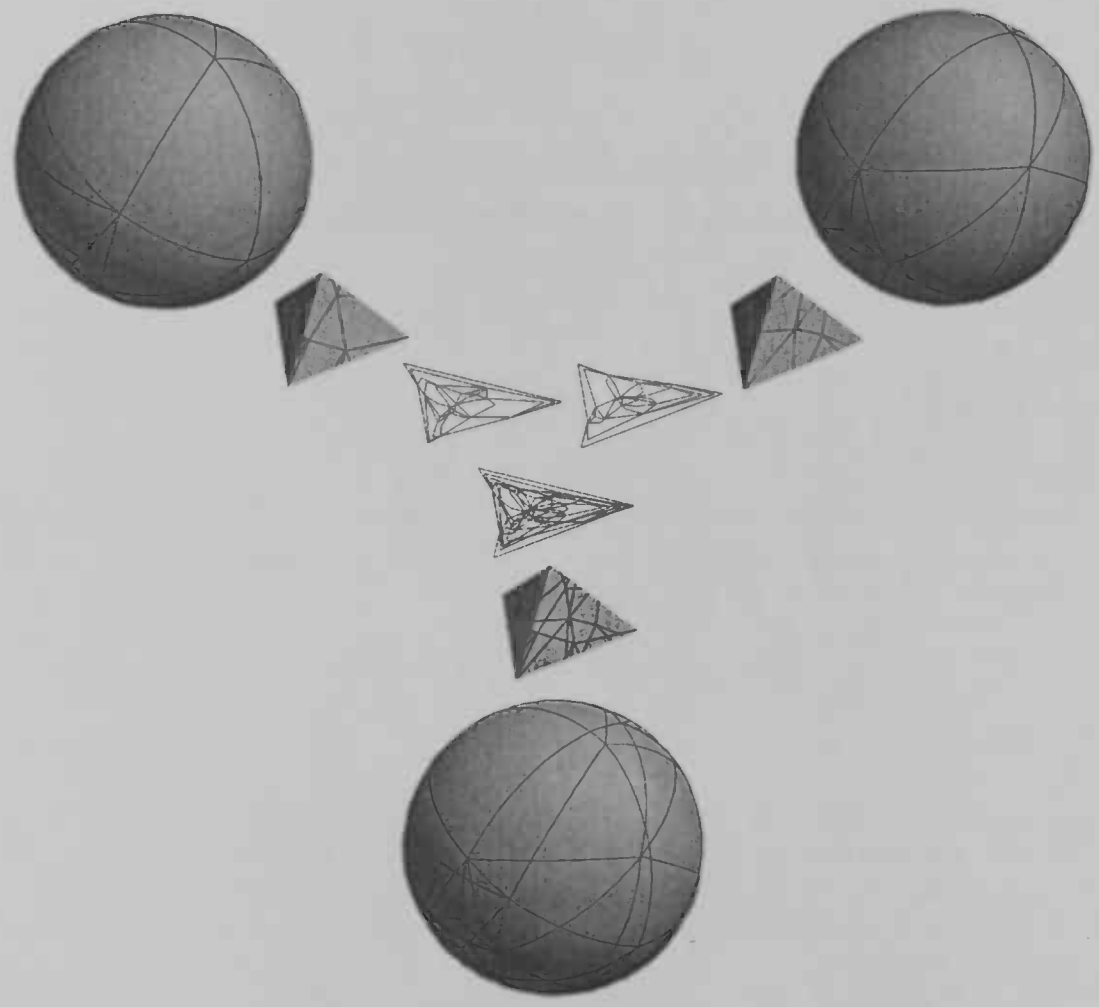
WORDT
NIET UITGELEEND

Master Thesis

Calculating the Overlay of Connected Subdivisions On a Sphere Using a Planar Overlay Algorithm



K. De Raedt
Thesis advisor: H. Bekker

Institute for Mathematics and Computing Science
University of Groningen
9700 AV Groningen
The Netherlands

WORDT
NIET UITGELEEND

Master Thesis

Calculating the Overlay of Connected Subdivisions On a Sphere Using a Planar Overlay Algorithm

27 June 2002
K. De Raedt
Thesis advisor: H. Bekker

Rijksuniversiteit Groningen
Bibliotheek Wiskunde & Informatica
Postbus 800
9700 AV Groningen
Tel. 050 - 363 40 01

One approach to determine the similarity between two convex 3D polyhedra is to calculate the Minkowski sum of the polyhedra. The Minkowski sum of two convex 3D polyhedra can be computed efficiently by an attributed graph overlay of their slope diagrams. In this paper we describe a method to calculate the graph overlay of the slope diagrams on the sphere using a planar overlay algorithm, via a mapping to a tetrahedron. Our planar overlay algorithm can be adapted to work directly on the sphere. However due to the complexity of the line intersection on the sphere the running time increases by a factor of three. We study the algorithmic complexity of the graph overlay algorithm and the mapping from the sphere to the plane theoretically and by numerical experiment.

1. Introduction

Comparing the shape of two three-dimensional convex polyhedra is an important problem in the world of computer vision. One approach is to introduce a similarity measure. The similarity measure is calculated to express the extend to which both polyhedra resemble each other.
The method to calculate the similarity measure in which we are interested is based on the Minkowski addition and inequalities for the mixed volume [1]. This method is based on the volume of both convex polyhedra and the volume of their Minkowski addition. The problem with this method however is that it is not rotational invariant. Let us take two identical convex polyhedra which have some orientation in 3D. Because they are identical the similarity measure should show this. But because the similarity measure introduced is not rotational invariant, we need to find the orientation of both polyhedra such that the similarity measure is maximal. It was shown [2, 3] that there is a set of critical orientations in which the similarity measure can be maximal. So we need to rotate the polyhedra to match an orientation of the critical set of orientations and compute is similarity measure. The maximal similarity measure of all critical orientations is the similarity measure of the polyhedra we were looking for.
A time consuming part of this algorithm is calculating the Minkowski sum of the two three-dimensional convex polyhedra A and B. A naive way to calculate the Minkowski addition works as follows. Take one vertex of A and one of B and vectorially add both coordinates, this creates a new point in a point set C. Repeat this process for all vertices of A and B. If A and B both contained $O(n)$ vertices C will now contain $O(n^2)$ points. Next we compute the convex hull of the points in C, this will give us the Minkowski sum of A and B. So the Minkowski sum of two three-dimensional convex polyhedra can be calculated in $O(n^2 \log n^2)$.
A more efficient method to calculate the Minkowsi sum of two three-dimensional convex polyhedra is introduced in [4]. The method works by calculating the slope diagrams of both polyhedra. Next the overlay of the slope diagrams is calculated. When we attach some additional information to the faces of the two original slope diagrams the Minkowski sum can be calculated from the overlay of the slope diagrams.
Calculating the overlay of both slope diagrams requires a overlay algorithm that can calculate an overlay of two graphs on a sphere. Because devising an optimized algorithm to calculate the overlay on the sphere is a difficult task we chose to compute the overlay in the two-dimensional plane. The reason for this choice is that many optimized algorithms to calculate the overlay in the plane exist so one of them can be used.

To calculate the overlay in the plane we first need to map the graph on the sphere to the plane. In cartography many mappings exist to map a structure on a sphere to the plane. All these mappings were designed to produce a two-dimensional map that could easily be interpreted by humans. However these mappings do not fulfil certain conditions needed to calculate the overlay using an existing planar overlay algorithm. In this paper we introduce a mapping that does fulfil these conditions. The new mapping we propose does not to give a good representation for humans, but a useful mapping for geometric applications.

The mapping we present maps a structure on a sphere to the plane in two steps. In the first step we map the structure on the sphere onto a tetrahedron as described in section 3.1. This mapping is just an intermediate step. Next we map the structure on the tetrahedron to the plane. We use the intermediate step because this is the only way known to us to map a structure on a sphere to the plane such that existing planar overlay algorithms can be used. Most planar overlay algorithms work on straight line segments. Here lies the main reason why we cannot use an existing cartographical mapping. These mappings do not necessarily create straight line segments. As will be shown in chapter 3 our mapping does create straight line segments.

When we started working on this project our main focus was to find a mapping that was able map a structure on sphere to the plane. We intended to use an existing overlay algorithm to compute the overlay in the plane. However when we started looking for a planar graph algorithm that could be used we could not find an algorithm that also calculated face information. When calculating the Minkowski sum of two convex polyhedra using the slope diagram method mentioned above a certain type of face information is needed. When the overlay in the plane is done and the structure in the plane is mapped back to the sphere we get a new slope diagram. This new slope diagram is the slope diagram of the Minkowski sum polyhedra. However to reconstruct the polyhedra from the slope diagram we need a extra piece of face information. For every face of the slope diagram of the Minkowski sum we need to know from which faces of polyhedra A and B it was formed.
Because we found no existing overlay algorithm that calculates this type of face information we developed it ourselves. Our first choice was to use an existing algorithm and adapt it such that the face information is calculated. This however was as much work as developing a new overlay algorithm, presented in chapter 4, that automatically yields the face information needed for the Minkowski sum.
While we were developing the overlay algorithm we saw that it could also easily be adapted to compute the overlay on the sphere without mapping the structure on the sphere to the plane first. The main change we need to make is that the intersection of two lines needs to be done on the sphere instead of in the plane. So we would not have to map the graph in the plane first but we could immediately compute the overlay on the sphere. However the intersection of two lines on the sphere is a much more time consuming operation than an intersection of two lines in the plane. Much more time consuming, as we will show in the conclusion, than the extra time needed to map the structure on the sphere to the plane first. So mapping the structure on the sphere to the plane first saves time. The complexity of using the algorithm on the sphere and in the plane is the same, however the real time needed to calculate the overlay on the sphere directly is larger.

By itself the mapping of a structure on the sphere to the plane is a useful algorithm. The conditions we imposed on the mapping are also needed for other applications. For example point location on the sphere can be done by mapping the structure on the plane and use a planar point location algorithm.

2. Data structure

To do any kind of mathematical operation on a subdivision on a sphere we need a representation for a subdivision. A subdivision will be represented by a graph. Because of the nature of the subdivision on the sphere the graph will have certain properties. A graph meeting all these properties is called a polyhedral graph.

- The graph is simple; there exists only one directed edge between two nodes of the graph.
- The graph contains no self-loops. A self-loop is defined as an edge that connects a node to the same node.
- The graph is planar, the graph can be mapped to in a two-dimensional plane so that no two edges of the graph intersect each other.
- The graph is bidirected, for every edge that connects the nodes u and v there exist a reversal edge. A reversal edge is the edge that connects the nodes v and u.
- The graph is 3-connected. When graph is said to be k-connected it means that no matter which k - 1 edges are removed the graph will remain connected.

The data structure that will be used to describe a graph throughout this paper is the standard graph data structure as defined by LEDA [5]. The structure uses three lists of records, one list for nodes, one list for edges and one list for faces. The faces of a graph can only be determined when a graph meets the conditions of a simple planar graph without self-loops. Since a polyhedral graph meets these properties we can determine the faces of a subdivision on the sphere, which is an essential part of computing the overlay of two subdivisions on the sphere.
A coordinate and a pointer to one outgoing edge e_0 defines a node record.
An edge e has a source node n and a target node m. The edge e (n, m) is called an outgoing edge of node n because it originates from n. Every edge can have a reversal edge. A reversal edge of e (n, m) is defined as the edge with source node m and target node n. So the reversal edge of the reversal edge of an edge e is e. Since the graph we use is a polyhedral graph it is also bidirected, therefore all edges must have a reversal edge. Every edge also has a pointer to the next edge with the same source node n. The next edge is defined as the edge that will be encountered first when we look in the clockwise direction around node n. All outgoing edges for a node n can be found by starting at e_0 found in the record of n, following the pointer to the next edge with the same source node n we get edge e_1. This process is repeated until the next edge e_n is equal to the starting edge e_0. All outgoing edges, $e_0..e_{n-1}$, for a node u are found sorted in clockwise order around node u. Every edge also has a pointer to the face f which lies directly to the left of it.
Only when a graph is a planar graph we can define faces. A face f is an region of the plane bounded by a collection of edges. When adding or removing nodes and edges to a graph, the graph can at some times not be planar and therefore we cannot define a face. For this reason the faces of a graph are not maintained but calculated when needed. How this is done is described in section 2.1.

2.1 Calculating the faces of a graph

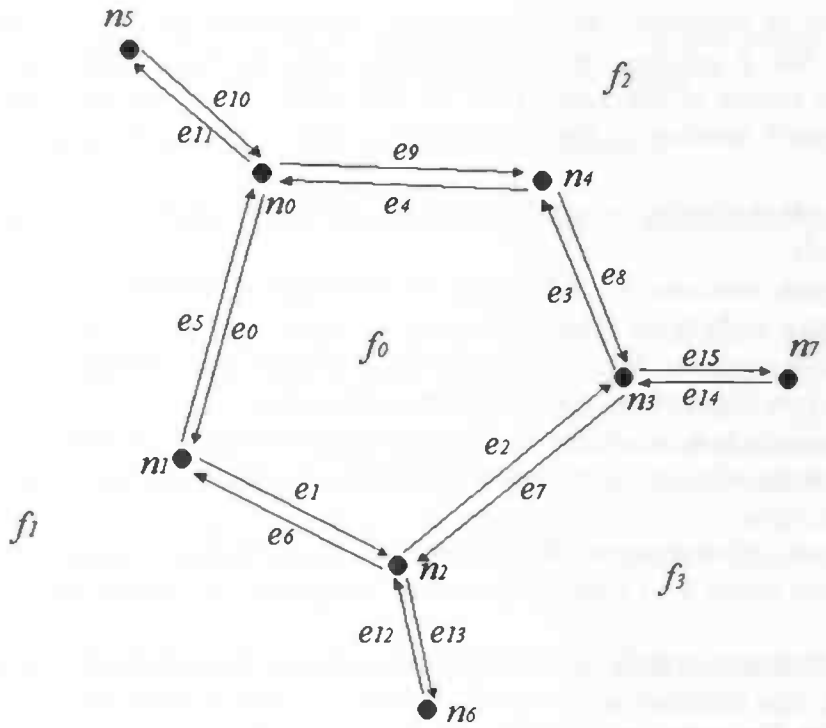


Figure 1: Example of a graph

In figure 1 an example of a graph is given, this graph has 8 nodes: n_0 to n_7, 16 edges: e_0 to e_{15} and 4 faces: f_0 to f_3. To keep figure 1 simple not all the edges bounding faces f1 to f3 are shown.
In clockwise order the outgoing edges of for instance node n_0 are edges e_0, e_{11} and e_9, because they originate from n_0. In this example we can also see that edges e_0 to e_4 bound face f_0 and that edge e_5 is the reversal edge of e_0.

Visually it is easily seen that edges e_0 to e_4 bound face f_0. Calculating this without a visual representation is more difficult. Let us look at figure 1 and suppose that the faces have not yet been calculated. Calculating them works as follows. We start at an edge of the graph say e_0. In figure 1 we can see that edge e_1 is the next edge bounding face f_0. Finding this using the data structure of the graph is done by first looking at the reversal edge of e_0, which is edge e_5. The edge e_5 is an outgoing edge of node n_1. Now we use the knowledge that all outgoing edges of a node are sorted in clockwise order. The edge after e_5 in clockwise order around n_1 is edge e_1, which is the next edge bounding face f_0. This process has to be repeated until we are back at the edge we started from. All edges that we pass in this process bound the same face as e_0.
When we are back at the edge we started from we restart the process with an edge for which we have not yet determined which face it bounds. When we know for all edges which face they bound we can also find all edges that bound a given face f.

2.2 The use of graphs

The coordinates of a node in both the two-dimensional space and three-dimensional space are expressed using exact arithmetic.
A number represented in exact arithmetic is a fraction, a rational number. A two- or three-dimensional point can be represented by 2 or 3 fractions. However because of storage and simplicity reasons the 2 or 3 fractions are brought on the same denominator. This way a point in two or three dimensions is represented as *(x, y, w)* or *(x, y, z, w)* where *w* is the denominator and x, y and z are the numerator of the coordinates. Coordinates expressed in this fashion are called homogeneous coordinates
The choice for exact arithmetic instead of floating point numbers was made because when using floating point numbers rounding errors can occur. Rounding errors could cause fatal topological changes. For example two lines that lie very near to each other could be falsely identified as intersecting due to a rounding error. These errors do not occur when all computations are done in exact arithmetic. Using exact arithmetic however will result in a slower program compared to using floating point because there has to be an extra layer of software between an operation on an exact number and the processor. Floating-point numbers do not have this extra layer and will therefore be faster, but more needs to be done to deal with rounding errors.

3. Mapping Graphs on the Sphere to the Finite Plane

Similar to a graph in the two-dimensional plane a graph on a sphere can be defined. A node in a graph on a sphere is a point on the surface of a sphere. Edges in a graph on a sphere are arcs of great circles on the sphere starting from the source node of an edge to the target node of an edge. There are always two possible arcs of a great circle bounded by two given non identical and non opposing points on the sphere. This circle consists of two great arcs beginning and ending at the two given points. Since a great circle has length 2π and both arcs cannot have length π, one of both arcs is smaller than π and the other one is larger than π. In this paper we assume all edges are smaller than π, but with some small modifications edges greater as π can also be allowed.
All the nodes and edges of a graph on a sphere lie on the surface of a sphere. Since the surface of a sphere is a two-dimensional plane faces can be seen as patches of the surface of the sphere.

Like in the two-dimensional plane the overlay of two graphs can be defined for a graph on a sphere. The intuitive solution is to create a specialized algorithm to compute the overlay of two graphs on a sphere. The other solution, which is described in this paper, is to first map the graph on the sphere to the plane. Now we can do the overlay of the two graphs in the two dimensional plane. Finally the computed overlay in the plane is mapped back to the sphere.
An obvious advantage of performing the overlay of the two graphs in the two-dimensional plane is that a lot of optimized algorithms already exist for the for the two-dimensional plane. Using one of these algorithms instead of creating a

specialized algorithm will save a lot of development time. The disadvantage of calculating the overlay in the plane is that a small overhead will be introduced when mapping a graph to and from the plane.

In order to use a fast algorithms to calculate an overlay of two graphs in a plane the mapping of the graph on the sphere has to fulfil three conditions:

1. The mapping has to be continuous and one-to-one.
2. The mapping has to be finite.
3. Great arcs on the sphere have to be mapped on straight-line segments in the plane.

The mapping described here fulfils all three conditions. With one exception condition three is extended. It states that a great arc on the sphere has to be mapped on a straight-line segment in the plane. However, with the mapping we propose a great arc on the sphere will be mapped on one, two or three straight-line segments in the plane. The mapping itself works in two steps. In the first step the graph on the sphere *SGA* (Sphere Graph A) is mapped to a graph on a tetrahedron *TGA* (Tetrahedron Graph A). In the second step the graph *TGA* is mapped to a planar graph *PGA* (Planar Graph A).

3.1 Mapping Graphs on the Sphere to a Tetrahedron

Axiom 1.
Every great arc on a sphere can be mapped to one or more connected straight-line segments on any convex polyhedron which contains the center of the sphere.

This mapping can be seen as a shadow experiment. Take a transparent sphere and draw some great arcs on its surface. Now place a light source in the center of the sphere. On the walls of the room we can see the shadows of the great arcs on the sphere. All these shadows will be straight line segments.

Our initial choice fell on a cube as the polyhedron to map the graph on the sphere on. Mapping a sphere on a cube is easy, but the overlay of the two graphs on the cube had to be done in six different planes. One overlay computation for every face of the cube. Because splitting the overlay into six patches results in a lot of overhead the final choice fell on a tetrahedron. The tetrahedron is the optimal choice since it is the simplest three-dimensional object with a nonzero volume. The tetrahedron is also the simplest polyhedron to map to the plane as will be shown in sections 3.2 and 3.3.

We want to map the graph on a sphere to the plane. Mapping it on a tetrahedron is just an intermediate step. If mapping of the graph on a tetrahedron T to the plane as described in the next two sections is to work a couple of conditions have to be met.

1. The base of T is parallel to the x, y plane. This way T has a unique top with a maximal z coordinate.
2. T is almost regular.

3. *T* contains at the origin. Because the sphere is centred at the origin the restriction made in theorem 1 that the polyhedron *T* has to contain the center of the sphere is fulfilled.

The tetrahedron *T* is constructed by starting with a regular tetrahedron *TTT* with base vertices (0,0,0)(1,0,0)(0.5,$\sqrt{3}/2$, 0) and top vertex (0.5, $\sqrt{3}/6$, $\sqrt{3}/3$). Because rational numbers are used to represent coordinates and the square root of 3 is not a rational number we have a problem. However because we displace all vertices in the next step by a random amount we do not need to use the square root of 3, a good approximation will do. Next we independently displace the three base vertices slightly in the x, y plane and we displace the top vertex slightly in 3D. This results in a tetrahedron *TT* that fulfils condition 1 and 2 but not condition 3. To fulfil this last condition the center of mass of *TT* will be placed on the origin. This can be done be translating every vertex of *TT* with the average of the coordinates of all 4 vertices. This results in tetrahedron *T* that fulfils all three conditions.
Now we can map the graph *SGA* onto *T*. We start by copying all nodes and edges of *SGA* to the resulting graph of this mapping called TGA (Tetrahedron Graph A). Next we map all the nodes of *TGA* onto *T*. Like the shadow experiment a ray (of light) is drawn from the center of the sphere through the node *n* of *SGA* we wish to map on *T*. This ray will intersect one of the faces of *T*. This intersection point is the mapping of node *n* in *TGA*. This process is repeated for every node of *SGA*.
Every node of *TGA* now lies on *T*, but this is not necessarily true for the edges of *TGA*. If the source node and target node of an edge *e* of *TGA* lie in the same face of *T* the edge *e* between the two nodes lies entirely in this face. If the source node and target node of an edge *e* of *TGA* do not lie in the same face of *T* this edge does not run over the surface of *T*. If an edge *e* source node lies on face f_i and its target node lies on face f_j then *e* will have to pass over edge *eT* of *T* connecting faces f_i and f_j. In the extreme case *e* will also run over face f_k with $i \neq k$ and $j \neq k$, in this case *e* will pass over two edges of *T*. To find where edge *e* will pass over edge *eT* we construct a ray starting from the origin and intersecting *e* and *eT*. The intersection point on edge *eT* is added as a new node *v* to *TGA* and *e* is split into two edges: (source of *e*, *v*) and (*v*, target of *e*). In the extreme case two intersection points on different edges of *T* are found and *e* is split into three parts. An edge can never be split into more than three parts. If this were the case it intersected three or more edges of *T* and therefore it would lie in all four faces of *T*. Since all edges are smaller than π this can never happen.
For mapping *TGA* to the plane it is very important that no node of *TGA* lies on the top of *T*. If a node of TGA were to lie on the top, TGA could not be mapped to the plane using the algorithm described in the next two sections. Since *T* is chosen randomly the chance of this happening is almost zero, but in case this should happen we just choose a new *T* and restart the process.

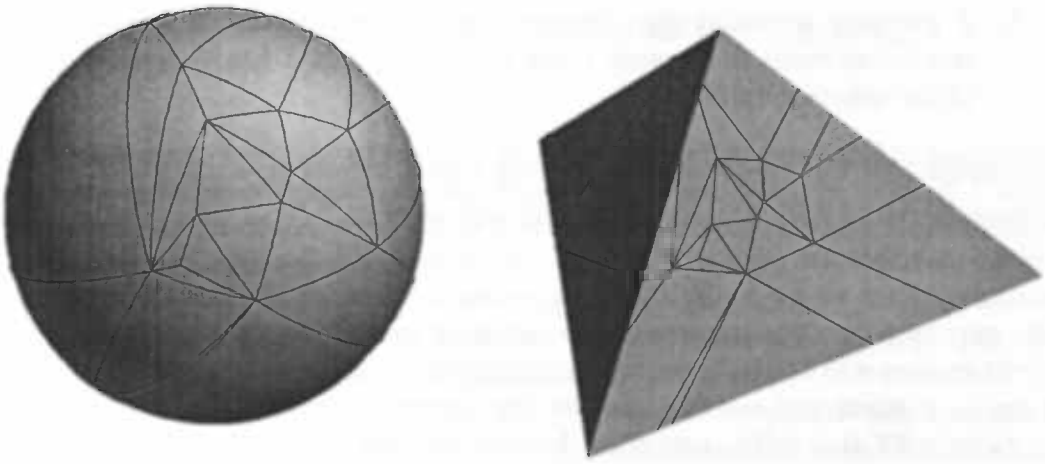


Figure 2. Sphere → Tetrahedron

3.2 Mapping Graphs on a Tetrahedron to the Plane

At this stage a graph on the sphere *SGA* has been mapped onto a tetrahedron *T*, giving *TGA*. The entire structure of *TGA* lies on the surface of *T*. Now we wish to map *TGA* onto a plane *P* that contains the base of *T* and therefore it is parallel to the x, y plane. The simplest way of mapping *TGA* to *P* is using a point projection scheme. The point projection works by choosing a projection point from which to draw a ray containing the node n to be mapped onto P. The intersection point of this ray with the plane P is the mapping of n.

The projection point *pp* has to satisfy two conditions:

1. *pp* lies inside *T*. *pp* should not lie on the boundary of *T*.
2. *pp* lies above all nodes *TGA*; the z-coordinate of *pp* is greater than the maximal z-coordinate of the nodes of *TGA*.

Based on these two conditions we can see why no node of *TGA* may lie on the top vertex of *T*. If a node of *TGA* were to lie on the top vertex of *T* and *pp* lies above this node, as it should according to condition 2, *pp* would lie outside *T* and condition 1 would therefore not be fulfilled.
If no node lies on the top vertex of *T* *pp* can be constructed as follows. Call the top vertex of *T* T_{top} and Z_{max} the z coordinate of the highest node of *TGA*. Now we set *pp* on the position ($T_{top.x}$, $T_{top.y}$, ($T_{top.z}$ + Z_{max}) / 2). The x, y and z coordinates of T_{top} are written as $T_{top.x}$, $T_{top.y}$ and $T_{top.z}$. Since *T* is almost regular the top of *T* lies directly above the base of *T*, therefore every position directly under the top and above the base of *T* will lie inside *T*. Since the top vertex and the top-most node of *TGA* lie above the base of *T* their average will also lie above the base of *T* so, condition 1 is met. Because $T_{top.z} > Z_{max}$ their average will be greater than Z_{max} thus satisfying condition 2.

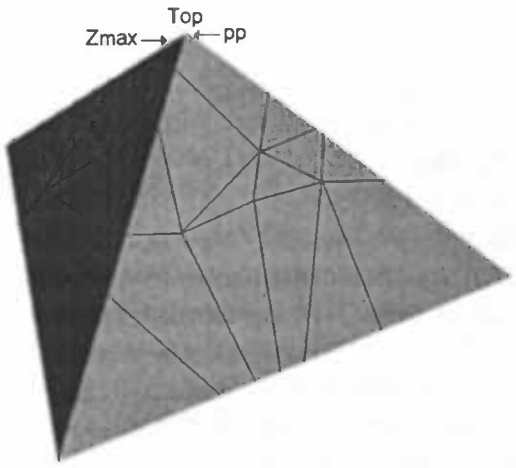


Figure 3. T with pp. pp is located between the top vertex of T and the highest node of TGA (Z_{max}).

Now we can use *pp* as a projection point to project the nodes of *TGA* on the plane *P*. The projection works as follows. All nodes of *TGA* are copied to *PGA* (Planar Graph A). If a node of *PGA* lies in the base of *T* it is already located in *P*. If a node *n* of *PGA* does not lie in the base of *T* it lies on one of the side faces. If this is the case the node needs to be projected onto *P* from the projection point *pp*. We construct a line *l* containing pp and *n* and compute the intersection point *Q* of *l* and *P*. The intersection point *Q* is the mapping of *n*.

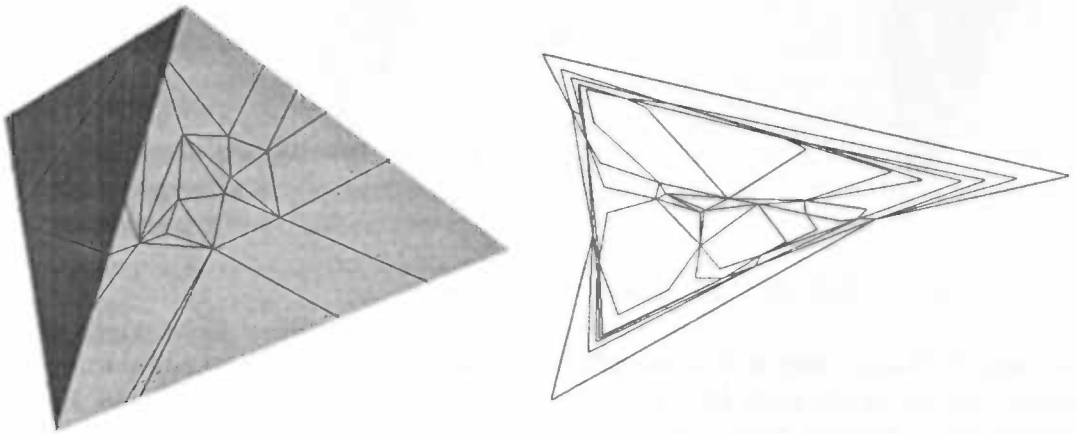


Figure 4. Tetrahedron → Plane.

This mapping is theoretically a good mapping. However as we can see figure 4, edges that were located on the sides of *T* are mapped very close together. Because we use exact arithmetic computational errors due to rounding will not occur. However this means that the fractions used to describe coordinates using exact arithmetic will be very large, which implies that a lot of computations are needed do arithmetic operations on these two fractions.

3.3 Mapping Graphs on a Tetrahedron to the Plane (2)

In this section we describe an alternative mapping to the one described in section 3.2. The problem with the previous mapping was that edges on the side faces of *T* would be mapped very close together. Because this brings a lot of problems we introduce a different and more complex mapping.
Just like the previous mapping we want to map *TGA* to the plane. We assume that the graph on the sphere *SGA* has been mapped onto a tetrahedron *T*, giving *TGA*. The plane *P* on which we want to map *TGA* contains the base of *T*.
This mapping works by folding the three sides faces of *T* down until they are parallel with *P*. This way the face of *T*, which is the base, will remain the same. The side faces have to be stretched out such that an edge that bounded two faces on *T* also bounds the two faces on *P*. Every point on the side faces of *T*, except the top vertex, has a one-to-one mapping to the stretched faces. The top vertex will be mapped onto the entire rim of the mapping and thus the mapping would not be one-to-one. This is the reason why no node should be mapped onto the top vertex of *T* as stipulated in the section 3.1.

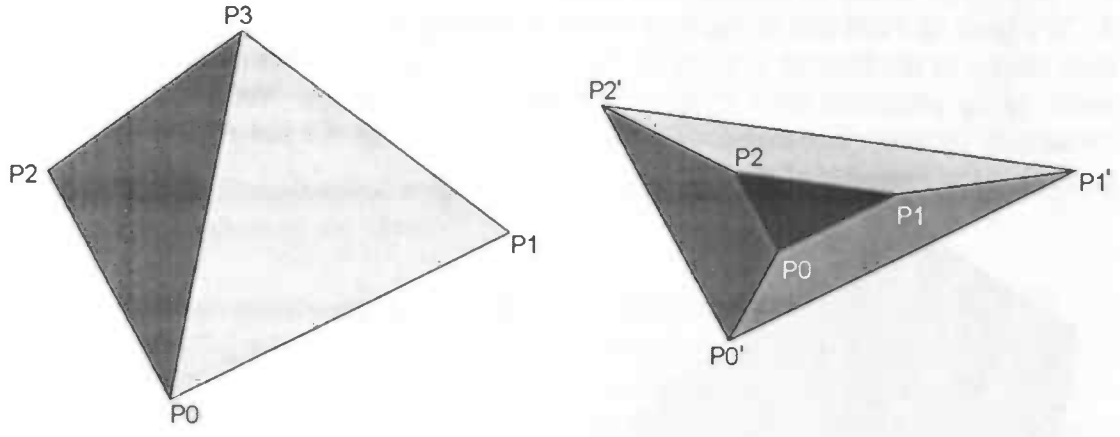


Figure 5. Left: T, right: T after the mapping

To map *TGA* according to this mapping we first copy all nodes and edges to *PGA*. All nodes that lie in the base face of *T* are already in the plane *P* and do not need any computation. For all other nodes we first determine in which face of *T* they are located. For example let us look at a node *n* inside face *(p0, p1, p3)*. After the mapping *n* will be in face *(p0, p0', p1', p1)*. This is done be construction a plane *P* parallel to the plane *(p0', p1', p3)*, containing *n*. A line *L* perpendicular to the base of *T* containing the top vertex of *T* is also constructed. The intersection point of *P* and *L* is called *Q*. *Q* is used as the projection point projecting *n* onto the face *(p0, p0', p1', p1)*. The intersection point is the mapping of node *n*.
All nodes of *PGA* are now located in *P*. Experimentally we determined that a straight-line segment of one of the faces of *T* will remain a straight line in the plane.

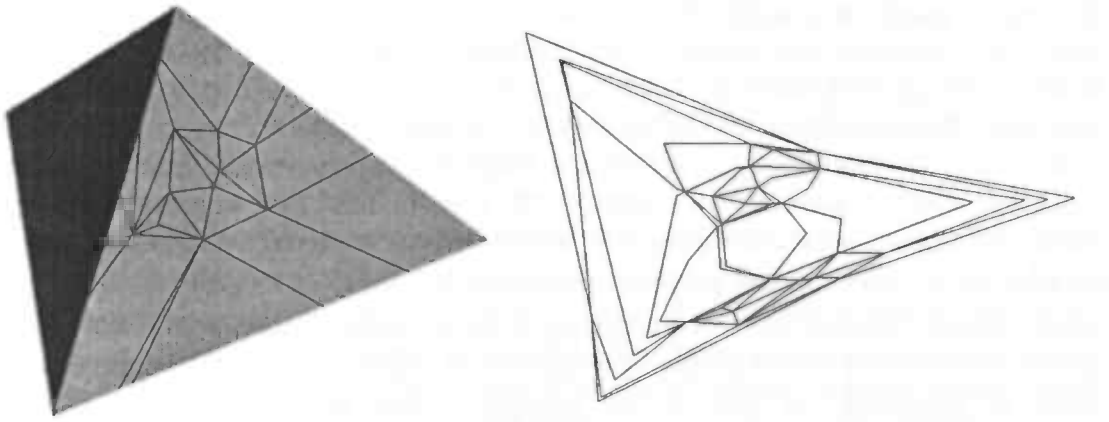


Figure 6. Tetrahedron → Plane

When comparing figures 4 and 6 it is seen clearly that the spacing between the edges in figure 6 is bigger. Therefore rounding errors and large fractions will not be a problem like they were in the mapping described in section 3.2.

3.4 Mapping Graphs in the Plane back to the Sphere

To map a graph in the plane back to the sphere we simply need to do the two steps of the mapping in reverse order. There is one small difference; when mapping the graph on the sphere to a tetrahedron we added some new nodes and edges. These added nodes and edges need to be removed when we map the intermediate graph on a tetrahedron to the sphere. This is done by remembering all the nodes added when mapping a graph on the sphere to the tetrahedron. When the back mapping is done these extra nodes are known and can be removed. Because all these nodes were introduced to split an original edge in two or three edges, the two or three edges can be rejoined as one.

3.5 Discussion

Now we can map a graph on the sphere to the plane adhering to the three conditions mentioned in the beginning. The mapping is continuous, one-to-one and finite and great arcs on the sphere are mapped on one, two or three straight-line segments in the plane.
Mapping a graph *SGA* to the planar graph *PGA* is done with the following pseudo code.

```
Do
 Choose Random Tetrahedron(T)
 Map Sphere Graph Onto Tetrahedron(SGA, T, TGA)
Until(no node of TGA on top vertex of T)
Map Tetrahedron Graph Onto Plane(TGA, PGA)
```

To map a graph on a tetrahedron to the plane we use the more difficult mapping described in section 3.3 instead of the mapping of section 3.2. The reason for this choice is simple, we need not worry about rounding errors and large fractions and both algorithms are $O(n)$ so complexity is not an issue.
Determining the complexity of mapping a graph on a sphere to a tetrahedron is more complex. When a great arc is mapped onto two or three line segments additional nodes and edges have to be added. This is done whenever an edge of *TGA* passes over an edge of *T*. The expectancy of this happening is $O(\sqrt{n})$ where n is the number of edges in *SGA*. This can be seen by looking at an embedded graph in 2D. A triangle is drawn onto the embedded graph. The number of edges inside the triangle is of the order of magnitude of area of the triangle, which is $O(n^2)$. The number of intersections of the graph with the rim of the triangle is in the order of magnitude of the perimeter of the triangle, which is $O(n)$. Therefore the expected number of intersections of the graph with the triangle is $O(\sqrt{n})$. When $n \to \infty$ the relative overhead of this mapping $\sqrt{n}/n$ will go to 0.

Number of nodes	Number of edges	Added edges	Relative overhead
100	594	138	19%
1000	6228	450	7%
10000	59748	1406	2%
100000	604984	4648	0.8%

Table 1. Experimental results of the overhead when mapping a random graph on the sphere to the plane.

In table 1 we can clearly see that the relative overhead of this mapping is $O(\sqrt{n})$. For example when the number of nodes is increased by a factor of 100, from 100 to 10000 edges the relative overhead drops from 19% to 2%, which is a decrease by almost a factor 10. This behaviour can also be observed when the number of nodes is increased from 1000 to 100000.

Mapping a graph on the sphere to the plane and back again can therefore be done in linear time and space using this algorithm.

4. A Planar Graph Overlay Algorithm

Now that the graph on the sphere is mapped to the plane we can use any graph overlay algorithm to calculate an overlay of graphs *A* and *B* in the plane. Most of these algorithms however do not take faces into account. For some applications like 3D Minkowski addition it is necessary to known from which two faces of *A* and *B* a face in the resulting graph *C* originates. The algorithm presented in this section does return this information.

Face information of a graph can only exist when a graph is planar, so when adding or removing nodes and edges to a graph face information is not available. Therefore in

LEDA this information is not maintained but calculated when needed. Before we start calculating the overlay the faces of graphs A and B need to be calculated. LEDA uses the algorithm described in section 2.1 to calculate the faces of both graphs A and B. This algorithm has running time $O(n+m)$, where n is the number of nodes and m is the number of edges in a graph.

The overlay algorithm described here works using the faces we just calculated. The concept of the algorithm works on the property that when an edge e of graph A has its source node in some face f of B we need to know if and where e leaves f. To do this we only need to see if e intersects one of the edges bounding face f. If no intersection point is found we know that e ends if face f. In the other case we know through which edge of f edge e leaves face f. Using the reversal edge information we can determine in which face of B edge e continuous in. Since we know in which face of B edge e ends we can repeat this process for all outgoing edges of the target node of e.

Because we are going to add new nodes and edges to the base-graph Y when calculating the overlay and we constantly need to know the edges bounding the original faces of Y we make a list for every face of Y of edges bounding each face before the overlay process begins.

In the resulting graph C of the overlay we can calculate the faces using the algorithm of section 2.1. But what we want to know for each face of C is from which face of A and B it was formed. This information is stored in the edges of C. Every edge of C has a pair of pointers, one pointing to a face of A and the other pointing to a face of B. This pair of pointers is set on the fly in $O(1)$ when doing the overlay of A and B. If we want to know from which two faces a face f of C originated we only need to follow the edge pointer of f to an edge bounding face f. The information in this edge will contain the pair of pointers to the faces of A and B we wanted to know.

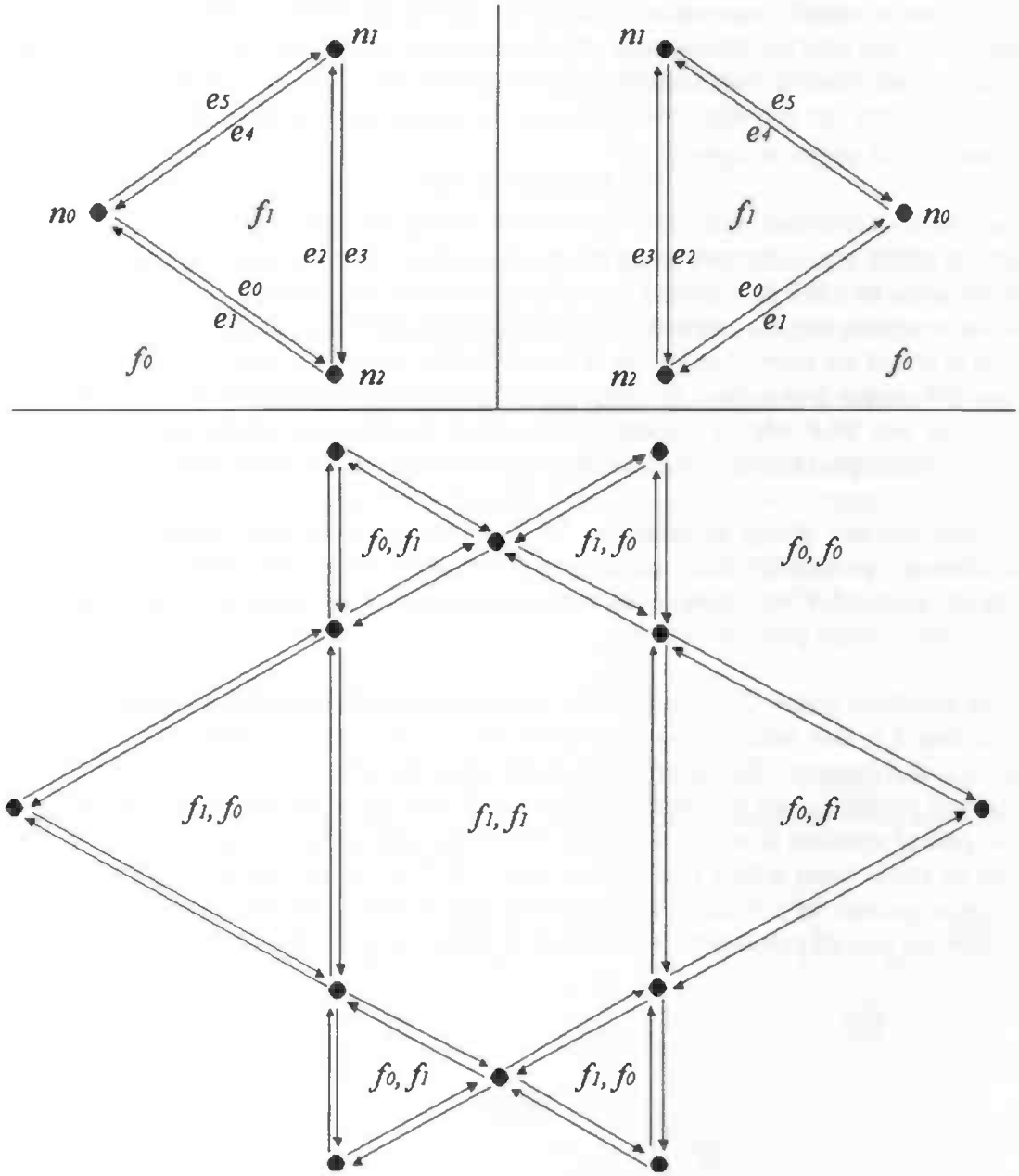


Figure 7: Upper-left: the first graph A of the overlay. Upper-right: the second graph B of the overlay. A and B are centred in the origin. Lower: The resulting graph C of the overlay. Faces are now numbered by a face from graph A, and a face from graph B. All edges in C now have two face pointers, one pointing to a face of A, and the other to the face of B from which they originated. The graph C is rescaled for visibility.

4.1 Finding a Good Starting Position

The algorithm works by incrementally adding one graph to the other one. The only choice we need to make before we start the overlay is which of both graphs will be the base-graph *Y*. The other graph is called *X* and will be added to *Y*. What we also need to have is a starting node of *X*. When we have a starting node the process

described above can be repeated until all nodes and edges X are added to Y. To make this choice we find the top-leftmost node TA and TB of graphs A and B. Based on these two points we can distinguish the cases described below. In the notation used below we write $TA.x$ meaning the x coordinate of point TA. The outer face of a graph is defined as the face that contains infinity.

1. $TA = TB$, in this case it does not matter which we choose as the base-graph.
2. $TA.x = TB.x$
    a. $TA.y > TB.y$, TA lies above TB and therefore in the outer face of B. In this case we copy B to Y and A to X.
    b. $TA.y < TB.y$, TB lies above TA and therefore in the outer face of A. In this case we copy A to Y and B to X.
3. $TA.x < TB.x$, TA lies to the left of TB and therefore in the outer face of B. In this case we copy B to Y and A to X.
4. $TA.x > TB.x$, TB lies to the left of TA and therefore in the outer face of A. In this case we copy A to Y and B to X.

In cases 2, 3 and 4 we have found a starting node of X, which lies in the outer face of Y. But in case 1 the top-leftmost node of X lies on a node of Y and we cannot determine in which face the starting node lies. In general there are two cases we need to handle differently.

I. The node of X lies in the interior of one of the faces of Y.
II. The node of X lies on the structure of Y; the structure of is either a node or an edge of Y.

During the execution of the algorithm a list L is maintained. Each item of L contains a node-face pair. The node parameter of the pair is a node of X we have added to Y and for which we still need to add all outgoing edges to Y. The face parameter is the face in which the node of X lies in Y. The face parameter is null if the node lies on the structure of Y. If a node of X lies on a node or edge, the structure of Y, we cannot say in which face of Y the node of X lies. For this reason we set the face parameter to null. As long as L is not empty the first node-face pair (n, f) is taken from L and all outgoing edges of n are added to Y. The target nodes of the edges we just added are appended to the list L if we did not already visit them. Because the graphs handled in this algorithm are connected we add every node and edge of X to Y if we start with just one node.
What the first element is we place on the L depends on which case we have. In case 1 the pair (the top-leftmost node of X, null) is added to L. In the other case we can add the pair (the top-leftmost node of X, the outer face of Y) to L. In cases 2, 3 and 4 we also need to add a copy of the top-leftmost node of X to Y because this node does not yet exist in Y.

4.2 Adding Edges to the Base-Graph

When we take the top node-face pair (n, f) from L we want to add the outgoing edges to Y. There are a lot of different possibilities how an outgoing edge of n should be added to Y; all these possibilities have to be handled in differently. If we add the edge

(n, v) to Y we can add its reversal edge to Y at the same time. Because they are the same line segment they have the same intersection points with the edges of *Y*. So there is no need to calculate this intersection point twice. Thus if we add an edge e_x to *Y* we also add the reversal edge of e_x to *Y*.

4.2.1 The node of *X* lies on the structure of *Y*

If the face parameter is not null we know in which face *f* of *Y* node *n* of *X* lies in. Thus we know that all outgoing edges of *n* start in face *f*. However if *f* is null *n* lies on the structure of *Y* and we need to separately determine for every outgoing edge e_x of *n* in which face of Y it runs.
Determining in which face of *Y* e_x runs in can be done by comparing angles. We know all faces of *Y* that lie around node *n*, two in case *n* lies on an edge of *Y* and more than two if *n* lies on a node of *Y*. Now all we need to do is check if the angle of e_x is between the two angles of the edges bounding a face *f* of *Y*. This process needs to be done for all outgoing edges of *n* separately. If the face is found in which e_x runs we can treat this case as if *n* lies in a face of *Y* see section *(4.2.2)*. If however e_x runs does not lie between two edges of *Y* then e_x runs over an edge of *Y*. This case needs to be handled as follows.

4.2.1.1 Edges of X and Y overlap

Let us see what needs to be done when e_x and an edge e_y of *Y* overlap. If we have established that the edges e_x and e_y overlap, we need to distinguish three cases: e_x is shorter, equal in size or longer than e_y.

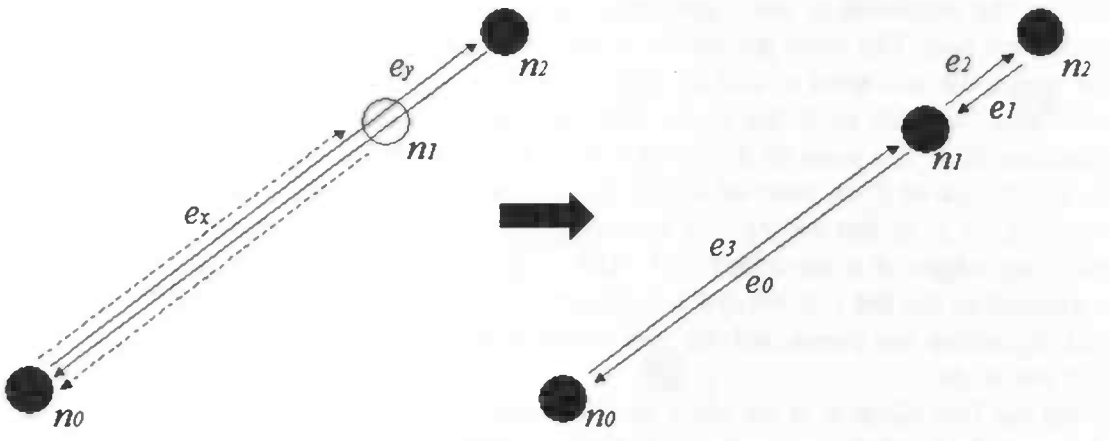


Figure 8. Left: e_x and its reversal edge displayed as the dotted edges from n_0 to n_1 is shorter than e_y displayed as the full edge from n_0 to n_2. Right: after e_x is inserted into Y, n_1 is now solid because it belongs to Y.

e_x is shorter than e_y: the target node of e_x is added to Y and e_y is split in two edges (e_3 ,e_2). Because the reversal edge of e_y is the same line segment it will also overlap the reversal edge e_x so we can handle this edge at the same time and thus we also split the reversal edge of e_y into two edges (e_1, e_0). The lists of edges bounding the faces to the left and right of e_x also need to be updated since they contain e_y or the reversal of e_y. e_y (or the reversal edge of e_y) is replaced in both the faces by the two edges we added to

split e_y. If we did not already visit the target node of e_x we need to add it to L of node-face pairs. Because the target node of e_x lies on the structure of Y the face parameter is set to null. Next we need to update the pair of pointers to faces from which the new edges originated. This can only be done for the edges (e_3,e_0) since they are both edges existing in X and Y, we can set both pointers to the faces of X and Y which are stored in e_x and e_y.

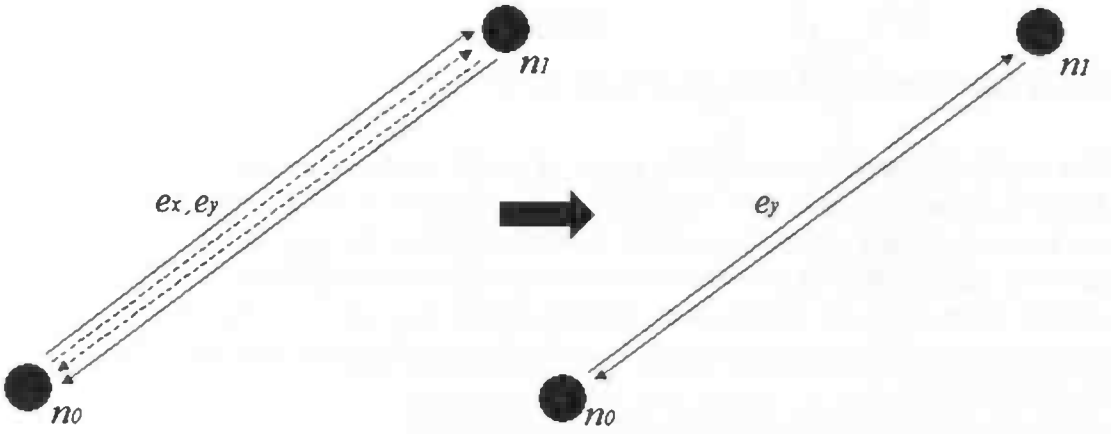


Figure 9. Left: e_x and its reversal edge displayed as the dotted edges from n_0 to n_1 is equal in size to e_y displayed as the full edge from n_0 to n_1. Right: after e_x is inserted into Y, e_x is equal to e_y so we can keep e_y.

e_x is equal in size to e_y: Because e_y exactly contains e_x there is no need to split e_y into two edges as in the previous case. Again we need to update the pair of pointers pointing to the faces an edge in Y originated from, but now it can be done for the entire edge e_y. The target node of e_x already exist in Y so we do not need to add it. Next we add (the target node of e_x, null) to L if we did not already visit the target node of e_x.

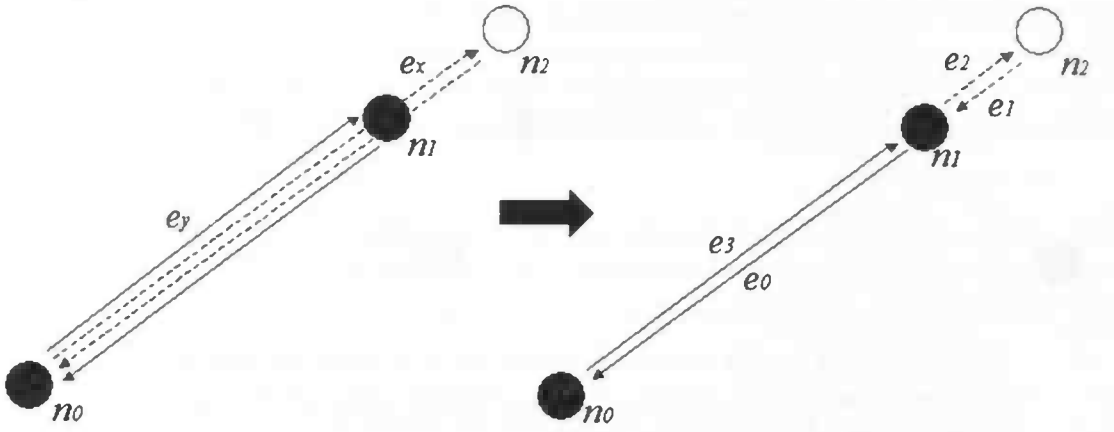


Figure 10. Left: e_x and its reversal edge displayed as the dotted edges from n_0 to n_2 are longer than e_y displayed as the full edge from n_0 to n_1. Right: e_x is split into two parts. The first part (e_0,e_3) is inserted into Y, The second part (e_1,e_2) will be handled later.

e_x is longer than e_y: Because e_x entirely contains e_y we do not have to split e_y into two edges. For the part that e_x overlaps e_y, edges (e_0, e_3), we can set the pair pointers to the faces of X and Y it bounds. But for the remaining part edges (e_1, e_2) we have no

information of the face of Y they bound. This is solved by splitting e_x into two parts by adding the target node of e_y to X and thus splitting e_x into two edges. The first part is exactly e_y and the second part is the remaining piece. The first part can be handled as if it was the previous case. To handle the second part we add (the new edge, null) pair to L, the second part is an outgoing edge of the new node and will now be handled as if it were a normal edge.

4.2.2 The node of *X* lies in a face of *Y*

If n lies in a face of Y all outgoing edges of n will start in this face, what needs to be found is where they end. For every face f of Y we have a list of edges that bound f. If we know an edge e starts in face f we find if or where this edge leaves face f. This is done by determining all intersections of e with the bounding edges of f. When f is not convex more than one intersection will be found, but we want the first. That is the point where we first leave this face, we will re-enter this face later on but this cannot be handled yet.
Again we need to look at a couple of cases separately.

4.2.2.1 An edge e_x of X ends in a face of Y

This is the easiest case, e_x does not intersect any edge of Y. All we need to do is add the target node of e_x to Y and add e_x and its reversal edge to Y. We can also set the pair of pointers to the original faces. The edges e_x and its reversal contain the information which face of X they bound and because the entire edge lies in face f of Y the pointer to the face of Y of both edges is f. If we did not already visit the target node of e_x we can add the pair (target node of e_x, f) to L.

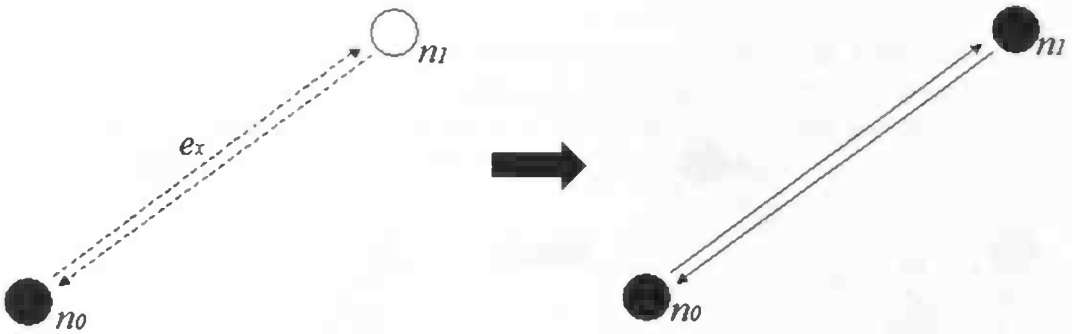


Figure 11. Left: e_x the dotted edge from n_0 to n_1 does not intersect any edge Y. The target node of n_1 and e_x and its reversal edge can be added to Y, which results in the situation on the right.

4.2.2.2 An edge e_x of X leaves a face of Y through e_y

In this case e_x and e_y will intersect in some point P as shown in figure 12.

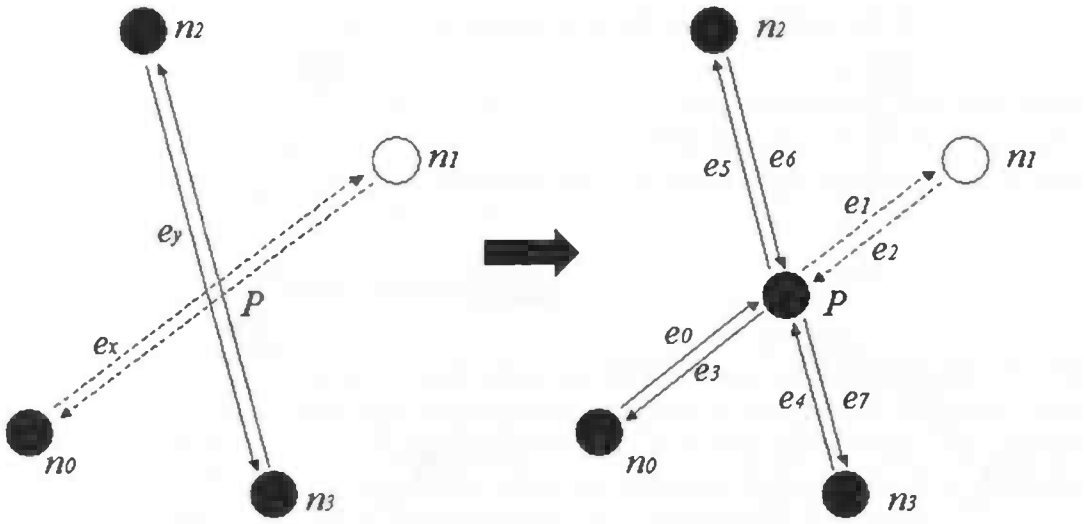


Figure 12. Left: e_x and e_y intersect at P. Right: e_x and e_y and there reversal edges are split into two pieces around a new node P. e_1 and e_2 can not yet be added to Y since they could intersect another edge of Y.

What needs to be done first is to insert a node to *X* and *Y* at point *P*. Next we split the edges e_x and e_y and their reversal edges in two pieces around *P*. Of the first part of e_x, the edges e_0 and e_3 we know for sure that they have no other intersection point with the structure of *Y* and we can therefore add them both to *Y*.
Because e_y and it reversal edge were part of face f_0 and f_1 they have to be replaced in the lists of edges that bound faces f_0 and f_1 by the two new edges that split e_y and its reversal edge into two pieces.
Now we can update the pair of pointers to faces from which they originated. This can be done for six new edges of *Y* (e_0,e_3 ,e_4 ,e_5 , e_6 ,e_7). It is also possible to set the pointers for the edges and (e_2 ,e_3) but this will always be done at a later stage so we need not worry about that now. The edges e_0 and e_5 both bound the face created by the face of e_y and the face of e_x. Similarly the edges e_3 and e_4 bound the face created by the of e_y and reversal edge of e_x. The edges e_6 and e_7 both lie in the face of the reversal edge of e_y, and e_6 also lies in the face of e_x whereas e_7 lies in the face of the reversal edge of e_x.

Because at this stage it is unknown what the edge e_1 and its reversal edge e_2 will intersect we can only add its source node, which is the new node in *X* at *P*, to *L*. The face parameter can also be set to the face of the reversal edge of e_y since we know that e_1 runs in this face.

4.2.2.3 An edge e_x of *X* ends on edge e_y of a face of *Y*

The difference between this case and the previous is that e_x of X ends on edge e_y of a face of *Y* in point *P*. Since e_x ends on an edge we do not need to split it into two pieces. A new node of *Y* at *P* needs to be added and e_y and its reversal edge have to be split in two edges around *P*. e_x and it reversal edge can be added to *Y* because they have no other intersection with the structure of *Y*.
Similar to the previous case we need to update the lists of edges that bound faces containing e_y and its reversal edge.

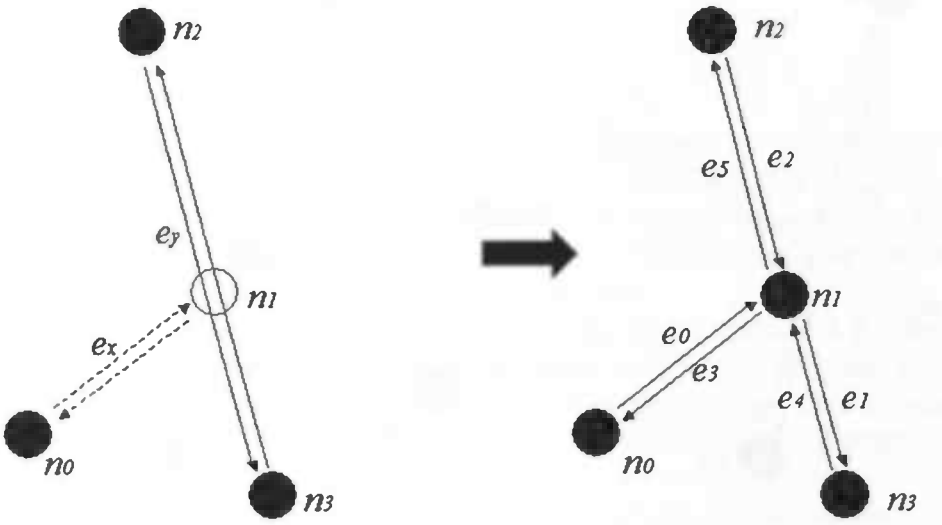


Figure 13. Left: the target node n1 of edge ex ends on the edge ex. Right: the node n1 is added to Y and the edge ey is split into two pieces.

The pair of pointers to faces from which the four new edges (*e0* ,*e3* ,*e4* ,*e5*) of *Y* originated can now be set. The edges *e0* and *e5* both bound the face created by the face of *ey* and the face of *ex*. Similarly the edges *e3* and *e4* bound the face made by the of *ey* and reversal edge of *ex*. For the other two edges we cannot determine which face of *X* they bound.
Finally we have to add the target node of *ex* to *L* if it was not visited yet. Because the target of *ex* lies on the structure of *Y* the face parameter has to be set to null.

4.2.2.4 An edge e_x of X ends on node n_y of a face of Y

If *ex* ends on a node of *Y* we do not need to add a new node to *Y* or split into two pieces. All that needs to be done is add *ex* and its reversal edge to *Y* with the correct pair of pointers. Because no edge of *Y* is split into two pieces we do not need to update the list of edges that bound the face of *Y* that *ex* runs in.

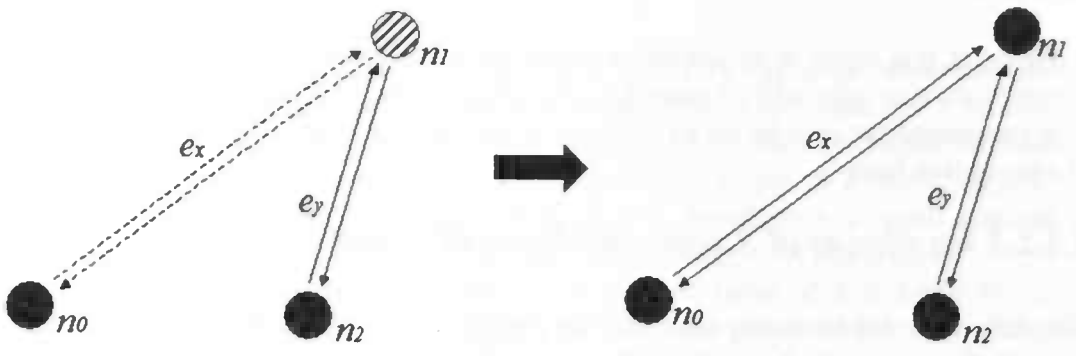


Figure 14. Left: ex and ey both end on node n1. Right: since n1 already exist we only need to add ex and its reversal edge to Y.

Again we have to add the target node of *ex* to *L* if it was not visited yet. Because the target of *ex* lies on the structure of *Y* the face parameter has to be set to null.

4.2.2.5 An edge e_x of X runs through a node n_y of a face of Y

Because e_x does not end in this face we have to split it into two pieces. The first piece is entirely contained in the face. We do not yet care what happens in the second piece, this will be handled later. The first piece can be seen as an edge that ended on n_y and it can be handled according to section 4.2.2.4.

4.3 Setting Final Face Information

When L is empty all edges of X have been handled by one of the cases above. All edges X are now added to Y and have the correct pair of pointers to the faces they bound. However edges that have no intersection with an edge of X were not touched by adding X to Y. This means that these edges do not have the correct pair of pointers set. They only have information of the face of Y they bound, see figure 15.
The missing information can be set as follows. If an edge e_y of Y does not intersect an edge of X we know that in is entirely enclosed in a face of X. To find which face of X this is we look at its neighbouring edges. If one of these edges of Y has an intersection with an edge of X we know which face e_y is located in. If none of these neighbouring edges have an intersection point with an edge of X we look at their neighbouring edges and so on.

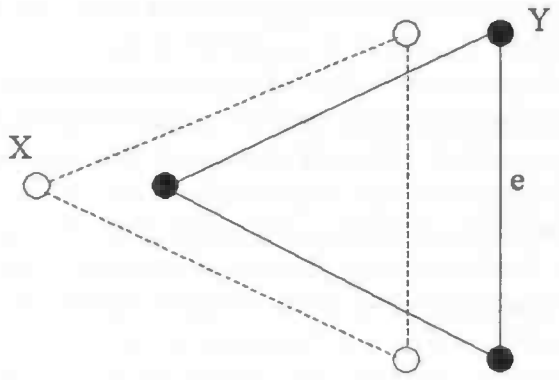


Figure 15: Example of an incomplete overlay. For simplicity reasons an edge and its reversal edge are represented by one line. When calculating the overlay of X and Y the edge e will have no intersection with an edge of X. And therefore it will miss the face information.

There is one last case that needs to be handled here. If graphs X and Y have no intersection points the algorithm described above to fill in the missing information will not work. In this case the entire graph Y lies in one of the faces of X. The only solution for this case is to use a point location algorithm. We just compute in which face of X one of the nodes of Y lies and since Y lies entirely in this face of X we can set all missing information. When calculating the Minkowski sum of two polyhedra as described in the introduction, we calculate the overlay of the slope diagrams of the two polyhedra. Because of the structure of the slope diagrams there will always be at least one intersection point of both slope diagrams.

4.4 Complexity

To determine the complexity of the presented graph overlay algorithm we will count the number of times we test if two edges intersect. The work that needs to be done for an intersection depends on which type of intersection we have. But looking back at section 4.2 were we handled all possible types of intersections we can see that all types of intersections can be handled in constant time.
Special attention needs to be given to the case handled in section 4.2.1. In this case a node *nx* of *X* and a node *ny* of *Y* have the same coordinates. Because *nx* lies on the structure of *Y* we needed to determine for every outgoing edge of *nx* in which face of *Y* it runs. This was done by comparing the angles of the outgoing edges of *nx* to the outgoing edges of *ny*. Because we have stored the outgoing edges of all nodes in counter-clockwise order the angles of the edges are already sorted and we therefore need to do a constant amount of work for each outgoing edge of *nx*. Since all edges are only looked at once we can also handle this special intersection point in constant time.
When an edge *ex* of *X* lies in a face of *Y* we need to test whether *ex* intersects with any edge if *Y* bounding face *f*. On average the number of edges bounding a face is the number of edges divided by the number of faces. Sometimes we need to split an edge of *X* into two pieces, where the second piece itself could again be split into two pieces. Each of these pieces is contained in a different face of *Y*. So for some edges of *X* we need to test if it intersects the edges of more than one face. For this reason the complexity of the algorithm dependents on the number of intersections. In the case of an intersection an edge of *X* is split into two parts. For each part we will have to determine if or where it leaves the face of *Y* it is in.
Based on the number of edges of *X (Nx)* and the number of edges and of *Y (Ny)* and the number of faces of *Y* (*Fy)* we can calculate the expected number of intersections we need to check for. The complexity of the overlay algorithm can be approximated using the following formula.

$$(Nx + k)\frac{Ny}{Fy} \qquad (1)$$

The average number of edges bounding a face of *Y (Ny/Fy)* is an important part of formula 1. This is based on the fact that if it is known in which face *f* of *Y* an edge of *X* lies we need to determine through which edge of *f* it leaves. To do this we need to check for an intersection between the edge of *X* and all edges of *Y* bounding *f*. If the average number of edges bounding a face of *Y* is small only a small number of intersection tests will be needed to determine were an edge of *X* lies a face.
The worst case scenario happens when the number of faces of *Y* is small while the number of edges is large. For example let us look at a graph of a regular polygon. This graph will have *n* edges but only two faces. This would mean that we test whether every edge of *X* intersects every edge of *Y*. and therefore we can see in the formula that the worst case complexity of this algorithm is $O(n^2)$.
In order for this algorithm to have a low complexity the average number of edges bounding a face needs to be small. The best case for this algorithm is when the base graph is a triangulation with a triangle as convex hull. In this way all faces will be

bound by three edges. Looking at the complexity formula we can see that in the best case scenario this algorithm is $O(n)$.
The case in which this algorithm has a low complexity is when the number of edges and faces of Y is linear in the number of nodes of Y. In this way the average number of edges bounding a face will be a constant as we can see in formula 1
But what we want to know is the complexity when we calculate the overlay of two random planar graphs.

Graph X		**Graph Y**		**Resulting Graph**		**Time (seconds)**
Nodes	*Edges*	*Nodes*	*Edges*	*Nodes*	*Edges*	
100	422	100	410	525	2132	0.17
500	2120	500	2116	2699	11032	0.80
1000	4288	1000	4344	5539	22788	1.59
5000	21586	5000	21594	26980	111100	8.10
10000	43162	10000	42888	53778	221162	16.2

Table 2. Experimental results of calculating the overlay of two random planar graphs X and Y.

From table 2 we can derive that the experimental complexity of the overlay algorithm is $O(n)$ for random planar graphs. For instance if the number of nodes is increased by a factor of 2 from 500 to 1000 the time needed to calculate the overlay also increases by a factor of 2.
Note that all graphs are created randomly in the two-dimensional plane and are not a mapping of graph on a sphere. So in all the examples the coordinates of the nodes of the graphs X and Y are small fractions. By small fraction we mean that both the numerator and denominator are small integers represented by 32 bits. When mapping a graph on the sphere to the plane the coordinates will not necessarily be small fractions. So the time needed to compute the overlay of two graphs which are mappings of graphs on a sphere will take more time then presented in table 2. Because the coordinates will have to be represented in a lot more bits the time needed to do one arithmetic operation will be greater and therefore the algorithm will take more time. However this does not mean that the complexity of the algorithm changes. The complexity of the algorithm remains the same, only the time needed will increase as a result of the increased time needed to do one arithmetic operation.

4.5 Overlay examples

The overlay algorithm presented in this chapter can be used separately from the mapping of the graph on the sphere to the plane. It can be used as any other overlay algorithm to calculate the overlay of two planar graphs.

The first example we want to show is the overlay as demonstrated in figure 7 of two triangles X and Y. X pointing to the left and Y pointing to the right as shown in figure 16A. The inner face of both triangles is called face 0 and the outer face is called face 1. In figures 16B to 16E we can see the area that was constructed from face x of X and face y of Y. The area is also equal to the union of the areas of faces x and y.

In the example of figure 16 the unions of all permutations of the faces x and y have a nonzero area, but this is not necessarily true as demonstrated in figure 17. In figure 17 we calculate the overlay of a square with one diagonal *X* and a triangle *Y*. The triangle has only two faces: the outer face 0 and the inner face 1. The square has three faces: the outer face 2, the lower face 1 and the upper face 0. As can be seen easily in figure 17A the lower face 1 of the square and the outer face 0 of the triangle have no common area. So there will not be a face in the overlay that was constructed of faces 1 of the square and 0 of the triangle. This can be seen in figure 17D, no face is constructed from faces 1 and 0.

Using the algorithm presented in this paper we have an extra piece of information that lacks with other planar overlay algorithms. The information contained in the faces of the resulting overlay from which two faces they were constructed is very valuable information. If for instance we want to know which faces of a graph *X* have a nonzero union with some face *y* of a graph *Y*. If we only need to know this kind for more than one face of *Y* we only need to calculate the overlay once. After we calculated the overlay of graphs *X* and *Y* we can determine in *O(1)* for every face of *X* (or *Y*) with which faces of *Y* (or *X*) it has a nonzero union.
An example of a real life application were this information would be useful is in cartography. Let us assume we have a map of the counties in the Netherlands. We also have a map of the level of pollution of the Netherlands. After calculating the overlay of both maps once we can calculate a lot of statistics we want to know. For example the counties that have an area of high pollution, or the average pollution of an entire county. All these statistics can be determined in *O(1)*.

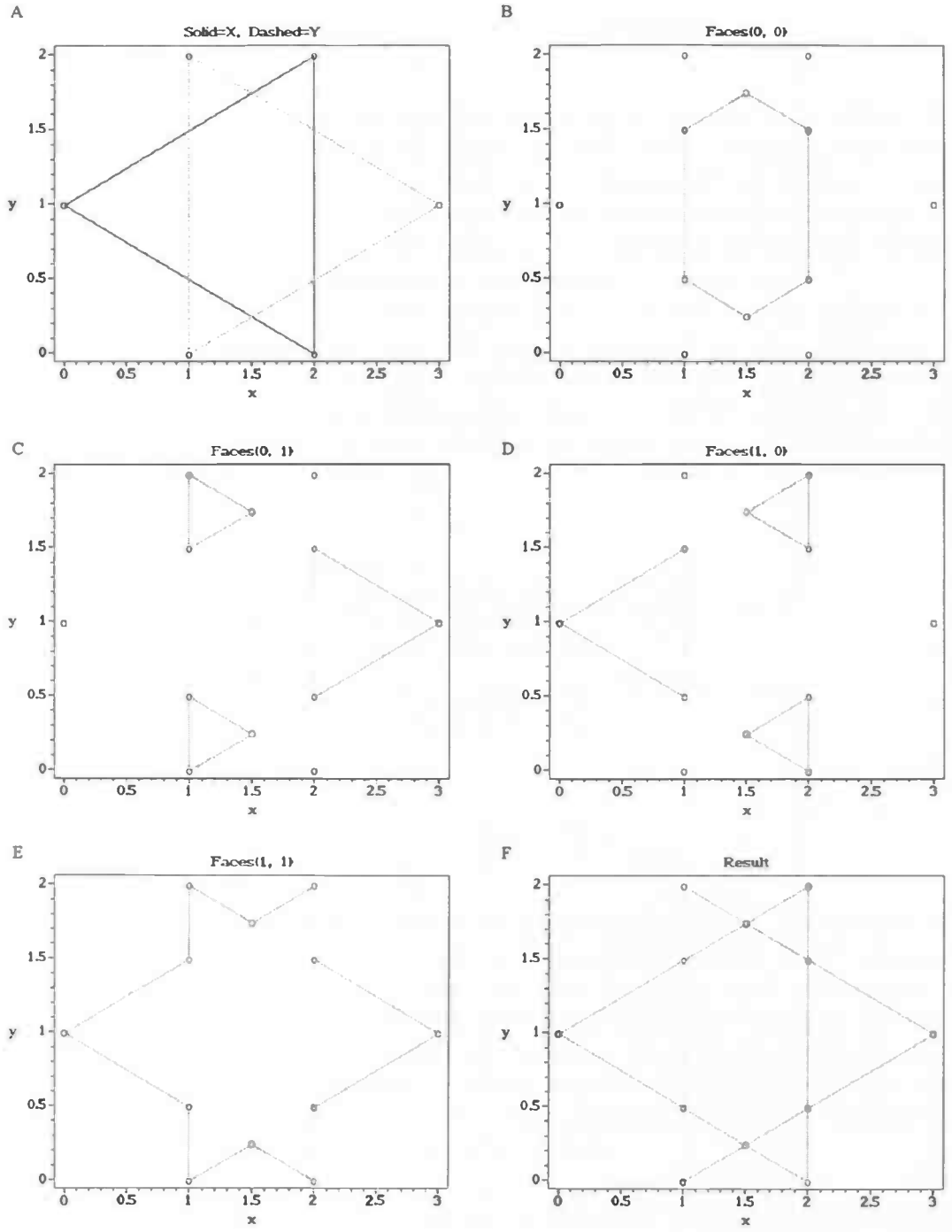


Figure 16. Example of an overlay of two triangles X and Y.
A: the two triangles before the overlay.
B: The edges that bound the face constructed from face 0 of X and face 0 of Y.
C: The edges that bound the face constructed from face 0 of X and face 1 of Y.
D: The edges that bound the face constructed from face 1 of X and face 0 of Y.
E: The edges that bound the face constructed from face 1 of X and face 1 of Y.
F: The result of the overlay of triangles X and Y.

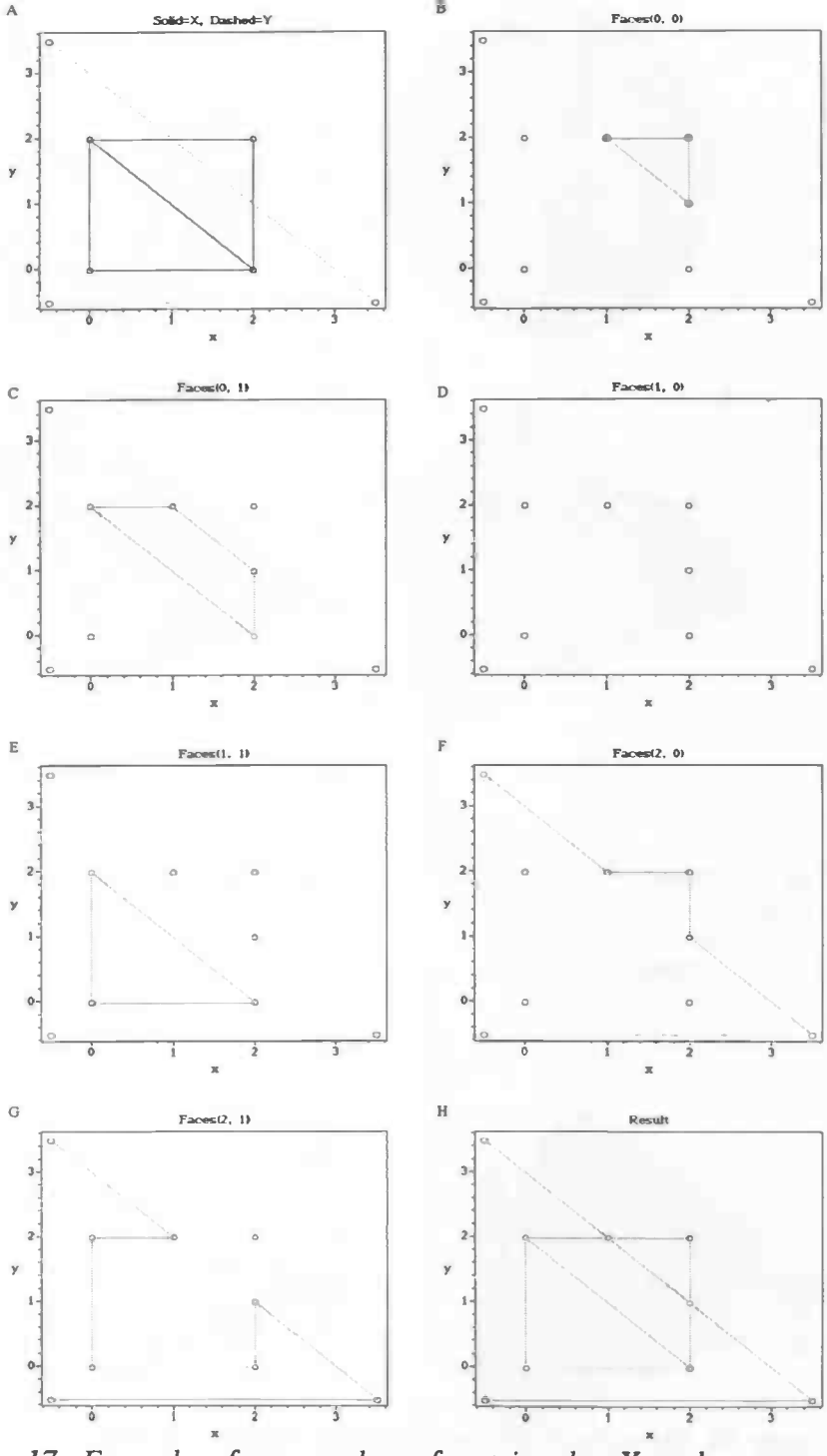


Figure 17. Example of an overlay of a triangles X and a square with a diagonal Y.

A: the two shapes before the overlay.

B: The edges that bound the face constructed from face 0 of X and face 0 of Y.

C: The edges that bound the face constructed from face 0 of X and face 1 of Y.

D: The edges that bound the face constructed from face 1 of X and face 0 of Y.

E: The edges that bound the face constructed from face 1 of X and face 1 of Y.

F: The edges that bound the face constructed from face 2 of X and face 0 of Y.

G: The edges that bound the face constructed from face 2 of X and face 1 of Y.

H: The result of the overlay of shapes X and Y.

5 Results

Now we have all the tools to calculate the overlay of two graphs on the sphere. So we can put them all together. Using the algorithm described in section 3.1 we can map the two graphs to a tetrahedron. Next we use the algorithm from section 3.3 to map both graphs to the plane. When both graphs are mapped to the plane we use the planar graph overlay algorithm presented in chapter 4 to calculate the overlay. Finally we map the computed overlay graph back to the sphere.
In pseudo code the overlay of sphere graphs *SGA* and *SGB* is written as follows. The graphs on the tetrahedron *TGA* and *TGB* and the graphs in the plane *PGA* and *PGB* are only intermediate graphs needed to calculate the overlay graph in the plane *PGAB*. *PGAB* and *TGAB* are also only intermediate graphs needed to calculate the overlaid graph on the sphere *SGAB*. So when *SGAB* is calculated all graphs on the tetrahedron and in the plane can be forgotten.

```
Do
  Choose Random Tetrahedron(T)
  Map Sphere Graph Onto Tetrahedron(SGA, T, TGA)
  Map Sphere Graph Onto Tetrahedron(SGB, T, TGB)
Until(no node of TGA or TGB on top vertex of T)

Map Tetrahedron Graph Onto Plane(TGA, PGA)
Map Tetrahedron Graph Onto Plane(TGB, PGB)

Compute Overlay(PGA, PGB, PGAB)

Map Plane Graph Onto Tetrahedron(PGAB, TGAB)
Map Tetrahedron Graph Onto Sphere(TGAB, T, SGAB)
```

Based on the algorithm above we did some experiments to verify the complex of the algorithm. In all these experiments we created two random graphs on the sphere given a number of nodes. Next we measured the time it took to map both graphs on a tetrahedron and finally to the plane. We also measured the time it took to calculate the overlay and the time it took to map the resulting graph back to the sphere. The mapping to and form the sphere are added together and displayed in the middle column of table 3. This time is the extra time needed to calculate the overlay in the plane instead of calculating the overlay on the sphere.

Number of nodes of each of the graphs on the sphere	Time needed to map both graphs to the plane and back (seconds)	Time needed to calculate the overlay (seconds)
100	16.4	173
200	27.7	371
500	56.7	894
1000	104.5	1901

Table 3. Average results of experiments of mapping two random graphs on the sphere to the plane. Calculating the overlay in the plane and mapping the resulting graph back to the sphere.

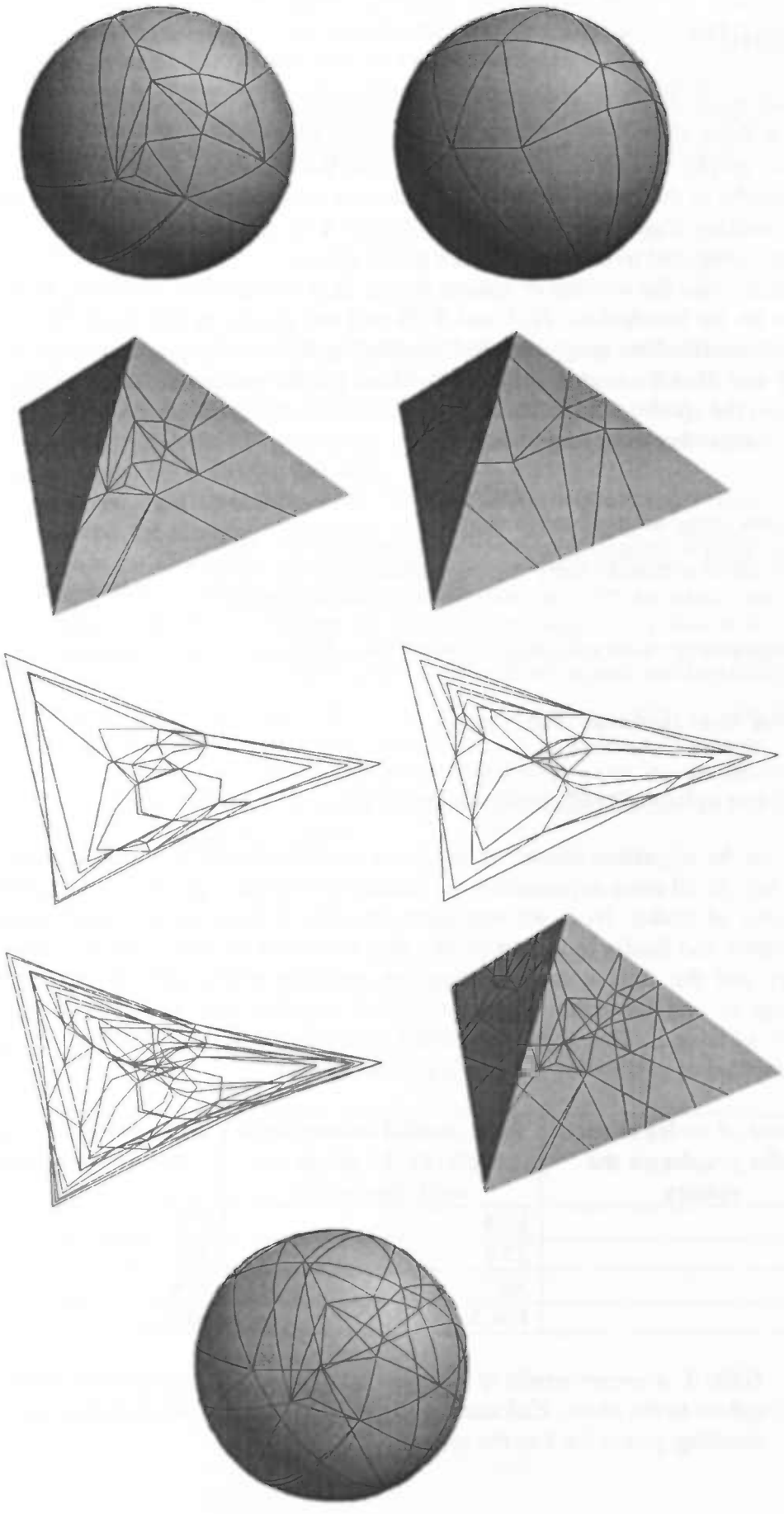


Figure 18 on page 30. The entire process of overlaying two graphs on the sphere SGA and SGB (top-left and top-right). Both graphs are mapped onto a tetrahedron. The resulting graphs TGA and TGB are presented in the second row. Next TGA and TGB are mapped onto the plane giving PGA and PGB as shown in the third row. In the fourth row on the left is the overlay of PGA and PGB. On the fourth row on the right the resulting graph is mapped back to the tetrahedron. Finally the overlay of SGA and SGB is mapped back from the tetrahedron to the sphere.

6 Conclusion

We have developed an algorithm to calculate the overlay of two connected subdivision of a sphere. This algorithm has three parts. In the first part we map the structure on the sphere to the plane. So we can calculate the overlay of the subdivisions on the sphere in the plane. After the overlay is computed in the plane we do an inverse mapping and map the structure on the plane back to the sphere.

Mapping a structure on a sphere to the plane is done in two steps. First the structure on the sphere is mapped to a tetrahedron. This step is merely a intermediate step necessary to map the structure on the sphere to one finite plane. In the second step we map the structure on the tetrahedron to the plane. This mapping can be used independently from the problem of computing an overlay on the sphere. For example if we wish to do point location on the sphere we could map the structure on the sphere to the plane, and do the point location in the plane. Thus removing the extra development time needed to create an optimized algorithm to do point location on the sphere.

The overlay algorithm described in this paper was developed to mainly to calculate additional face information needed to calculate the Minkowski sum [1]. Every other planar overlay algorithm could be use to calculate the overlay in the plane. One could also use the presented planar overlay algorithm stand alone. The additional face information that is calculated by this algorithm is also useful for other applications like cartography. An example of how the face information is useful in cartography is described in section 4.5.

Mapping a structure on the sphere to the plane has a time complexity of $O(n)$, where n is the number of nodes of the structure. The planar overlay algorithm has a best case time complexity of $O(n)$. In table 3 we presented results of calculating the overlay of two random structures on the sphere. From table 3 when can conclude that the expected time complexity of calculating the overlay of two subdivisions on the sphere using a planar overlay algorithm is $O(n)$. However since the worst case time complexity of the overlay algorithm is $O(n^2)$, the worst case time complexity of the overlay of two subdivisions on the sphere is also $O(n^2)$.

As we already mentioned in the introduction mapping the structure on the sphere to the plane is not necessary. The planar algorithm described in this paper can be easily adapted to calculate the overlay of both subdivisions directly on the sphere. The extra time needed to map the structure on the sphere to the plane would not be needed. In table 3 we can see that this is less than 10% of the entire operation. If we were to

calculate the overlay on the sphere directly we would have to calculate the intersection point of two arcs on the sphere instead of two lines in the plane. However calculating the intersection of two arcs on the sphere takes about 3 times more time than the intersection of two lines in the plane. This would result in 200% extra time if we were to calculate the overlay on the sphere instead of in the plane. So mapping the structure on the sphere to the plane first is a cost effective step if we look at the real time of the actual overlay.

7 References

1. H. Bekker , K. De Raedt Mapping Graphs on the Sphere to the Finite Plane Volume 2331, Issue , pp 0055 Lecture Notes in Computer Science
2. Tuzikov, A. V., Roerdink, J. B. T. M., and Heijmans, H. J. A. M. Similarity measures for convex polyhedra based on Minkowski addition. *Pattern Recognition* 33, 6 (2000), 979-995.
3. Bekker H., and Roerdink, J. B. T. M. Calculating critical orientations of polyhedra for similarity measure evaluation. In Proc. 2nd Annual IASTED International Conference on Computer Graphics and Imaging, Palm Springs, California USA, Oct 25-27 (1999), pp. 106-111.
4. Roerdink, J. B. T. M., Bekker H. Similarity measure computation of convex polyhedra revised. Lecture notes in computer science, vol. 2243 (2001), pp. 389-399
5. Bekker H., and Roerdink, J. B. T. M. An efficient algorithm to calculate the Minkowski sum of Convex 3D polyhedra. Lecture notes in computer science, vol. 2073 (2001), pp. 619
6. Mehlhorn K., Naher S., *Leda: A Platform for Combinatorial and Geometric Computing*, (Cambridge, 2000)