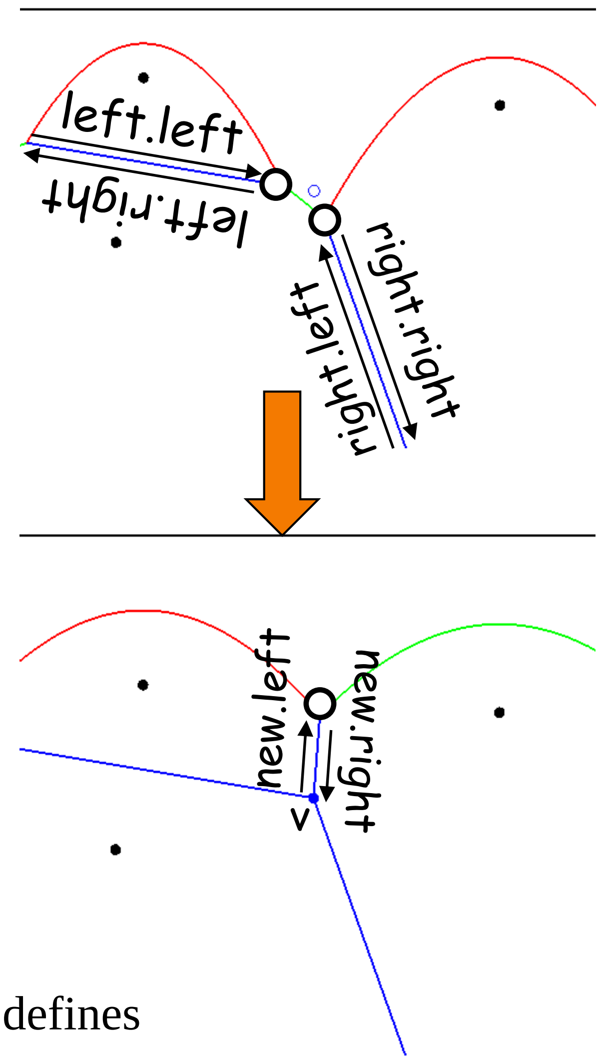
*A half-edge is left/right of an end-point if its associated face defines the left/right arc at the end-point.

Handling the Events

Deletion:

» Link up everything

- right.right->startVertex=v
- left.right->startVertex=v
- new.left->startVertex=v



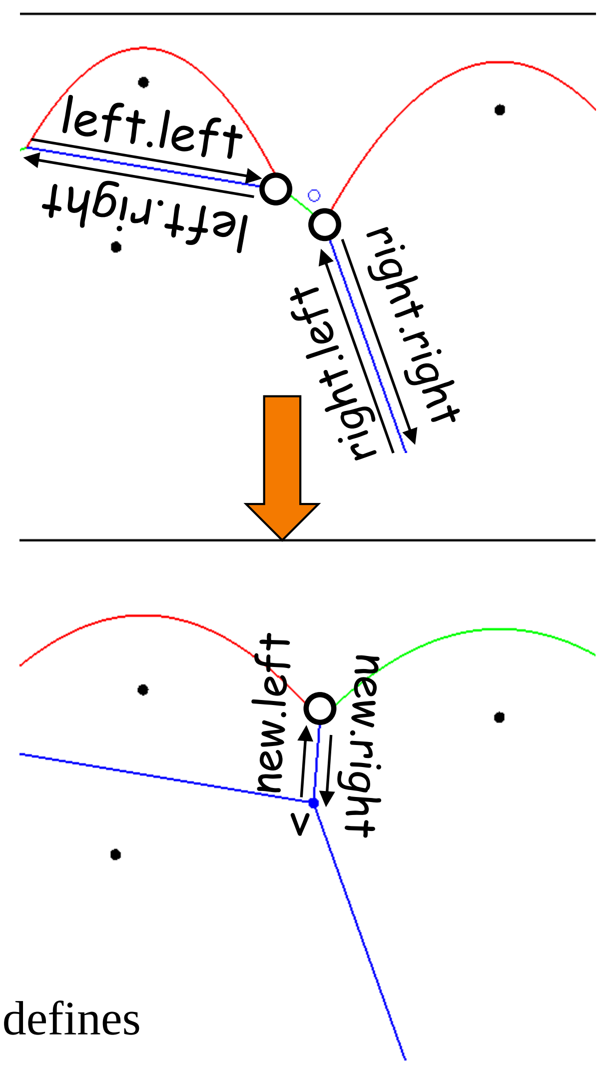
*A half-edge is left/right of an end-point if its associated face defines the left/right arc at the end-point.

Handling the Events

<u>Deletion</u>:

» Link up everything

- v->startVertex=new.left



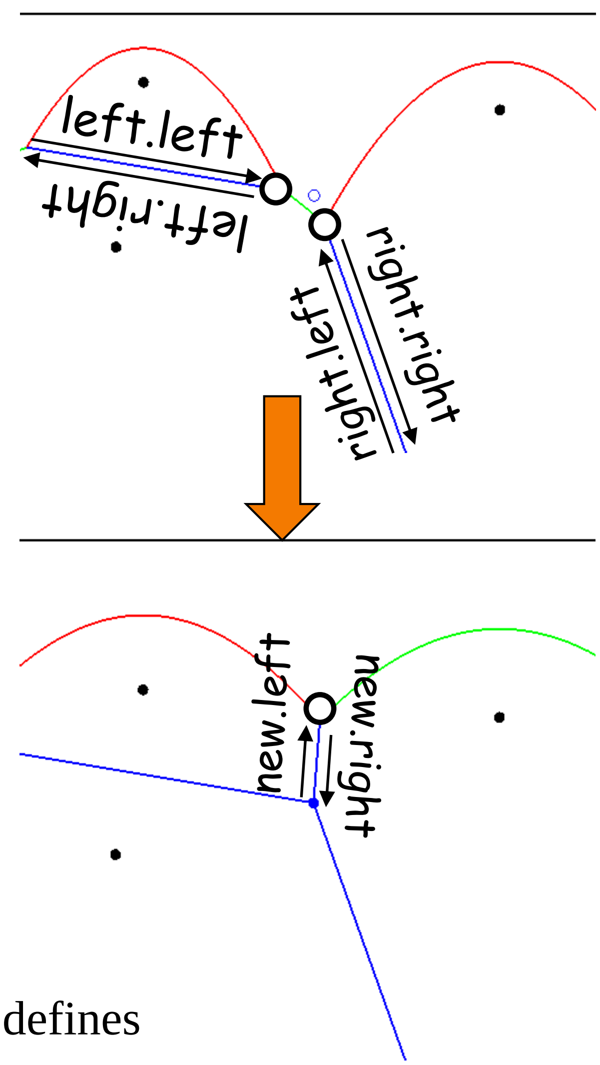
*A half-edge is left/right of an end-point if its associated face defines the left/right arc at the end-point.

Handling the Events

Deletion:

» Link up everything

- new.left->opposite=new.right
- new.right->opposite=new.left
- new.left->prev=left.left
- left.left->next=new.left
- new.right->next=right.right
- right.right->prev=new.right
- right.left->next=left.right
- left.right->next=right.left

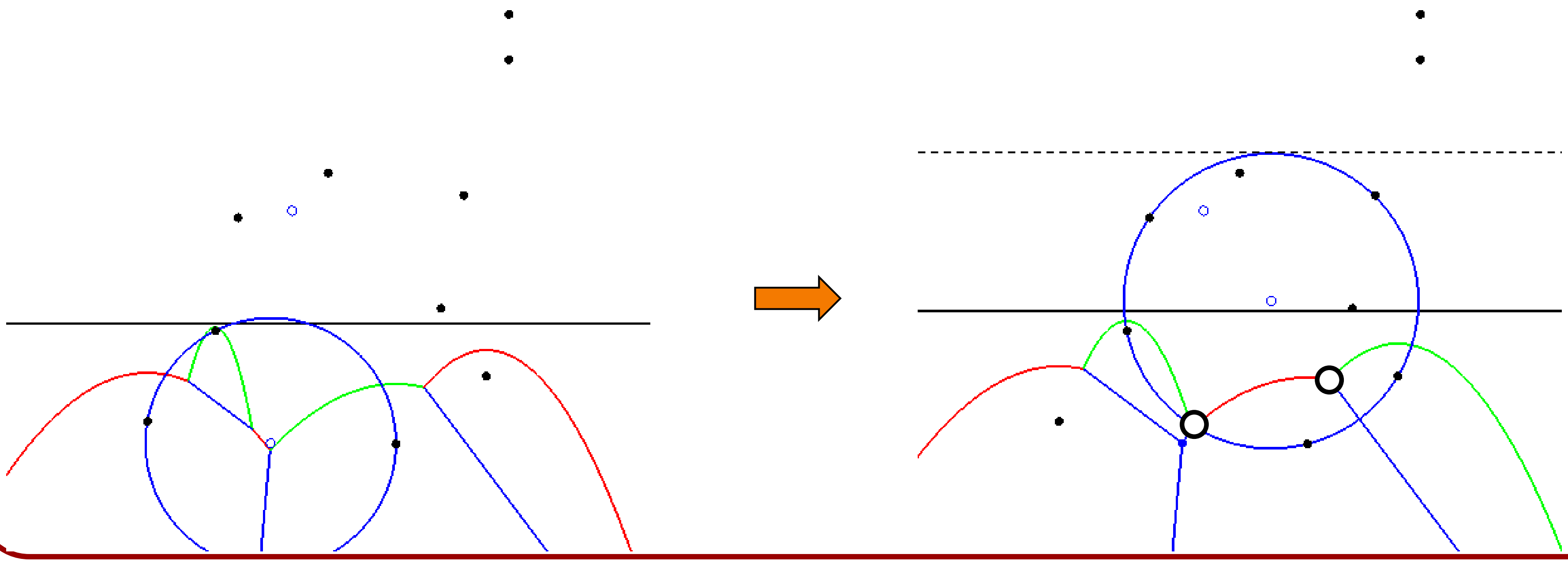


*A half-edge is left/right of an end-point if its associated face defines the left/right arc at the end-point.

Outline of the Algorithm

Deletion (if active):

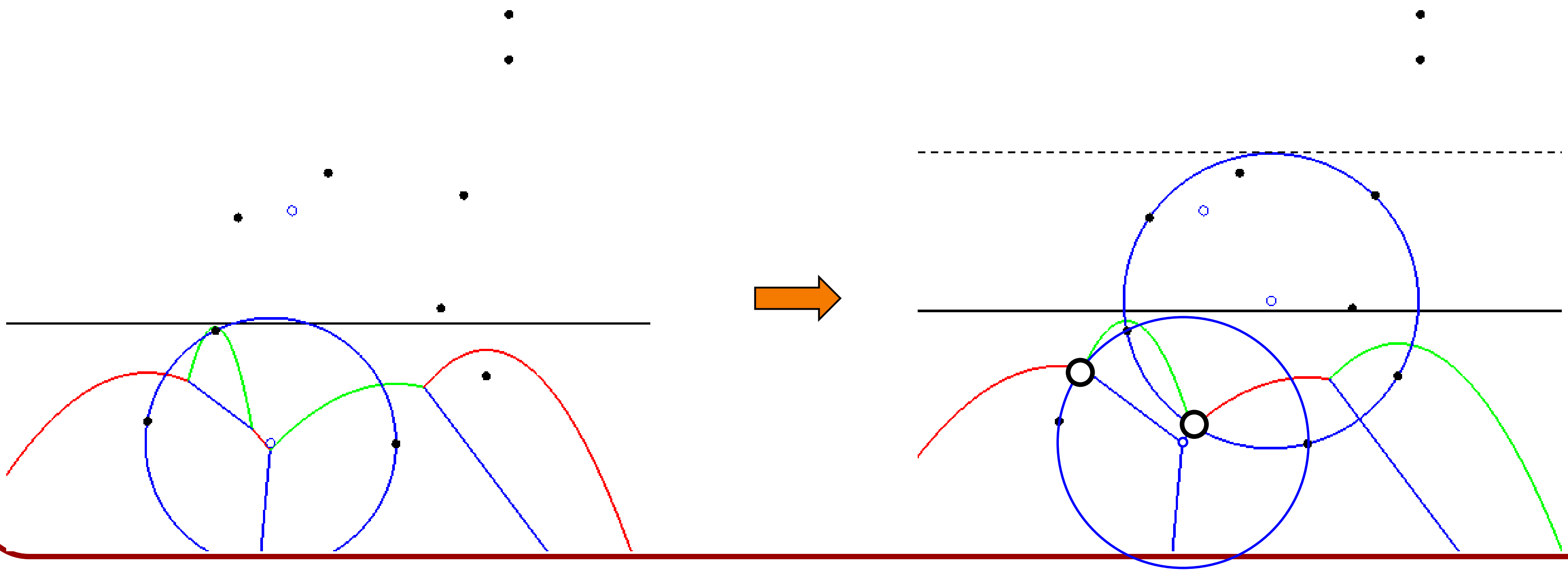
- Try creating a *Deletion* using the new end-point and its right neighbor



Outline of the Algorithm

<u>Deletion (if active)</u>:

- Try creating a Deletion using the new end-point and its right neighbor
- Try creating a Deletion using the new end-point and its left neighbor



Outline of the Algorithm

Warnings:

- As with the triangulation code, you need to be careful about whether you want to be just above or just below the sweep-line as you update the beach-line.
- Unlike the triangulation code, it's hard to get by with integer arithmetic because deletion events can happen at irrational heights.