ISSN 0280-5316
ISRN LUTFD2/TFRT--5816--SE

Intuitive Lead Through-Programming of Steel Grinding Robots

Jesper Notander

Department of Automatic Control
Lund University
May 2008

Lund University
Department of Automatic Control
Box 118
SE-221 00 Lund Sweden

Document name
MASTER THESIS

Date of issue
May 2008

Document Number
ISRN LUTFD2/TFRT--5816--SE

Author(s)
Jesper Notander

Supervisor
Jens Hofschule ABB CRC, Mannheim
Anders Robertsson Automatic Control, Lund
Rolf Johansson Automatic Control, Lund (Examiner)

Sponsoring organization

Title and subtitle
Intuitive Lead Through-Programming of Steel Grinding Robots (Intuitivs instruerat programmerande av stålslipande robotar)

Abstract
The SMErobot$^\texttrademark$ is an EU project aiming at giving small and medium enterprises the benefits of robotic automation. This thesis tries to solve the problem of teaching a robot how to grind a surface. The algorithm presented in this thesis reconstructs a surface by triangulating a point cloud. From this surface a path, covering the whole surface, is computed and translated into a RAPID program. The algorithm is integrated into the ABB Lead-Through Server.

Keywords

Classification system and/or index terms (if any)

Supplementary bibliographical information

ISSN and key title
0280-5316

ISBN

Language
English

Number of pages
85

Recipient's notes

Security classification

Acknowledgements

This thesis has been done within the European Commission's Sixth Framework Programme as part of the project SMErobot^{TM} under grant no. 011838.

I would like to thank Associate Professor Anders Robertsson at the department of Automatic Control, Lund University Faculty of Engineering, for arranging the contact with ABB Corporate Research in Ladenburg Germany. Without his help I would not have had the opportunity to do my Master Thesis there.

I also want to thank my supervisor at ABB, Doctor Jens Hofschulte, for his belief in me and the support he gave me throughout my work. My colleague, Sanparit Puanaiyaka, deserves a special thanks for his patience and good spirit during the last days.

Abstract

The SMErobot is an EU project aiming at giving small and medium enterprises the benefits of robotic automation. This thesis tries to solve the problem of teaching a robot how to grind a surface. The algorithm presented in this thesis reconstructs a surface by triangulating a point cloud. From this surface a path, covering the whole surface, is computed and translated into a RAPID program. The algorithm is integrated into the ABB Lead-Through Server.

Contents

List of Figures

Algorithms

Notations

Sets, vectors and scalars	
$X = \{\underline{p}_1,\ ...,\ \underline{p}_n\}$	A set of coordinates
$X_B = \{\underline{p}_1,\ ...,\ \underline{p}_n\}$	A set of bucket coordinates
s	A scalar
$\underline{p}$	A coordinate or point
$\mathbf{v}$	A vector
$\underline{v}$	A vertex
$G(V, E)$	A graph
$M(V, E)$	A mesh
$V = \{\underline{p}_1,\ ...,\ \underline{p}_n\}$	A set of vertices
$E = \{\{\underline{v}, \underline{w}\}_1,\ ...,\ \{\underline{x}, \underline{y}\}_n\}$	A set of edges

Operators	
$\mathcal{D}(\underline{p}, r)$	The discretization operator
$\mathcal{D}(B)$	The bucket index operator
$\mathcal{N}_X(\underline{p}, r)$	Neighborhood operator
$\mathcal{N}_X(B)$	Bucket vertex neighborhood operator.
$\mathcal{N}_{X_B}(\underline{p})$	Bucket index neighborhood operator
$\mathcal{N}_{adj}(\underline{p})$	Adjacency operator

Chapter 1

Introduction

A short introduction of this thesis and its place within the SMErobot project is presented in this chapter. First a brief description of the SMErobot project will be made. This is followed by the reasons for incorporating the grinding process into the project. Finally an overview of the developed algorithm will be given.

1.1 SMErobot

The SMErobot project is an Integrated Project within the 6th Framework Program of the European Union. The aim of the project is to grant small and medium sized enterprises the benefits of robotic automation.

The SMErobot project is split into three parts.

- Shop-floor-suitable devices for intuitive robot interaction.
- Task and motion definition without explicit programming.
- Physically harmless and low-cost robot mechanic design.

The intended user is a worker without any previous programming knowledge thus the target platform should be a system capable of understanding human-like instructions e.g. voice commands, manual guidance of robot etc. The setup of the system should be done by an experienced operator.

This thesis focuses on the second part of the SMErobot project and utilizes the concept of Lead-Through Programming to achieve the defined goals.

1.2 Grinding in SMErobot

The process of grinding a metal casting is today a hard and unhealthy job. The vibrations damage the blood vessels in the hand and this result in a condition called white fingers. Because of this a worker is not allowed to work more than 90 minutes without a break.

The time it takes to finish the grinding of a casting varies but in average it is around 30 minutes. This in combination with the health restrictions creates a bottleneck in the workshop process.

Automating the grinding would lead to a better situation for the workers and higher efficiency of the workshop.

1.3 Problem Description

Before automating the grinding process there are several issues to take into consideration. A grinding algorithm should be able to work for most situations a worker can handle today.

The most important factors concerning the grinding process is the surface geometry and the size of the contact area of the grinding disc.

It is assumed in this thesis that the surface geometry is simple. The surfaces under consideration are planar or slightly curved. They are also free from fine details such as edges and holes.

The area of the contact area of the grinding disc is important to know. This is to assert that the surface of grinding will be completely covered by the grinder. It is also useful for minimizing overlapping strokes.

Both the disc angle towards the surface and the disc radius influence the size of the contact area. The disc radius decreases during operation and the angle varies between 30° and 45°.

In this thesis the aim is to consider the dynamic properties of the grinding disc. It is not, however, in the scope of this thesis to construct a model of the changing contact area.

A final issue is that the material to be removed is not uniformly distributed on the surface. This problem is only briefly covered in this thesis.

1.4 Algorithm Description

The main idea behind the grinding algorithm is to reconstruct a grinding surface from a point cloud. A path, visiting all mesh vertices exactly once, is then generated and converted into robot coordinates.

The treating of the reconstructed surface as a graph is due to the work in [11]. They present an algorithm for covering a surface by splitting the surface into a grid, which is easily converted into a graph. It is assumed that by visiting all nodes in the graph the complete surface will be covered.

The assumption holds when the size of the contact area of the grinding disc is sufficiently large. The algorithm developed in this thesis will cover the surface if the tool used does not change its contact area during operation.

The grinding algorithm developed in this thesis can be split up in four significant steps: surface recording, surface reconstruction, path planning and path execution, see Figure 1.1. The main focus of this thesis is on the second, third and fourth part. These three steps will be presented first and the remaining two steps are presented together with the Lead-Through Programming.

Surface recording, or point sampling, is a simple step that should only be run once, for each surface. It records the robot position and tool orientation and makes the data available for the surface reconstruction step.

Surface reconstruction together with surface optimization is the most important steps of the algorithm. Several parameters determine the result and it

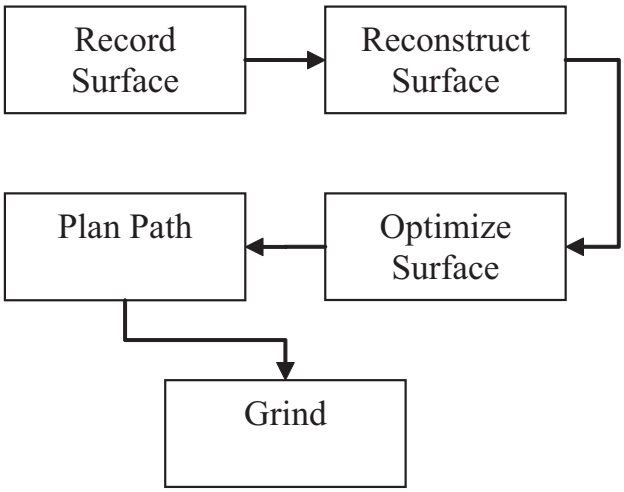


Figure 1.1: A schematic flowchart of the algorithm developed in this thesis.

is important for the operator to be familiar with the influence theses parameters have on the outcome. The surface optimization depends on the disc diameter which means that when the disc diameter changes the optimization has to be redone.

The path planner calculates a path, covering the whole surface, from the reconstructed surface. This step can fail. If the path planner cannot find a path one or more of the previous steps has to be done again, with different settings.

Path execution converts the generated path into a sequence of robot coordinates that is passed to the robot. It also considers the setting of robot parameters, e.g. speed, tool and zone, and the correct handling of the grinder.

The different steps of the algorithm are implemented with C# on the .NET platform, version 3.0.

Chapter 2

Surface Reconstruction

The problem of reconstructing a surface from a point cloud has attracted a lot of attention the past few years. Several different techniques have been developed, focusing on different aspects of the mesh construction.

When considering an algorithm for surface reconstruction one has to analyze the problem at hand to see which aspects the meshing algorithm should optimize. The surface reconstruction problem considered in this thesis has two significant properties: the collection of points is done previous to the meshing and the meshing is repeated several times with different optimization parameters.

The algorithm in this thesis is a modified version of the algorithm proposed in [1] whereas that algorithm is an online algorithm the one developed in this thesis is running offline. Online in this context means that the collection of points is done simultaneously with the meshing. Offline means that all points are collected prior to the meshing.

In the following sections the different steps of the meshing algorithm will be described in more detail. Emphasis will be laid on the difference between the algorithm in [1] and the one developed in this thesis.

Section 2.1 describes the data structure containing the point cloud, Section 2.2 covers the normal estimation, Section 2.3 covers the triangulation and Section 2.4 deals with the final optimization of the mesh.

2.1 Data Acquisition and Representation

The meshing algorithm developed in this thesis relies on the underlying data structure, containing the collected points, to be fast and efficient at finding local point sets close to a given point. The reason for this is that the meshing is done locally: each meshing step is only affecting a set of points close to each other, also called neighborhoods. A data structure that fulfills this requirement is the bucket space, see [7] or [1]. This data structure will be used for all set of points in the surface reconstruction algorithm.

2.1.1 Neighborhood

The spherical neighborhood of a point $\underline{p} = (p_x, p_y, p_z)^T$ in a point set X is defined as all points in X that are inside a sphere of radius R and with center at $\underline{p}$.

$$\mathcal{N}_X(\underline{p}, R) = \{\underline{q} \in X | \ ||\underline{q} - \underline{p}||_2 \leq R\} \tag{2.1}$$

R is called the neighborhood radius.

2.1.2 Bucket Space

The main idea behind the data structure is to divide the point space into a grid with a fixed resolution r. Each grid point corresponds to a cube with side length r. The cubes are called buckets. The bucket space is visualized in Figure 2.1.

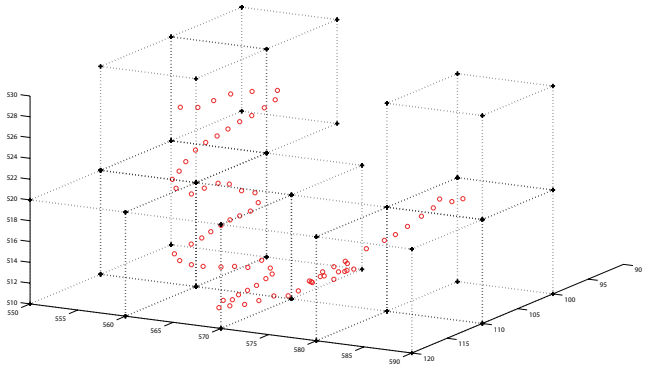


Figure 2.1: A point set divided into buckets.

To determine which bucket a point belongs to it is divided by r and rounded towards negative infinity. The operator $\mathcal{D}(\underline{p}, r)$ is the discretization operator and is defined as,

$$\mathcal{D}(\underline{p}, r) = \left\lfloor \frac{\underline{p}}{r} \right\rfloor \tag{2.2}$$

where $\lfloor \ \rfloor$ is the floor operator, rounding the components of $\underline{p}$ towards the integer closest to negative infinity. The discretization operator is also defined for a bucket B, with side resolution r, as,

$$\mathcal{D}(B) = \mathcal{D}(\underline{p}, r), \ \underline{p} \in B \tag{2.3}$$

when $B \neq \emptyset$. It is also called the (bucket) index operator. This operator is used in the following definitions of the bucket neighborhoods.

Similar to the definition of the point neighborhood, the bucket neighborhood, of a bucket B, is defined as:

$$\mathcal{N}_{X_B}(B) = \{\underline{q} \in X_B | \ ||\underline{q} - \mathcal{D}(B)||_2 \leq \sqrt{3}\} \tag{2.4}$$

where X_B is the set of bucket indices. In words the definition can be expressed as: the bucket neighborhood of B is all buckets whose index has no component that differs more than 1 from the corresponding component in B:s index.

In this thesis the use of the term bucket neighborhood will be used both for describing the set of discrete indices of the buckets and the volume defined by the indices and the resolution r. The set of vertices in the bucket neighborhood of B is defined in (2.5), where X is the set of points in the bucket space.

$$\mathcal{N}_X(B) = \{\underline{q} \in X | \ \mathcal{D}(\underline{q}, r) \in \mathcal{N}_{X_B}(B)\} \tag{2.5}$$

Implementation

Each bucket has a binary tree containing the points which is inside the volume of the bucket. The bucket also contains a list of its neighboring buckets, determined by (2.4). The definition of the bucket neighborhood in (2.4) says that there can only be 26 bucket neighbours to a bucket B.

The buckets are inserted into a binary tree and sorted by their discrete coordinates. The binary tree assures good insertion and search speed. The time complexity for both operations are $O(log(n))$.

When inserting a new point, into the bucket space, the corresponding bucket is retrieved and the point is inserted in it. If there is no bucket corresponding to the discretized point coordinate a new one is created. The new point is inserted into the bucket, see Algorithm 2.1 for the source code.

Algorithm 2.1: Insertion of a new coordinate into a bucket space.

```
void Insert(TValue value) {
    Vector key = this.BucketKey(value);
    Bucket<TValue> bucket;
    this.count++;
    if (this.buckets.TryGetValue(key,out bucket)) {
        bucket.Add(value);
    } else {
        bucket = new Bucket<TValue>(this.neighborhoodRadius, key);
        bucket.Add(value);

        // Adds the new bucket to the neighborhood of the old buckets
            and them to it.
        foreach (Bucket<TValue> neighbour in this.BucketNeighborhood(
            key)) {
            bucket.Add(neighbour);
            neighbour.Add(bucket);
        }
        this.buckets.Add(key, bucket);
    }
}
```

2.1.3 Neighborhood Search

Conducting a neighborhood search or retrieving the neighborhood of a point, $\underline{p}$, shows the power of the bucket space. The neighborhood search builds on the assumption that all neighbors of $\underline{p}$ are inside the bucket neighborhood of its associated bucket, B. This means that only the bucket neighborhood of B, $\mathcal{N}_X(B)$, needs to be searched for neighbors. In Figure 2.2 the relevant sets of vertices can be seen.

Implementation

The search is done by retrieving the bucket, B, corresponding to the discretized coordinate of $\underline{p}$, see Algorithm 2.2. As previously assumed all neighbors are inside the neighborhood of B. Retrieving the neighbors of B can be done in constant time. This is because that each bucket contains a list of its neighbors. If a point in the bucket neighborhood fulfills the criteria in (2.1) it is a neighbor.

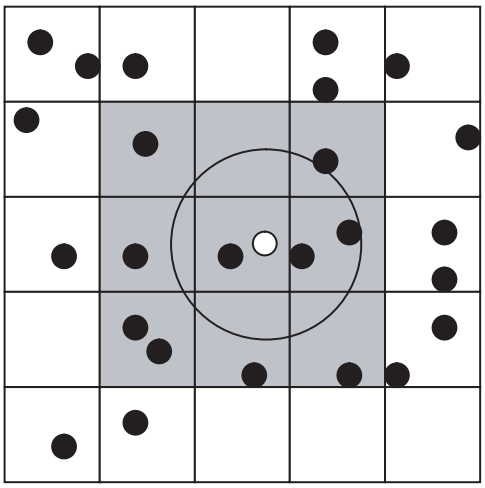


Figure 2.2: A two dimensional neighborhood search. The light grey area is the white points neighborhood. The dark grey squares are the buckets that will be checked in the neighborhood search, of containing a neighbor.

Algorithm 2.2: Discretizing of a point coordinate.

```
Vector BucketKey(TValue value) {
    double x = Math.Floor(value.Point[0] / this.neighborhoodRadius);
    double y = Math.Floor(value.Point[1] / this.neighborhoodRadius);
    double z = Math.Floor(value.Point[2] / this.neighborhoodRadius);
    return new Vector(x, y, z);
}
```

By setting the grid resolution to R, see (2.1), the search is time optimal and the previous assumption is valid. The search is time optimal because R is the smallest r for which the assumption is valid. If the resolution is smaller than the neighborhood radius then the assumption is invalid, because a neighbor could be in a bucket outside the bucket neighborhood. If the resolution is greater than the neighborhood radius the assumption is valid, but it also means that a larger volume has to be tested. This means that several more unnecessary neighbor checks will be conducted than if the neighborhood radius is used. This results in a less than optimal search.

The implementation of the search is straight forward and can be seen in Algorithm 2.3. Note that the `center` point is in the neighborhood if it is inserted into the bucket space.

Algorithm 2.3: Neighborhood search and neighborhood criteria.

```
List<TValue> Neighborhood(TValue center, double radius) {
    IList<Bucket<TValue>> buckets = this.BucketNeighborhood(center);
    List<TValue> values = new List<TValue>();

    foreach (Bucket<TValue> bucket in buckets) {
        foreach (TValue value in bucket) {
            if (this.IsNeighbour(center, radius, value)) {
                values.Add(value);
            }
        }
    }
    return values;
}

bool IsNeighbour(TValue center, double radius, TValue value) {
    return center.DistanceTo(value) <= radius;
}
```

2.2 Normal Estimation

The remeshing algorithm, presented later, relies on the correct estimation of the surface normal in each point. Estimating the normal of a point, $\underline{p}$, is done by fitting a tangent plane through the point neighborhood of $\underline{p}$

By treating the point cloud as a distribution of three random variables the problem of calculating the normal of the tangent plane is the same as finding the direction where the distribution has the smallest variance. A small variance corresponds to a small separation of points.

Two different cross sections of the same neighborhood are shown in Figure 2.3 and Figure 2.4, with vectors pointing in the direction of largest variance.

2.2.1 Estimation

Finding the variance of the distribution is done by calculating the covariance matrix. The standard definition of the covariance matrix and the mean value can be seen below,

$$c_{i,j} = \sum_{k}^{N} (x_{k,j} - \mu_i)(x_{k,j} - \mu_j) \tag{2.6}$$

$$\mu_i = \frac{1}{N} \sum_{k}^{N} x_{k,i} \tag{2.7}$$

where $c_{i,j}$ corresponds to the matrix element with row index i and column index j, $x_{k,j}$ is the k:th point and j:th component and μ_i is the mean value, see (2.7), of the i:th component.

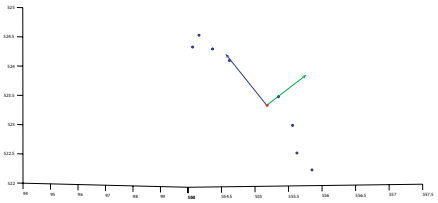


Figure 2.3: A couple of points taken from the same surface seen at a small variance cross section.

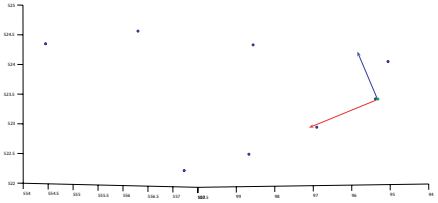


Figure 2.4: Same points as in Figure 2.3 but seen from angle with a high variance cross section.

Diagonalizing the covariance matrix of the point distribution gives the eigenvectors and eigenvalues which correspond to the variance and direction of the variance respectively. The eigenvector belonging to the largest eigenvalue points in the direction of the largest variance, the second eigenvector points in the direction with the largest variance orthogonal to the first eigenvector and so on.

The smallest eigenvalue, λ_{min}, corresponds to the eigenvector pointing in the direction of the smallest variance, e_{min}, of the distribution. This direction is taken as the normal of the tangent plane.

The normal of the tangent plane can be directed in the opposite direction of the actual surface. To avoid this the normal is multiplied with the tool orientation, $\mathbf{o}$, at the point, so that the correct sign can be obtained.

$$\mathbf{n} = \mathbf{e}_{\min}\text{sign}\left(\mathbf{e}_{\min}^{\mathrm{T}}\mathbf{o}\right) \tag{2.8}$$

Implementation

The estimation of the surface normals in the algorithm proposed in [1] is done continuously. For each new recorded point the normal is estimated and its neighbor's normals are reestimated. As described in the overview, see Figure 1.1, the surface recording, in this thesis, is done before surface reconstruction. This makes it possible to estimate the normal for all vertices in one step hence reducing calculation time.

The normal estimation method in Algorithm 2.4 is executed for all points once.

Special care has to be taken when a point has less than two neighbors, it is impossible to fit a plane through two points. The normal will not be

estimated for such a vertex and the vertex will be removed after the normal estimation of all vertices is completed.

Algorithm 2.4: Normal estimation of a point cloud

```
void EstimateNormal(SurfacePoint point) {
    List<SurfacePoint> points = this.values.Neighbourhood(point);
    if (points.Count > 2) {    // can't estimate the normal with less
        than three points.
        double[,] covMat = this.CovarianceMatrix(points);
        Vector normal = this.Eigen(covMat);
        point.Normal = normal * (normal * point.Orient < 0 ? -1 : 1);
    }
}
```

The method used to calculate the eigenvalues and eigenvectors is based on applying several transformations to a symmetric matrix with the aim of reducing it to diagonal form. The transformation used is the Jacobi rotation which applied to a matrix strives to eliminate one of the off diagonal elements.

The convergence of this method is quadratic and its time complexity is $O(n^3)$, for $n = 3$ this is approximately constant time. See [4] for theory and detailed implementation in C.

2.3 Meshing

The idea behind the algorithm developed in this thesis is to reconstruct the surface of grinding and then apply a path planning algorithm to this surface. The reasoning behind this approach is that, although it rises the level of complexity, it is a very flexible solution. It clearly separates the problem of physically teaching the surface of grinding and the subsequent path planning step.

An alternative approach is the so called brute force method. The idea behind that approach is to simply record the motion of the robot and then replay the recorded path. This is not a flexible solution and it is hard to apply later modifications to the path. The requirement of adjusting the path planning to the changing disc diameter was the strongest argument against this approach.

The meshing step can be divided into four sub-steps: data representation, neighborhood projection, edge intersection and edge insertion. This leads to the question of which data structures should be used to represent the mesh? Mainly two approaches have been considered and will be discussed in Section 2.3.1

Inserting a vertex into the mesh is described by simple rules. An overview of theses rules and the steps required to insert a new vertex is described in Section 2.3.2.

When inserting a new vertex, $\underline{v}_{new}$, in the mesh it is required to project its neighborhood onto the plane defined by its normal and coordinate. Care has to be taken so that all endpoints of edges that can intersect the new edges will be projected. This is described in Section 2.3.3. Determining if two line segments intersect is described in Section 2.3.4

2.3.1 Mesh Representation

Several data structures exist that can be used for representing a mesh. The simplest is the undirected edge. A mesh described by undirected edges stores information of its vertices and which vertices that are connected to which. It is easy to implement a representation based on the undirected edge. No special care has to be taken considering the ordering of edges.

The drawback is that no information about the triangles, or faces, of the mesh is stored. It is also hard to define the boundary of the mesh with the undirected edge representation. These drawbacks are important enough to consider alternatives to the undirected edge. The approach chosen is the half-edge data structure, which is also used in the meshing algorithm in [1].

An edge in a mesh represented by half-edges consists of a pair of half-edges. These half-edges are directed and connected in such a way that face and border information is available. This additional information comes with a price in additional computation time and implementation complexity.

The meshing step does not require the additional information provided by the half-edge and is implemented with the undirected edge. This is not the case when considering the optimization step. Before optimizing the mesh it is converted into a mesh represented by half-edges. The tradeoff between the different representations has some drawbacks which are discussed at the end of this chapter.

Definitions

A mesh, $M(V, E)$ is defined as a set of vertices, $V = \{\underline{v}_1,\ ...,\ \underline{v}_n\}$, connected by a set of edges, $E = \{\{\underline{v}, \underline{w}\}_1, ...,\ \{\underline{x}, \underline{y}\}_m\},\ v, w, x, y \in V$. A vertex correspond to a point in the point cloud and an edge connects two vertices. A vertex is said to be adjacent to another vertex if they are connected. This can be extended to the adjacency neighborhood of a vertex $\underline{v}$ defined in (2.9),

$$\mathcal{N}_{adj}(\underline{v}) = \{\underline{w} \in V | \{\underline{v}, \underline{w}\} \in E\} \tag{2.9}$$

where V is the set of vertices in the mesh and E is the set of vertex pairs

corresponding to the edges. The degree of a vertex is the number of edges connected to it.

Considering the half-edge data structure a face is defined as the surface between three counterclockwise connected half-edges, see Figure 2.5. The border of a mesh is defined as the vertices with half-edges not belonging to a face.

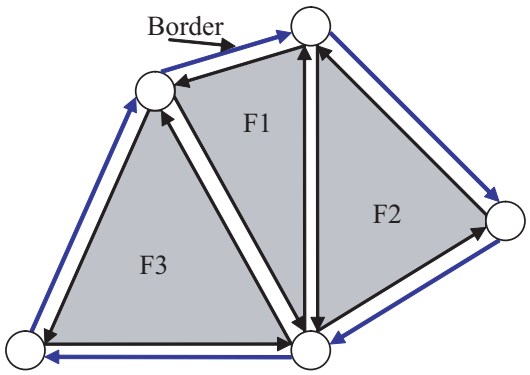


Figure 2.5: Graphical interpretation of a simple mesh with a half-edge data structure.

Implementation

Representing the mesh with undirected edges is straightforward. Each vertex correspond to a point in the point cloud, represented by a `robtarget`, see Appendix C for a definition. Each vertex also contains a list of vertices that is their adjacency neighborhood. Inserting and removing a vertex from the list corresponds to inserting and removing an edge. The edges can be retrieved by creating temporary edges from the vertex and its adjacent vertices, see Algorithm 2.5.

Algorithm 2.5: Retrives the edges of a vertex.

```
List<Vertex> neighbours;
IList<Edge> Edges {
    get {
        List<Edge> list = new List<Edge>(this.Degree);
        foreach (Vertex neighbour in this.neighbours) {
            list.Add(new Edge(this, neighbour));
        }
        return list;}}
```

The half-edge data structure is more complex. Different implementations exist and the one used in this thesis can be seen in Algorithm 2.6. Only the most important attributes have been included.

Algorithm 2.6: Attributes of the half-edge data structure used in this thesis.

```
class HalfEdge : IComparable<HalfEdge> {
    // ... //
    Vertex target;
    HalfEdge next;
    HalfEdge previous;
    HalfEdge mirror;
    Face face;
    // ... //
}
```

An edge consists of two half-edges referenced by each other with the `mirror` pointer. The `target` of a half-edge, `he`, is the vertex where `he` is incident to. The `next` pointer points at the next counterclockwise outbound half-edge at the target of `he`. The `previous` pointer is pointing to the half-edge with its `next` pointer pointing to `he`. If the half-edge belongs to a face it is stored in `face`, see Figure 2.6 for a graphical interpretation of the references.

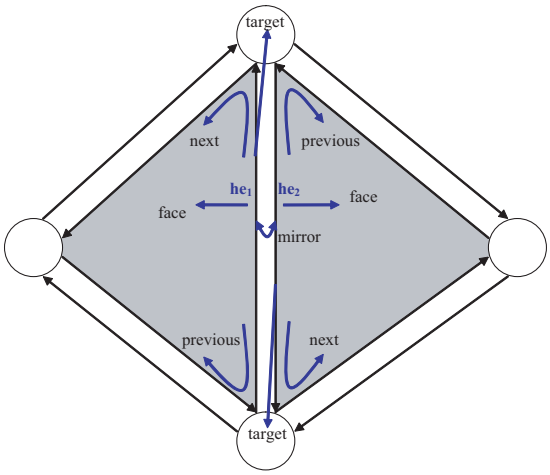


Figure 2.6: Graphical interpretation of the half edge data structure. Blue arrows correspond to pointers and the arrow points at the target of the pointer.

Switching representation

In the beginning of this section it is stated that a conversion between the two mesh representations is needed. The conversion is conducted by replacing all edges with half-edge pairs. Then for each vertex, $\underline{v} \in V$, and its adjacency neighborhood is projected onto the plane defined by $\underline{v}$:s normal. The half-edges are then sorted according to their angle from the positive x-axis. Finally the `next` and `previous` pointers are adjusted according to the sorting. The source code can be seen in Algorithm 2.7

Algorithm 2.7: Converts the undirected edge representation into a half-edge representation

```
void ConstructFaces() {
    // removes previous face information and sorts the half-edges for
        each vertex
    foreach (Vertex vertex in this.values) {
        vertex.ClearFaces();
        vertex.Adjust();
    }

    // Creates the faces and identifies the border //
}
```

Algorithm 2.8: Sorts the vertex' half edges counterclockwise according to the vertex' projection.

```
void Adjust() {
    if (this.Degree > 0) {
        List<HalfEdge> halfEdges = new List<HalfEdge>(this.Degree);
        halfEdges.AddRange(this.HalfEdges);
        if (halfEdges.Count == 1) {
            halfEdges[0].Mirror.Next = halfEdges[0];
        } else if (halfEdges.Count == 2) {
            halfEdges[0].Mirror.Next = halfEdges[1];
            halfEdges[1].Mirror.Next = halfEdges[0];
        } else {
            // project the neighborhood
            double[,] rotMat = this.MakeRotationMatrix();
            this.Project2D(rotMat, this.Point);
            foreach (HalfEdge edge in halfEdges) {
                edge.Target.Project2D(rotMat, this.Point);
            }
            // Sort edges and adjust the pointer references.
            halfEdges.Sort(new AngleComparer());
            halfEdges.Add(halfEdges[0]);
            for (int i = 1; i < halfEdges.Count; i++) {
                halfEdges[i].Mirror.Next = halfEdges[i - 1];
            }}}}
```

After the half-edges are constructed, and correctly connected, the faces and the border can be constructed. It is done by going through all half-edges and checking wether they belong to a face. A half-edge, `he`, belongs to a face if the following condition hold `he.next.next.next = he`, if not the half-edge belongs to the border, see Algorithm 2.9 for the face construction source code.

Algorithm 2.9: Constructs the faces in a halfedge mesh.

```
void ConstructFaces() {
    // ... //
    foreach (Vertex vertex in this.Vertexes) {
        foreach (HalfEdge edge in vertex.HalfEdges) {

            if (edge.Face == null && edge.Next.Next.Next == edge) {
                Face face = new Face(edge);

                edge.Face = face;
```

```
                edge.Next.Face = face;
                edge.Next.Next.Face = face;

                faces.AddLast(face);
            } else {
                edge.Target.OnBorder = true;
                edge.Origin.OnBorder = true;
                edge.Face = Face.Outside;
                border.AddLast(edge.Target);
            }
        }
    }
    // ... //
}
```

2.3.2 Insertion

When a new vertex, $\underline{v}_{new}$, is inserted into the mesh its neighborhood has to be remeshed. This is done by first projecting the neighborhood onto the plane defined by $\underline{v}_{new}$:s normal $\mathbf{n}_{v_{new}}$. This is followed by constructing new edges between the neighbours and $\underline{v}_{new}$. These new edges are sorted according to length. Beginning with the shortest, the new edges are tested for insertion according to the following two rules.

- The new edge is inserted if no old edges intersect the new edge.
- The new edge is inserted if all intersecting edges are longer than the new edge.

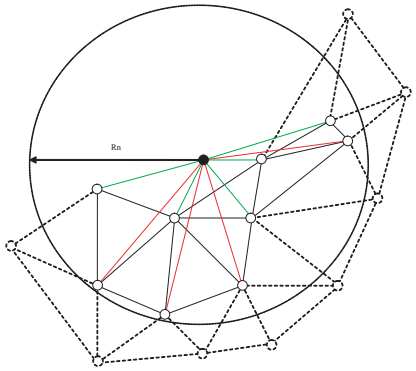


Figure 2.7: A new vertex is inserted. Candidate edges are drawn in red if they are invalid and in green if they are valid.The circle is the neighborhood radius of the mesh.

Figure 2.7 shows the insertion of a new vertex. The red edges will not be inserted. The green edges will be inserted and all black edges that intersects with them will be removed from the mesh. Pay especially attention

to the case when a green edge intersects an edge originating from inside the neighborhood but ending outside, see Figure 2.8 for the result.

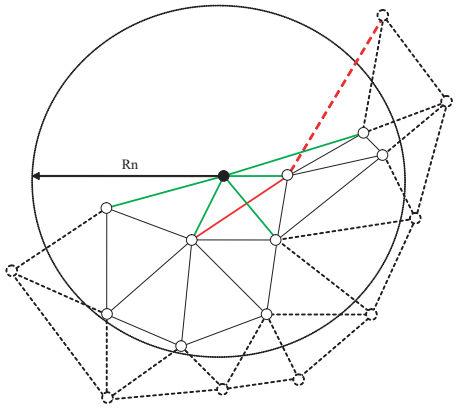


Figure 2.8: The result of the remeshing around a newly inserted vertex. Green edges are inserted reds removed.

Implementation

The method for inserting a new vertex is presented in Algorithm 2.10. The current implementation removes all old edges from the mesh prior to the insertion test. The insertion criteria concerns properties of edges and it is thus better to use edges instead of pairs of vertices. By removing the connections and storing them as edges locally it is also guaranteed that the local set of old edges will be unique.

Algorithm 2.10: This method inserts a new vertex into a mesh.

```
public bool Add(Vertex newVertex) {
            if (this.values.IsInsertable(newVertex)) {
                // retrieve the vertex neighborhood, without the
                    newVertex.
                List<Vertex> neighbourhood = this.values.Neighborhood(
                    newVertex);
                // add the vertex to the underlying bucket space,
                    without any vicinity check.
                this.values.Insert(newVertex);
                newVertex.Id = Vertex.currentId++;

                //project the neighbourhood.
                this.RotateVertexes(newVertex, neighbourhood);

                List<Edge> candidateEdges = new List<Edge>();
                List<Edge> oldEdges = new List<Edge>();

                // remove old edges from the mesh and create new
                    candidate edges.
                foreach (Vertex vertex in neighbourhood) {
```

```
                    candidateEdges.Add(new Edge(newVertex, vertex));
                    IList<Edge> edges = vertex.Edges;
                    oldEdges.AddRange(edges);
                    this.Remove(edges);
                }

                // insert the new edges and remove old intersecting
                    edges.
                this.Insert(candidateEdges, oldEdges);
                return true;
            } else {
                return false;
            }
        }
```

The insertion method can be seen in Algorithm 2.11. Given a list of new edges and the old edges from the neighborhood of $\underline{v}_{new}$ the new edges are tested for insertion. After all new valid edges are inserted the remaining old edges are reinserted into the mesh.

Algorithm 2.11: Insertion and testing of new edges.

```
private void Insert(List<Edge> candidateEdges, List<Edge> oldEdges) {
    foreach (Edge candidateEdge in candidateEdges) {
        List<Edge> intersectingEdges = new List<Edge>();
        bool valid = true;
        foreach (Edge oldEdge in oldEdges) {
            if (candidateEdge.Intersects2D(oldEdge)) {
                if (oldEdge.Length < candidateEdge.Length) {
                    valid = false;
                    break;
                } else {
                    intersectingEdges.Add((Edge) oldEdge);
                }
            }
        }
        if (valid) {
            foreach (Edge edge in intersectingEdges) {
                oldEdges.Remove(edge);
            }
            this.Insert(candidateEdge);
        }
    }
    this.Insert(oldEdges);
}
```

2.3.3 Projection

When a new vertex, $\underline{v}_{new}$, is inserted into the mesh the neighborhood of $\underline{v}_{new}$ has to be remeshed. The first step of remeshing is to project the neighborhood onto the plane defined by $\underline{v}_{new}$ and its normal, $\mathbf{n}_{\mathrm{v_{new}}}$. The normal of $\underline{v}_{new}$ defines a rotation matrix used in the projection.

$$\mathbf{R}_{\mathrm{v_{new}}} = \begin{pmatrix} \mathbf{n}_{\mathrm{x}} & \mathbf{n}_{\mathrm{y}} & \mathbf{n}_{\mathrm{new}} \end{pmatrix} \tag{2.10}$$

The column vectors, in (2.10), $\mathbf{n}_{\mathrm{x}}$ and $\mathbf{n}_{\mathrm{y}}$ are orthogonal to $\mathbf{n}_{\mathrm{new}}$ and each other. A good choice is to use the eigenvectors from the normal estimation step, see Section 2.2. The projection of a vertex, $\underline{v}_i$ is defined as follows.

$$\underline{v}_i' = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix} (\mathbf{R}_{\mathrm{v_{new}}} \underline{v}_i + \underline{v}_{new}) \tag{2.11}$$

Shown in Figure 2.7, edges, connecting vertices inside the neighborhood of $\underline{v}_{new}$ with vertices outside the neighborhood, also needs to be tested if they intersect a new edge. This means that the outside vertices, connected to neighborhood vertices, must be projected too. This extended set of vertices is called the extended neighborhood of $\underline{v}_{new}$, or the projection neighborhood.

Implementation

Projecting the extended neighborhood of a new vertex, $\underline{v}_{new}$, is done by the method seen in Algorithm 2.12. Checking if a vertex is in the neighborhood is a $O(D_{max} * n_{inside} * 2 + n_{outside}^2)$ operation, where D_{max} is the maximum degree of the neighborhood vertices. n_{inside} and $n_{outside}$ is the number of vertices inside and outside the neighborhood.

Algorithm 2.12: Method projecting the extended neighborhood of $\underline{v}_{new}$.

```
void RotateVertexes(Vertex newVertex, List<Vertex> neighbourhood) {
    double[,] rotMat = newVertex.MakeRotationMatrix();
    Vector offset = newVertex.Point;

    newVertex.Project2D(rotMat, offset);
    List<Vertex> outsideVertexes = new List<Vertex>();

    foreach (Vertex oldVertex in neighbourhood) {
        oldVertex.Project2D(rotMat, offset);

        foreach (Vertex neighbour in oldVertex.Neighbours) {
            if (!neighbourhood.Contains(neighbour) &&
               !outsideVertexes.Contains(neighbour)) {
                neighbour.Project2D(rotMat, offset);
                outsideVertexes.Add(neighbour);
            }}}}
```

2.3.4 Intersection

The insertion criteria build on the ability to check if two edges intersect. An edge can be considered as a segment of a line and the problem hence translates into checking if two line segments intersect. The intersection method described in this section uses the lines corresponding to the edges and check whether they intersect and if so determines if the intersection point is on both edges.

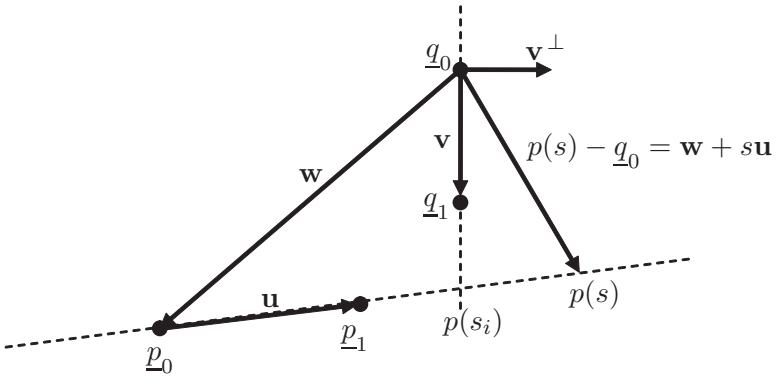


Figure 2.9: A graphical overview of the different vectors in the intersection algorithm.

Because both lines and segments will be used, in the intersection algorithm, a parametric representation is chosen. The parametric equation for a line is defined in (2.12). An edge is described by setting $\underline{p}_0$ and $\underline{p}_1$ to the edges endpoints and restricting s to $[0, 1]$.

$$p(s) = \underline{p}_0 + s(\underline{p}_1 - \underline{p}_0) = \underline{p}_0 + s\mathbf{u},\ s \in R \tag{2.12}$$

Two lines, $p(s) = p_0 + s\mathbf{u},\ q(t) = q_0 + t\mathbf{v}$, can either be parallel or intersect. The lines are parallel only if their directions are collinear, $\mathbf{v} = c\mathbf{u}, c \in R$. This means that the fractions of the components of the direction vectors must be the same.

$$\frac{\mathbf{u}_\mathrm{k}}{\mathbf{v}_\mathrm{k}} = \frac{\mathbf{u}_1}{\mathbf{v}_1}$$

In two dimensional space this translates into the *perp product* condition in (2.13).

$$\mathbf{u}^{\perp}\mathbf{v} = u_1 v_2 - u_2 v_1 = 0 \tag{2.13}$$

The perp product condition in (2.13) states that two lines are parallel in Euclidian space if they are both perpendicular to the same direction $\mathbf{u}^{\perp}$.

The easiest way to check if two parallel lines coincide is to test if a point on one line also lies on the other. When both lines are line segments, as in the edge case, they may or may not overlap. The solution is to solve, for two values of t, t_0 and t_1, the following equations.

$$\underline{p}_0 = q(t_0) \tag{2.14}$$

$$\underline{p}_1 = q(t_1) \tag{2.15}$$

If the parameters fulfills the requirement in (2.16) then the line segments overlap.

$$[t_0, t_1] \cap [0, 1] \neq \emptyset \tag{2.16}$$

In the case where the lines are not parallel they will always intersect at one point, in two dimensional space. The aim is to calculate the intersection point $p_i = p(s_i) = q(t_i)$. The parameters can be calculated as

$$s_i = -\frac{\mathbf{v}^\perp \mathbf{w}}{\mathbf{v}^\perp \mathbf{u}} = \frac{v_2 w_1 - v_1 w_2}{v_1 u_2 - v_2 u_1} \tag{2.17}$$

$$t_i = -\frac{\mathbf{u}^\perp \mathbf{w}}{\mathbf{u}^\perp \mathbf{v}} = \frac{u_1 w_2 - u_2 w_1}{u_1 v_2 - u_2 v_1} \tag{2.18}$$

where $\mathbf{w} = p_0 - q_0$. Two line segments intersects if,

$$s_i \in [0, 1] \tag{2.19}$$

$$t_i \in [0, 1] \tag{2.20}$$

For a graphical interpretation of the vectors see Figure 2.9.

Implementation

The implementation presented in this section will give a positive answer if two edges share the same end vertex. This is not desirable because it would result in that no edges could be connected to a common vertex. This has to be changed if the intersection algorithm is to be used to construct a mesh.

The first thing the intersection algorithm checks is whether the lines intersect. This is done by calculating the dot product of the vectors $\mathbf{u}^\perp$ and $\mathbf{v}$, see Algorithm 2.13, and subsequently checking if the product obeys the criteria in (2.13).

Algorithm 2.13: Outline of the intersection method.

```
bool IntersectTest(Edge other) {
    Vector u = this.Point2 - this.Point1;
    Vector v = other.Point2 - other.Point1;
    Vector w = this.Point1 - other.Point1;
    double D = u.Perp2D(v);
    if (D == 0) {
        // the lines are parallel, check if the segments overlap
        // ... //
    } else {
        // the two lines intersects, check if the segments intersects.
        // ... //
    }

double Perp2D(Vector other) {
    return this.data[0] * other.data[1] - this.data[1] * other.data
        [0];
}
```

If the lines are parallel the code in Algorithm 2.14 will be executed. First it checks if the lines overlap. If they overlap, the overlapping is determined by calculating the parametric values t_0 and t_1, according to (2.14) and (2.15). The parameters are then checked if they obey the criteria in (2.16). If they do the line segments also overlap except in the special case, when only two endpoints overlap.

Algorithm 2.14: The following code is executed if two lines are parallel.

```
if (u.Perp2D(w) != 0 || v.Perp2D(w) != 0) {
    return false;
} else {
    Vector w2 = this.Point2 - other.Point1;
    double t0 = 0;
    double t1 = 0;
    if (v[0] != 0) {
        t0 = w[0] / v[0];
        t1 = w2[0] / v[0];
    } else {
        t0 = w[1] / v[1];
        t1 = w2[1] / v[1];
    }
    if (t0 > t1) {
        double t = t0; t0 = t1;t1 = t;
    }
    // notice the >= and <= operators.
    if (t0 >= 1 || t1 <= 0) {
        return false;
    }
    return true;
}
```

In the case where two lines are not parallel the parametric values of the intersection point is calculated according to (2.17) and (2.18). If both parameters obeys the criterion in (2.19) and (2.20) the line segments intersects, see Algorithm 2.15. Before the parameter values are calculated the special case, when the intersection point is an endpoint, is taken care of.

Algorithm 2.15: If two lines are not parallel this code will be executed.

```
// calculate the intersection point
double si = v.Perp2D(w) / D;
if (si < 0 || si > 1) {
    return false;
}
double ti = u.Perp2D(w) / D;
if (ti < 0 || ti > 1) {
    return false;
}
// if the intersection point is an endpoint, return false.
if (this.Point22D == other.Point22D || this.Point22D == other.
    Point12D ||
    this.Point12D == other.Point22D || this.Point12D == other.
        Point12D) {
    return false;
}
return true;
```

2.4 Mesh Optimization

The aim of this step is to make the mesh more regular and to make the edges the same length as the diameter of the contact area of the grinding disc. The reason for doing this is that the algorithm assumes that by visiting all vertices exactly once the whole surface will be covered. Making all edges shorter than the disc diameter will accomplish this.

Two articles [8] and [10] propose ways of optimizing a triangular mesh. Both have good results but the first one is much faster than the second.

This section will describe a simple mesh optimization algorithm using one of the local operations presented in the previous articles. The section starts with a brief explanation of the edge and vertex operations used in mesh optimization. This is followed by the description of the algorithm.

2.4.1 Edge and Vertex Operations

The most commonly used operations in mesh optimization are: edge split, edge collapse, edge flip and vertex relocation. The edge operations are fundamental to changing the properties of a mesh.

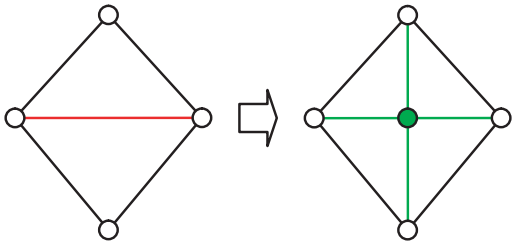


Figure 2.10: The edge split operation.

The edge split operation splits an edge into two and inserts up to two new edges between the middle point of the old edge and the opposite vertices in the two faces connected by the old edge.

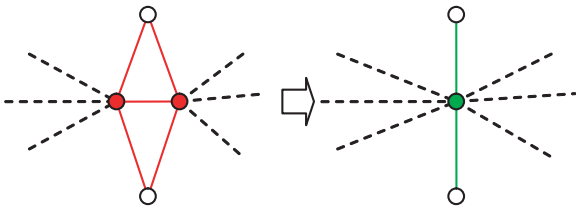


Figure 2.11: The edge collapse operation.

This edge collapse replaces an edge and its two endpoints with one vertex. This can be done in two ways. If both endpoints are inside the mesh then they are replaced by the center point of the edge. The new vertex is connected to the old edges endpoints neighbors that are still in the mesh. If one endpoint is on the border of the mesh then they collapse into that vertex. The reason for this is that the border geometry should not be changed due to an optimization operation.

If an edge is common to two faces then that edge can be flipped. The flipping operation removes the edge and inserts a new between the vertices not common to the two faces connected by the edge.

The vertex relocation is a more complex step and can be accomplished in several different ways. The idea is to move a vertex, inside its adjacency area, thereby making the mesh more regular.

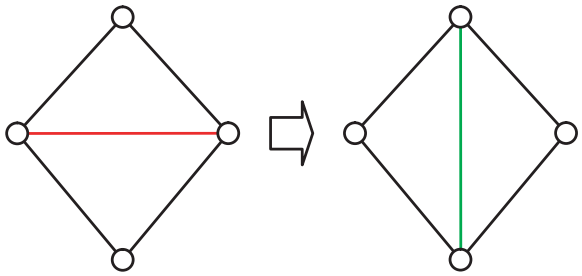


Figure 2.12: The edge flip operation.

2.4.2 Implementation

In the approach presented here only the edge split operation is used. Figure 2.13 shows an optimized mesh constructed by only using the edge split operation. The result is sufficient for the goals of this thesis.

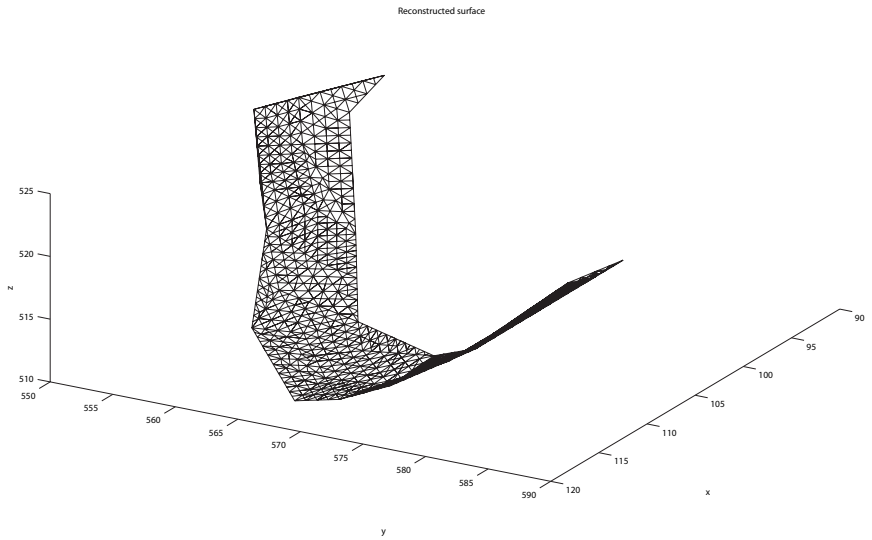


Figure 2.13: Optimized mesh after applying the edge split operation. All edges are shorter than the maximum optimization edge length.

The algorithm is simple. For each edge check if it is longer than the radius of the grinding disc. If true split the edge. Repeat this until all edges in the mesh are shorter or equal to the disc radius. See Algorithm 2.16 for the algorithm.

Algorithm 2.16: Optimizing the mesh.

```
void Optimize(double maximumOptimizationEdgeLength) {
    bool modified = true;
    while (modified) {
        modified = false;
```

```
        foreach (Edge edge in this.Edges) {
            if (edge.Length > 2 * maximumOptimizationEdgeLength) {
                this.Split(edge);
                modified = true;
                break;
            }}}}
```

Splitting an edge is performed by first creating a new vertex at the center of the splitting edge. The center vertex is the linear interpolation of the endpoints of the splitting edge. This is followed by retrieving the vertices that belongs to the face(s) the splitting edge belongs to. Finally the splitting edge is removed and new edges from the center vertex to the vertices retrieved in the previous step are inserted. The edge split code can be seen in Algorithm 2.17.

Algorithm 2.17: Splitting an edge.

```
private void  Split(Edge edge) {
    Vertex center = edge.MiddlePoint;

    List<Vertex> ends = new List<Vertex>(2);
    foreach (Edge edge1 in edge.Vertex1.Edges) {
        Vertex target1 = edge1.Vertex1 == edge.Vertex1 ? edge1.Vertex2
             : edge1.Vertex1;
        foreach (Edge edge2 in edge.Vertex2.Edges) {
            Vertex target2 = edge2.Vertex1 == edge.Vertex2 ? edge2.
                Vertex2 : edge2.Vertex1;
            if (target1 == target2) {
                ends.Add(target1);
            }
        }
    }

    List<Edge> newEdges = new List<Edge>(4);
    newEdges.Add(new Edge(center, edge.Vertex1));
    newEdges.Add(new Edge(center, edge.Vertex2));
    foreach (Vertex target in ends) {
        newEdges.Add(new Edge(center, target));
    }

    this.Remove(edge);
    this.values.Insert(center);
    this.Insert(newEdges);

}
```

The algorithm runs in $O(n^2)$, no formal proof is given only a short justification. Consider a mesh where all edges have the maximum allowed length. Assume that the desired edge length is a third of the maximum edge length [1]. Consider a mesh of two faces with edges at maximum length. After applying the optimization scheme a finite number of splits are done and that it is only the relation between the maximum edge length and the desired that determines the number of edge splits.

[1] This is the relation between the minimum edge length and the maximum edge length.

This implies that by splitting the edges an additional $C * n$ number of operations has to be done, where C is a constant. This justifies the assumption that the algorithm runs in $O(n^2)$.

2.5 Conclusions

In the current implementation of the surface reconstruction it is necessary to be careful when choosing the parameters. As reported in [1] their algorithm can construct holes and invalid triangles. This also applies to the algorithm in this thesis due to the fact that it is closely modeled on that algorithm.

The effect of the invalid triangles are only seen when the mesh is optimized, due to the conversion to the half-edge mesh representation. Splitting an edge belonging to an invalid triangle will result in edges crossing each other, creating even more invalid triangles.

A solution to this problem could be to abandon the half-edge representation. This will significantly make the algorithm slower. The reason is the additional effort required to construct the faces from undirected edges.

To construct the faces, of a mesh represented with undirected edges, the following algorithm could be used. For each vertex, v, check if its neighbors are connected to each other. A pair of connected adjacent vertices will form a triangle with v. This method in $\mathcal{N}(v_i),\ v_i \in \mathcal{N}(v)$ a comparison has to be made for every vertex in $\mathcal{N}(v)$. This results in a time complexity of $O(|\mathcal{N}(v)| * (max(|\mathcal{N}(v_i)|) * |\mathcal{N}(v)|))$.

The optimal degree is six for interior vertices and four for border vertices [8]. Because of the point density restrictions it can be assumed that the average degree will be close to the optimal. This means that the number of comparisons is of the same order as the mesh size in the examples considered in this thesis resulting in a $O(n^2)$ time complexity. Compared to the one implemented which has $O(n)$ time complexity.

The current implementation, with two different ways of representing the mesh, is a preliminary solution. Future development should only consider one way of representing the mesh. A suitable data structure is the half-edge, due to its ability to represent all aspects of a mesh that are important in this thesis.

Optimization

The results of the optimization are sufficient for this thesis. The regularity of the optimized mesh is good and the lengths of the edges are close to the desired length. The aim was to have edges with the same length but this is not possible with the current optimization algorithm.

The criterion for splitting an edge is that it is longer than the desired edge length. This creates edges with lengths between half the desired edge length and the edge length. This must be taken into consideration when setting the desired edge length. If edges are shorter than the desired edge length, which is the disc radius of the contact area of the grinding disc, the grinding strokes will overlap. This means that to much material is removed.

More complex optimization algorithms are presented in [8] and [10]. They use the full range of mesh operations, described in Section 2.4.1 and produces meshes of high quality.

Chapter 3

Path Planning

In Chapter 1 it was mentioned that the main idea behind the algorithm is to transform the surface mesh into a graph and find a path in the graph that visits all vertices exactly once. This problem belongs to the class of NP hard problems and is a special case of the traveling salesman problem. For an arbitrary graph there is no guarantee that there exist such a path, this requires special attention when searching for a path.

This chapter describes an algorithm for constructing the above mentioned path, also called a Hamiltonian path. The first section describes the underlying path construction algorithm. The second section explains the angle minimizing scheme which is used to increase the quality of the path. The third section introduces a timeout criteria which makes the algorithm more robust. Finally the conclusions are presented with some thoughts for future development.

3.1 Path Construction

Several different approaches exists for finding a Hamiltonian path, see [9] for a detailed survey. There exist two kinds of algorithms, heuristic and backtracking. The heuristic algorithms are based on a set of rules or techniques that solves the problem. The backtracking algorithms evaluate all possible solutions in a recursive manner. In general the heuristic algorithms are faster than the backtracking algorithms. The backtracking algorithm will always find the solution if it exists, which is not the case with heuristic algorithms.

In this thesis the Posá algorithm was chosen. The algorithm is a heuristic algorithm that is the result of the work of Posá in [5]. It was chosen because of its simplicity and extendability. Several additional heuristics was added, to the original algorithm, to improve the execution speed and path quality. One of these heuristics is the distance transform described in [11].

3.1.1 Definitions

Before delving any further into the path construction some basic definitions has to be made. A graph, $G(V, E)$, is defined in the same way as a mesh in the previous chapter, see Section 2.3.1. In this chapter there will be no distinction between a mesh, as defined in the previous chapter, and a graph. This is because only the connection information, which vertices are connected to which in the mesh, is used in the path planning step.

A path is defined as a sequence of vertices,

$$P = \{\underline{v}_1, ..., \underline{v}_n\} \tag{3.1}$$

where $\underline{v}_i \in V,\ i = 1, ..., n$. A vertex in the path must be adjacent to its predecessor and successor. Given a vertex, $\underline{v}_i$, this can be expressed as $\underline{v}_{i-1}, \underline{v}_{i+1} \in \mathcal{N}_{adj}(\underline{v}_i)$, the adjacency operator $\mathcal{N}_{adj}(\underline{v}_i)$ was defined in (2.9). The size of a path, $|P|$, is the number of vertices in the path.

A path is said to be partial if not all vertices in V are in P.

A loop is a subsequence, of a path, that starts and ends at the same path vertex. The path $\{\underline{v}_1, \underline{v}_2, \underline{v}_3, \underline{v}_4, \underline{v}_2, \underline{v}_5\}$ contains a loop, which is $\{\underline{v}_2, \underline{v}_3, \underline{v}_4, \underline{v}_2\}$.

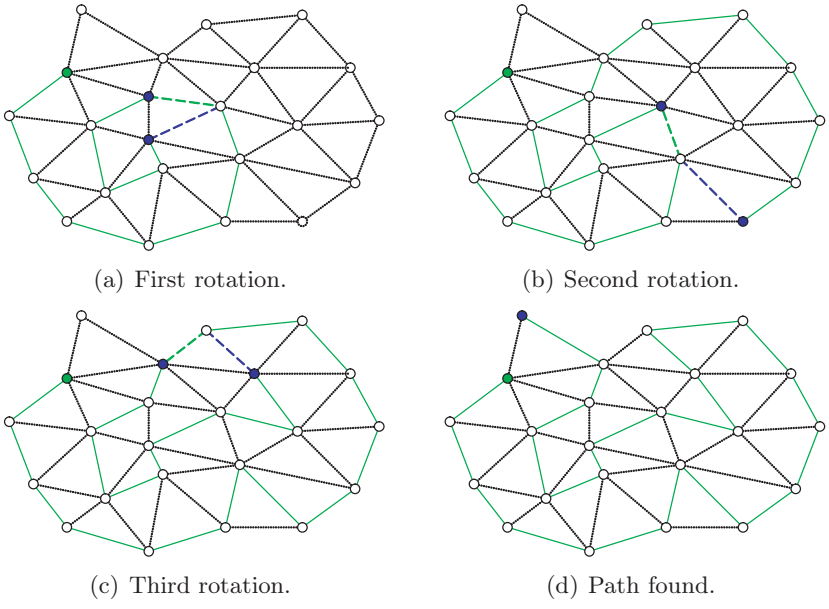
When a path contains all vertices in a graph and is without any loops it is said to be a Hamiltonian path e.g. each vertex is only present once in the path. In this chapter only Hamiltonian paths and partial paths without any vertex appearing more than once will be considered. Whenever a reference is made in this chapter to a path it is a Hamiltonian path if nothing else is stated.

3.1.2 Posás Algorithm

Posá's algorithm is a random heuristic algorithm that uses a rotation transformation to extend a partial path. It is straight forward to implement but suffers from the lack of a structured way of finding the path. It is guaranteed to find a path if it exists according to [5], but because it is a random search it could take a considerably amount of time to find a solution.

Starting at a random vertex the Posá algorithm construct the path by selecting a random vertex adjacent to the end vertex of the path. The algorithm only selects vertices that are not in the path. If no such vertices exist then the path is either finished or stuck. If the path is stuck the Posá transform is applied.

The transform creates a loop by connecting the endpoint, $\underline{v}_n$, to a random path vertex called the pivot vertex, $\underline{p}_p$. The next step is to break the path at the $\underline{v}_p\underline{v}_{p+1}$ transition, thus breaking the cycle. Reversing the order of the cycle vertices completes the transform, so that the last path vertex is v_{p+1}. See Figure 3.1 for a graphical interpretation of the transform.



(a) First rotation. (b) Second rotation.

(c) Third rotation. (d) Path found.

Figure 3.1: Illustrating the use of the rotation transform in the Posá algorithm. A blue point is a tested end point at the current path length. When all neighbors of the end vertex are in the path a loop is created. This is shown with a blue dotted line. By splitting the loop a new endpoint is acquired. The slitting is shown with a green dotted line.

The algorithm keeps track of the endpoints of the current path length. If an endpoint only has path vertices adjacent to it and these vertices have all been endpoints for the current path, then the algorithm fails and no path exists.

Implementation

The implementation of the Posá algorithm can be seen in Algorithm 3.1. The `GetRandomNeighbour` method will be described in more detail later in this chapter because it is heavily modified from the original Posá algorithm.

Algorithm 3.1: The Posá algorithm.

```
GrindingPath GeneratePath() {
    // ... //
    path = new GrindingPath(this.start, this.mesh.NumberOfVertexes);
    do {
        Vertex neighbour = this.GetRandomNeighbour(path, path.End);
        if (neighbour == null) {
                this.interrupted = true;
        } else if (path.Contains(neighbour)) {
            path.Rotate(neighbour);
        } else {
            path.Add(neighbour);
        }
    } while (path.Count < this.mesh.NumberOfVertexes && !this.
        interrupted);
    // ... //
    return path;
}
```

The rotational transform is executed on the path and can be seen in Algorithm 3.2. The rotation is performed as a reordering of the path vertices. Starting with the path vertices not in the loop they are reinserted in the path in their current order. When the pivot vertex is reached the remaining path vertices is inserted, in reverse order. This is an $O(n + \log n) = O(n)$ operation. The $\log n$ factor comes from the retrieval of the list element containing the pivot vertex.

Algorithm 3.2: The Posá rotational transform.

```
void Rotate(Vertex vertex) {
    LinkedListNode<Vertex> pivot = this.nodes[vertex];

    if (pivot == path.First) {
        this.path.RemoveFirst();
        this.path.AddLast(pivot);
        this.endPoints.Add(vertex, vertex);
    } else if (pivot != path.Last && pivot != path.Last.Previous) {
        LinkedListNode<Vertex> first = this.path.First;
        LinkedListNode<Vertex> last = this.path.Last;
        LinkedListNode<Vertex> current;
        do {
            current = this.path.First;
```

```
            this.path.RemoveFirst();
            this.path.AddLast(current);
        } while (current != pivot);
        current = last;
        LinkedListNode<Vertex> prev;
        while (current != null) {
            prev = current.Previous;
            this.path.Remove(current);
            this.path.AddLast(current);
            current = prev;
        }}// ... //}
```

3.1.3 The Distance Transform

The basic version of the Posá algorithm is based on the random selection of neighbors, as previously explained. This has some drawbacks concerning the quality of the path and the execution speed. The problem is that the random selection creates an unstructured path with many turns and without regard to the surface geometry.

There is also a problem of the algorithm getting stuck. This problem arises when there is no path in the graph. Even if there is a path from the start vertex the algorithm can be very slow. When few paths exist it is a high possibility that the algorithm will try partial paths that are not in a Hamiltonian path. If there are many of these false tries the algorithm will be considerably slowed down.

To address these issues a heuristic based on the distance transform described in [11] was added to the Posá algorithm. The distance transform forms the basis for an algorithm, also presented in the article, that calculates a structured path covering a surface. That algorithm will produce a path that is not Hamiltonian thus making it useless for this thesis. Combining the distance transform with the Posá transform gives an algorithm that retains the structured properties of the distance transform and the ability to find Hamiltonian paths of the Posá algorithm.

The distance transform calculates the distance for each vertex, where the distance is the minimum number of edges between a vertex and a selected goal vertex, see Figure 3.2.

Implementation

The distance transform is performed by picking a random vertex from the border and then performing a width first search in the graph. For each new level of the search the distance is incremented and is assigned to the vertices

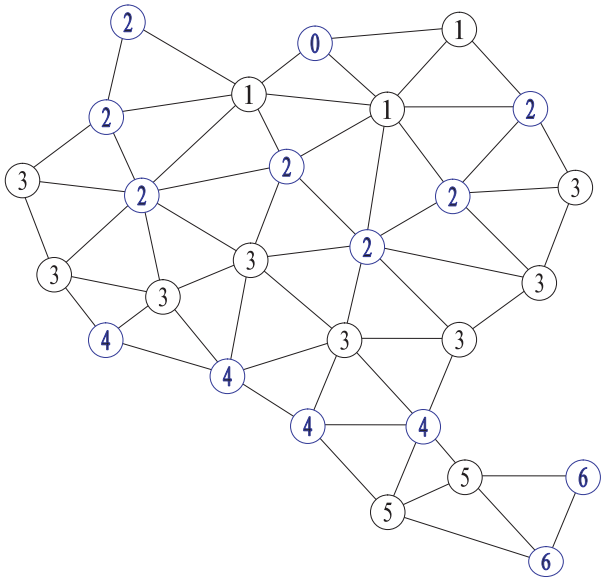


Figure 3.2: Figure showing the distance transform. The vertex with distance zero is the goal vertex.

at that level. The breath first search is sometimes referred to as a wave propagating through the graph.

Algorithm 3.3: Propagating a wave through the mesh by using the direction properties of half-edges.

```
private void PropagateWave() {
    // ... //
    current.Enqueue(goal);

    while (!finished) {
        if (current.Count == 0) {
            if (next.Count == 0) {
                finished = true;
            } else {
                distance++;
                current = next;
                next = new Queue<Vertex>();
            }
        } else {
            Vertex vertex = current.Dequeue();
            vertex.Distance = distance;
            foreach (HalfEdge edge in vertex.HalfEdges) {
                if (edge.Target.Distance == Vertex.NullDistance) {
                    next.Enqueue(edge.Target);
                }
                if (edge.Mirror.Target.Distance == Vertex.NullDistance
                    ) {
                    next.Enqueue(edge.Mirror.Target);
                }}}}}
```

The `GetRandomNeighbour` method in Algorithm 3.1 is modified to use the distance in the selection process. Instead of picking a vertex from the set of all adjacent vertices it picks a vertex from the adjacent vertices with the highest distance.

Algorithm 3.4: The neighbor selection method with emphasis on the distance transform.

```
Vertex GetRandomNeighbour(GrindingPath path, Vertex end) {
    // ... //
    foreach (Vertex vertex in end.Neighbours) {
        if (path.Contains(vertex)) {
            if (path.IsValid(vertex)) {
                if (vertex.OnBorder) {
                    onBorder.Add(vertex);
                } else {
                    inPath.Add(vertex);
                }
            }
        } else {
            if (vertex.Distance > maxDistance) {
                free.Clear();
                free.Add(vertex);
                maxDistance = vertex.Distance;

            } else if (vertex.Distance == maxDistance) {
                free.Add(vertex);
            }
        }
    }
    if(free.Count != 0) {
        return this.GetRandomNeighbour(free);
    } else if(inPath.Count != 0) {
        return this.RandomElement(inPath);
    } else if (onBorder.Count != 0) {
        return this.RandomElement(onBorder);
    }else {
        return null;
    }
}
```

3.1.4 Smooth Border Heuristics

The border of the surface should be as smooth as possible. Until now this has not been addressed by the path algorithm. A path, generated by the current algorithm, will have a zigzag border which poorly reflects the real surface border. Two changes to the `GetRandomNieghbour` method is introduced to make the path follow the border.

The first technique is to choose the vertices with the lowest degree. The idea behind this is that vertices on the border generally have lower degrees than the interior vertices, thus forcing the path towards the border. It also leaves high degree vertices until later in the path construction. Having high degree vertices in the later stages of the path construction gives more possibilities,

thus enhancing the success rate of the algorithm. The second technique is to prioritize border vertices over other vertices. In Figure 3.3 the path finding algorithm is illustrated.

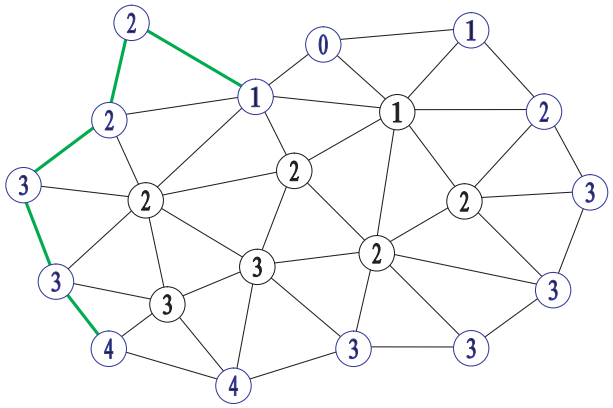


Figure 3.3: The path finding algorithm starting from the vertex with distance one. The path is constructed by moving in the direction of greatest distance. Secondly the vertex with lowest degree is chosen. The blue colored vertices is the border vertices.

Implementation

The implementation of the minimum degree heuristic can be seen in Algorithm 3.5. Each vertex, from the set of adjacent vertices with largest distance (`neighbours`), is added to a list if their degree is equal to the current minimum degree. If a vertex has a degree that is lower than the current minimum degree the list is cleared and the vertex is inserted in the empty list. A random vertex in the list is returned.

Algorithm 3.5: The modified random neighbor selection of the Posá algorithm.

```
Vertex GetRandomNeighbour(List<Vertex> neighbours) {
    // ... //
    double minDegree = neighbours[0].Degree;
    double tempDegree = minDegree;
    list.Add(neighbours[0]);
    for (int i = 1; i < neighbours.Count; i++) {
        tempDegree = neighbours[i].Degree;
        if (tempDegree == minDegree) {
            list.Add(neighbours[i]);
        } else if (tempDegree < minDegree) {
            list.Clear();
            minDegree = tempDegree;
            list.Add(neighbours[i]);
        }// ... //}
    return this.RandomElement(list);
}
```

In Algorithm 3.4 the border following is shown. It is only used when picking a random pivot vertex for the rotational transform.

3.2 Angle Minimizing

A problem with the Posá algorithm and with the extended version developed in this thesis is that it does not produce smooth paths. The previous heuristics tried to eliminate this problem by improving the vertex selection step. Another approach is to calculate some kind of path energy and then minimize this energy.

Minimizing this function can be done in different ways, for example by using some kind of genetic algorithm. This requires a method for constructing a Hamiltonian path from two existing Hamiltonian paths. This was deemed too hard a problem to solve so a simpler approach was pursued. The approach is to generate a couple of paths and pick the path with lowest energy.

Designing the energy function is the most important step. The goal is to minimize frequent turns, so called zigzag patterns, in the path and promote long and straight path segments. A suitable energy function can be seen in (3.2). It heavily penalize turning by having a quadratic factor of the turning angle at the current vertex. The function further penalize turning by having a trailing factor from the previous turning angle.

$$E = \sum_{i=1}^{n-1} (\alpha_i^2 + \alpha_{i-1}) \tag{3.2}$$

$$\alpha_i = \arccos \frac{\mathbf{u}_\mathrm{i} \mathbf{u}_{\mathrm{i}+1}}{|\mathbf{u}_\mathrm{i}||\mathbf{u}_{\mathrm{i}+1}|} \tag{3.3}$$

$$\mathbf{u}_\mathrm{i} = \underline{p}_{i-1} - \underline{p}_i \tag{3.4}$$

In Figure 3.4 the indexing and the defined properties used in (3.2) can be seen.

Implementation

The minimization scheme generates a sequence of random paths. Each time a path is generated it is compared to the current path with lowest energy. Each time a new path has lower energy than the current path the current path is updated. The algorithm terminates after it generates a specified number of

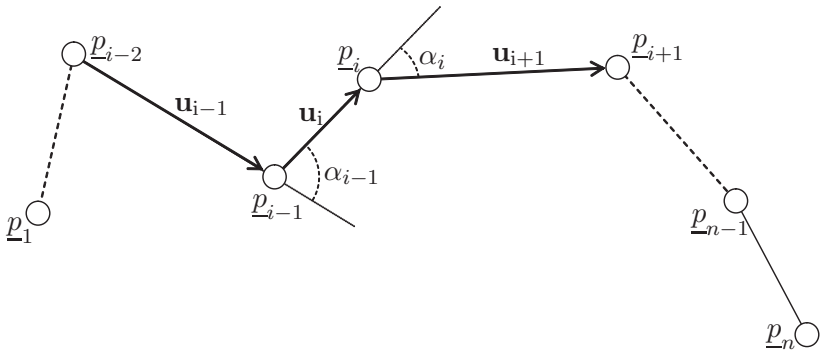


Figure 3.4: Showing a path and the numbering of the components used in the energy function.

paths in sequence which does not result in an update of the current path. The implementation can be seen in Algorithm 3.6.

Algorithm 3.6: The path generation method showing the energy minimizing scheme.

```
GrindingPath GeneratePath(Mesh mesh, int generations, long timeout) {
    // ... //
    while ((path = finder.GeneratePath()) == null) { }
    double energy = path.Energy;
    while (generation++ < generations) {
        temp = finder.GeneratePath();
        if (temp.Energy < energy) {
            energy = temp.Energy;
            path = temp;
            generation = 0;
        }
    }
    return path;
}
```

The energy function is computed according to (3.2). The implementation can be seen in Algorithm 3.7. Note that special care has to be taken before passing a value to the `Acos` method, this is due to roundoff errors.

Algorithm 3.7: Implementation of the energy function.

```
private void CalculateEnergy() {
    // ... //
    for (int i = 1; i < p.Count - 1; i++) {
        Vector u = p[i].Point - p[i - 1].Point;
        Vector v = p[i + 1].Point - p[i].Point;
        if (u.Length == 0 || v.Length == 0) {
            continue;
        }

        double an = (u * v) / (u.Length * v.Length);
        an = an > 1.0 ? 1.0 : an;
        an = an < 0.0 ? 0.0 : an;
        current = Math.Acos(an);
        this.energy += current * current + last;
        last = current;
    }
}
```

3.3 Timeout Criteria

In Section 3.1.3 it was mentioned that the algorithm could get stuck when there was no Hamiltonian path in the grid graph. It was also stated that it could seem like it got stuck when there only exists few Hamiltonian paths, due to the long execution time. In that section the distance transform was introduced to minimize the impact of the latter problem of to few Hamiltonian paths. In this section another approach to the second problem is introduced but it also handles the case when there is no Hamiltonian path in the grid graph.

The approach is to introduce a timeout criteria. After a specified amount of time the algorithm will terminate and restart, if it has not found a Hamiltonian path. Restarting gives the algorithm a new start vertex thus increasing the probability of finding a path.

If there is no path this approach would not be enough to handle the problem. A second criterion is introduced to handle the case where no paths exist. It simply states that if a fixed amount of restarts has not generated a path then the algorithm is terminated. When this happens the meshing has to be redone with different parameters. In the worst case the point cloud has to be resampled.

Implementation

The timeout functionality is implemented as a timer that raises an interrupt whenever the specified time limit is reached. The time limit is either set to a user defined value or the execution time of the meshing step. Setting it to the execution time of the meshing step is motivated by that the time complexity of the meshing step is of the same order as the average time complexity of the path finding algorithm. The `GeneratePath` method with timeout can be seen in Algorithm 3.8.

Algorithm 3.8: The path generation method showing the timeout implementation.

```
public GrindingPath GeneratePath() {
    // ... //
    this.interrupted = true;
    while (this.interrupted) {
        this.Reset();
        this.interrupted = false;
        using (new Timer(this.Interrupt, this, this.timeout, Timeout.
            Infinite)) {
            // ... //
            do {
                Vertex neighbour = this.GetRandomNeighbour(path, path.
                    End);
                // ... //
            } while (path.Count < this.mesh.NumberOfVertexes && !this.
                interrupted);
            // ... //
        }
    }
    // ... //
    return path;
}

private void Interrupt(object finder) {
    this.interrupted = true;
}
```

3.4 Conclusions

During simulations and real tests the path planning algorithm shows good results. The algorithm is able to generate a path even when the mesh quality is poor e.g. contain many holes. It is very fast, considering the number of vertices in the grid mesh. The execution time is around ten seconds for a grid with 500 vertices.

The path planning algorithm has not been tested for larger graphs than a few hundred vertices. If larger graphs are required it could be of interest to consider using another algorithm, with documented performance for larger graphs. Several algorithms for finding hamiltonian paths are covered in [9].

In [2] an algorithm designed for triangular grids is presented. It is a linear time heuristic algorithm that shows some promising results.

The angle minimizing algorithm reduces the number of turns in the general case. It is dependent on the quality of the generated paths which means that if only paths with high energy are generated then the angle minimizing will return a path with many turns.

It is not known whether the border following heuristics contribute significantly to the result.

Chapter 4

Intuitive Lead-Through Programming

The previous chapters have described the algorithm developed in this thesis. This chapter will introduce the Lead-Through Programming concept and the integration of the algorithm into the Lead-Through server developed at ABB Corporate Research Germany. A detailed description of the server can be found in [6].

The chapter begins with a brief introduction to the Lead-Through Programming and the Lead-Through server. This is followed by a description of the instructions developed for the server. This is then followed by an explanation of the system setup and finally the conclusions are presented.

4.1 Lead-Through Programming

Lead-Through Programming is a concept intended to make the teaching of a robot simple. The programming is done by teaching the robot a task by manually guiding it through each step of the task. A taught task is then executed when needed. Each task has an associated instruction set that is used to structure the programming. These instructions are called intuitive instructions and should be easy to use, by an ordinary worker.

Physically guiding the robot through the teaching step is accomplished by the use of force-torque-sensors. This makes it possible to reorient and move the robot as an ordinary tool. A task suitable guiding device should be used during teaching. In this thesis this means that the guiding device enables the user to access all surfaces on a casting.

Communication with the system is an important aspect of the Lead-Through Programming concept. It can be done in many different ways, see [3] for a detailed analysis. The report concludes that voice commands should be used. It is also suggested that a PDA or a Flex Pendant[1] could be used for hot editing a program.

Safety is paramount to the Lead-Through Programming concept. It has been treated extensively in both [6] and [3]. Safety issues have not been treated extensively in this thesis, although some basic guidelines have influenced the work. The algorithm was developed with the requirement that differences in logic and behavior of computers and humans should not lead to dangerous behavior of the robot. It should thus be possible for a worker to predict the outcome of any action.

At this point it is relevant to introduce a Developer-Consultant-Operator model. It describes the three different actors in the Lead-Through process. The first actor, the developer, is concerned with the development of the instructions. The second is the consultant who has expert knowledge of the task and knows how the template instructions are configured. The operator is the intended user.

4.1.1 Lead-Through Server

The Lead-Through server is a XML based server running on Windows XP©. It is possible to save the current states of the server. Starting the server will load the previously saved states. This thesis is concerned with two parts of the server: the template instructions and the instruction modes.

[1] The Flex Pendant is a robot remote control unit used by ABB.

A template instruction is a class describing a type of action for example *set speed*. The template instructions are the domain of the developer, in the Developer-Consultant-Operator model. An intuitive instruction, which was previously mentioned, is an instantiation of a template instruction. These intuitive instructions have usually one or more parameters that need to be specified. An intuitive instruction modeled on the set speed template instruction could for example be *set slow speed*. The modes are the intuitive instruction sets specific to a task.

The different modes are constructed by the consultant. The consultant uses the server's user interface to build the modes. It is done by dragging the appropriate template instructions to a mode and specifies their parameters. The interface can be seen in Appendix B.

Execution of the instructions is done by the operator from the client. The operator communicates with the client with voice commands. The name of an intuitive instruction is also its associated voice command. When finishing a teaching step a *compile* instruction should be issued. This instruction creates a RAPID[2] program from the instructions and the variables stored in the server. The generated program is then run on the robot by issuing a *play* instruction.

The server context is a space where global data are stored. An instruction can have several dependencies which are variables that need to be in the context before the instruction can be executed. These variables can either be specified by the consultant directly in the context or by issuing an instruction which sets the necessary variables.

A detailed explanation of the client-server architecture and implementation can be found in [6].

4.2 The Instruction Set

This section will describe the three template instructions developed for the grinding task. The `RecordSurface` and `StopRecordSurface` instructions will be presented first, followed by the `GeneratePath`instruction.

4.2.1 Surface Recording

The two instructions concerned with surface recording are the `RecordSurface` and `StopRecordSurface` instructions. Between the invocations of these two

[2]RAPID is the robot programming language used by ABB.

instructions, which have to be invoked in series, the robot position data, is recorded.

To be able to issue the `RecordSurface` instruction the server context need to contain a variable with the sampling interval, which is called `Grinding.Interval`. The instruction starts a timer that each interval stores the position data in a context variable called `Grinding.Surface`. The timer belongs to a context variable called, `Grinding.Timer`. To guarantee that the instructions are executed in sequence the `StopRecordSurface` is dependent on the `Grinding.Timer` variable.

In Figure 4.1 a simple flowchart of the instruction execution is displayed, it also includes the path generation instruction.

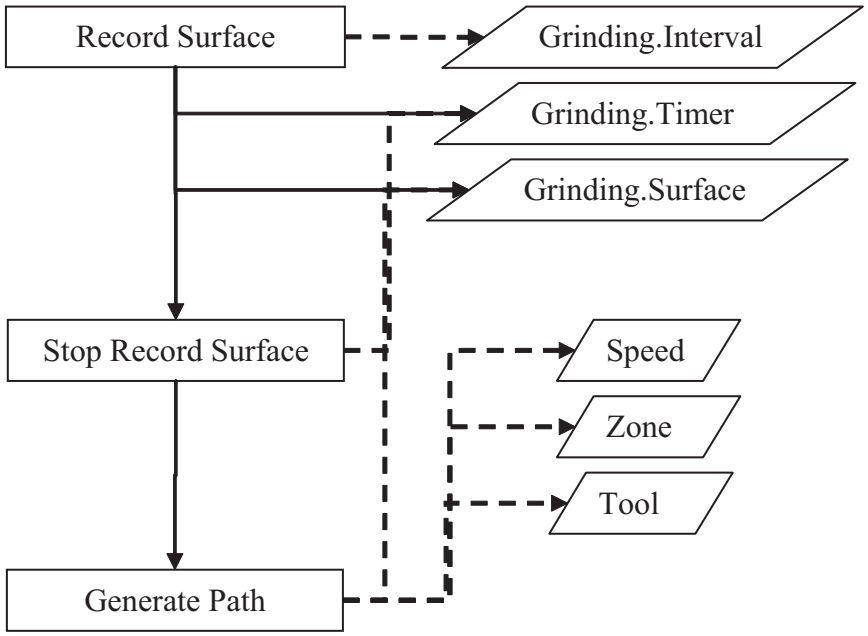


Figure 4.1: The flowchart describe the calling order of the grinding instructions. A dotted line marks a dependency. A solid line is either showing the creation of a variable or the order of the instruction calls. A rectangle represents an instruction a skew rectangle represents a variable.

4.2.2 Path Generation

The `GeneratePath` template instruction contains the algorithm described in the previous chapters. All steps from surface reconstruction to path planning is executed when this instruction is called. It is dependant on the context variable `Grinding.Surface`. The instruction also requires that the robot

speed, zone data[3] and tool data are specified.

Several parameters influence the instruction and they are summarized below.

MaximumPointDistance The neighborhood radius in the normal estimation step.

MinimumPointDistance The reduction radius in the normal estimation step. A point that is closer to another point than this value is discarded.

MaximumEdgeLength The neighborhood radius in the meshing step. Restricts the edge length and influences the curvature of the mesh border.

MinimumEdgeLength The reduction radius in the meshing step. A point that is closer to another point than this value is discarded. An edge can not be smaller than this value.

DiscRadius This is the length of the radius of the contact area of the grinding disc. During optimizing edges that are longer than this minimum length are split by the optimization algorithm.

Timeout The time limit of finding a path. A restart is made when the limit is reached.

Generations The number of random paths that will be generated before the angle minimizing is terminated.

Optimize Tells the algorithm to optimize the mesh or not.

RepeatNumber Number of times the robot should follow the path.

NormalShift The distance between each shift layer.

NubmerOfShiftLayers Number of normal shifts that will be generated.

The last three parameters govern some functionality that has not been covered before. It was mentioned in Chapter 1 that the distribution of material which is to be removed was uneven. An approach to this problem is to sweep repeated times over the surface, gradually descending towards the desired level. By shifting the path along the surface normal of its vertices this can be accomplished.

4.3 System Setup

The server is run on an ordinary PC running Windows XP©. It is connected to an industrial robot, the ABB IRB140, through the Lead-Through server.

[3]The zone data is a radius defining a sphere around each point. If a robot has been inside a points sphere it is considered to have visited the point.

Manual guidance is used together with the grinding instruction mode to teach the robot to grind a surface.

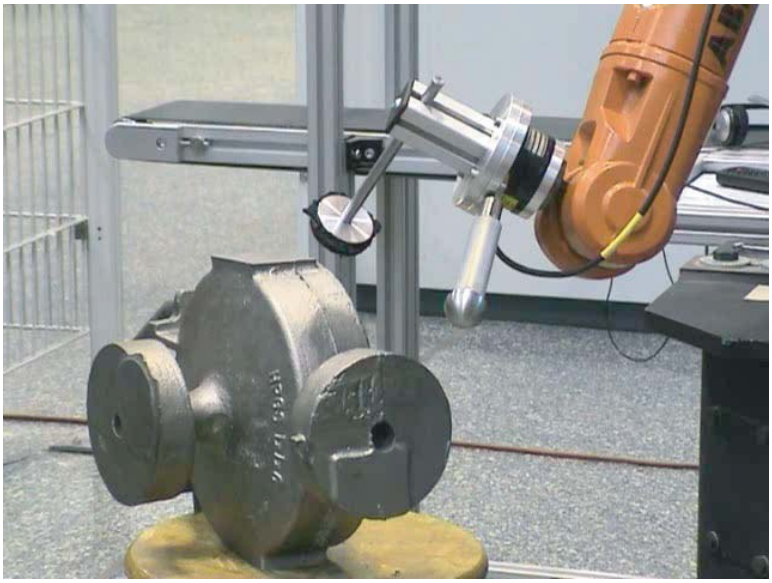


Figure 4.2: This picture shows the metal casting and the robot. The robot has the cleaning tool and guiding device attached.

Instead of a grinder a soft cleaning tool is used. Its properties differ significantly from a grinder. It is elastic in the tip, it has a constant contact area and it does not remove any material. It was chosen as a substitute because the IRB140 is too weak to handle a grinder. The cleaner is sufficient to demonstrate that the algorithm works.

In Figure 4.2 the robot can be seen hovering above the target metal casting. The grinding tool is attached to a simple guiding device on the robot flange[4].

Communication with the robot is either done by voice commands or by directly controlling the client from the computer.

4.3.1 Grinding Mode

The grinding mode is made from some basic instructions and those described in Section 4.2. The different instructions are described below. Those instructions that are not covered in this thesis are described in [6].

DefaultValue A collection of instructions that sets the speed, zone and tool data of the grinding process.

[4]The flange is the tip of the robot arm.

MainMode Instruction that switches to the main instruction mode.

Stop Instruction that stops the manual guidance of the robot.

Start Instruction that starts the manual guidance of the robot.

LoadProgram A collection of instructions that compiles the taught program and loads it into the robot.

Replay Instruction that executes the previously loaded program on the robot.

StopRecord Instruction that stops the sampling of surface points.

Record Instruction that starts the sampling of surface points.

GeneratePath Instruction that reconstructs the surface and generates a path.

CompileLog Instruction that compiles the taught program without loading it to the robot.

OptimizePath Instruction that reconstructs the surface, optimizes it and generates a path.

Move Instruction that marks a position the robot has to visit.

The graphical user interface of the server can be seen in Appendix B. It shows the grinding mode and the properties of the `GeneratePath` instruction.

4.4 Conclusions

A known issue, with the server, is that it is not possible to define the tool in the point recording and the path execution steps, with one instruction. The problem is that the tool, used when recording the points, can not be set from the server. The solution is to manually load the correct tool before record time to the robot. At execution time the same tool data is loaded into the program by an instruction.

To tune the parameters of a `GeneratePath` instruction the console version of the grinding algorithm is used, see Appendix D for the user manual. The `robtargets` in the `Grinding.Surface` variable are copied to a text file and used as input to the console application.

The console version generates the same mesh and path as the server instruction. The output from the console program are two files that contains the path and mesh. These files can be loaded in Matlab© making it possible to have a visual feedback of the surface mesh and the grinding path showing the result of different parameter settings.

Future development should consider integrating visual feedback into the Lead-Through system. An integrated visual feedback would make the tuning process much simpler and save the effort of going through Matlab© every time a new grinding instruction is configured. This would simplify the task of the consultant. It would also help the operator in choosing the correct configured grinding instruction.

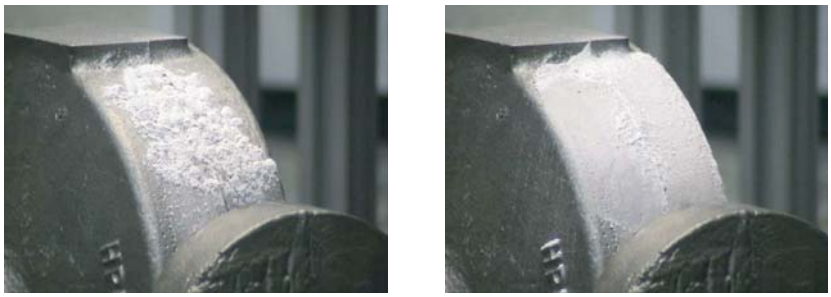
Chapter 5

Results and Conclusions

The algorithm has been tested on the system described in the previous chapter. Some promising results have been obtained and will be presented in this chapter. The chapter is concluded with some conclusions of the overall performance of the algorithm and suggestions of the direction of future work.

5.1 Results

The results of the algorithm are good. It is able to generate a path that covers a whole surface, this can be seen in Figure 5.1. In the figures a powder is used to show that the cleaner covers the whole surface.



(a) Before grinding. (b) After grinding.

Figure 5.1: The images shows the surface before and after the cleaning tool is used on it. The intention is to show that the cleaner is able to cover the whole surface.

5.2 Conclusions

The results demonstrate that the approach in this thesis works. The goals defined in the beginning are accomplished and a lot of experience with the grinding problem has been gained. The algorithm developed in this thesis forms a good basis for future development.

Although the promising results there are some unsolved issues and problems with the algorithm and the server. One issue is that the robot reorients the tool more than 180° thus forcing an emergency stop. The origin of this problem is unknown but possible sources are the use of different tools during teaching and path execution. It is also possible that the axis configuration of the robot position is corrupted during one of the steps of the algorithm.

Another orientation problem is originating from the construction of the center vertex in the edge split operation. Interpolating the position data and the tool orientation is simple this is not the case with the axis configuration, which needs special treatment. If the endpoints of the edge have different axis configuration it is not possible to determine the correct axis configuration of the center vertex.

Because of this problem the optimization is currently not working properly. If all points in the reconstructed surface have the same axis configuration

then the optimization works. It is possible to calculate the axis configuration from the joint angles. These are currently not available in the algorithm.

When calibrating the tool it is important to define the z-axis of the tool so that it is directed out from the surface. If the z-axis is pointing into the surface problems will arise when shifting the path. This is because the sign of the surface normal is determined by the tool orientation. A surface normal that points in the wrong direction will result in a path shift that starts inside the surface and gradually moves upward.

It was reported in the meshing chapter that the meshing creates invalid triangles and holes. This can be seen in Figure 5.2 and Figure 5.3. It was stated that this affects the coverage of the surface. It also affects the success of the path planner. A poorly constructed mesh will result in many holes and so called cut vertices which makes it impossible to find a Hamiltonian path in the mesh. A cut vertex is a vertex that splits a graph into two unconnected parts, if it is removed.

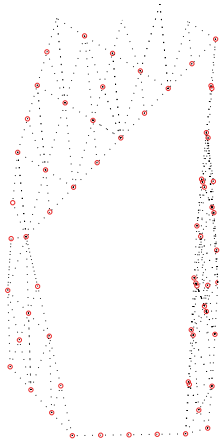


Figure 5.2: An optimized mesh featuring a hole.

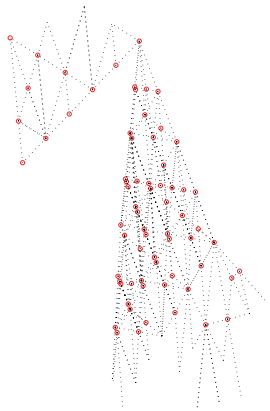


Figure 5.3: An optimized mesh showing the effect of splitting illegal faces.

The distribution of cut vertices indicates the possibility of finding a Hamiltonian path in a graph. A future algorithm should feature some way of detecting the distribution of cut vertices and force a remesh if it is deemed impossible to find a Hamiltonian path. The existence of cut vertices also indicates that there are holes in the mesh.

Appendix A

Eigenvalue Estimation

The source code of the eigenvalue and eigenvector calculations, used in the normal estimation step, is presented in this appendix.

```
Vector Eigen(double[,] a) {
    int n = a.GetLength(0);
    int i, j, iq, ip, nrot;
    double tresh = 0.0, theta = 0.0, tau = 0.0;
    double t = 0.0, sm = 0.0, s = 0.0, h = 0.0, g = 0.0, c = 0.0;
    double[] b = new double[n], z = new double[n];
    double[] d = new double[n];
    double[,] v = new double[n, n];

    for (i = 0; i < n; i++) {
        for (j = i; j < n; j++) {
            v[i, j] = 0.0;
            v[j, i] = 0.0;
        }
        v[i, i] = 1.0;
        b[i] = d[i] = a[i, i];
        z[i] = 0.0;
    }

    nrot = 0;
    for (i = 0; i < 50; i++) {
        sm = 0.0;
        for (ip = 0; ip < n - 1; ip++) {
            for (iq = ip + 1; iq < n; iq++) {
                sm += Math.Abs(a[ip, iq]);
            }
        }
        if (i > 40 && sm == 0.0) {
            // the eigenvalues and eigenvectors are passed to a
                function that sorts them and returns the eigenvector
                corresponding to the smallest eigenvalue.
            return this.Normal(d, v);
        }

        if (i < 3) {
            tresh = 0.2 * sm / (n * n);
        } else {
            tresh = 0.0;
```

```
        }

        for (ip = 0; ip < n - 1; ip++) {
            for (iq = ip + 1; iq < n; iq++) {
                g = 100.0 * Math.Abs(a[ip, iq]);

                if (i > 3 && (Math.Abs(d[ip]) + g) == Math.Abs(d[ip])
                    &&
                   (Math.Abs(d[iq]) + g) == Math.Abs(d[iq])) {
                    a[ip, iq] = 0.0;
                } else if (Math.Abs(a[ip, iq]) > tresh) {
                    h = d[iq] - d[ip];

                    if ((Math.Abs(h) + g) == Math.Abs(h)) {
                        t = a[ip, iq] / h;
                    } else {
                        theta = 0.5 * h / a[ip, iq];
                        t = 1.0 / (Math.Abs(theta) + Math.Sqrt(1.0 +
                            theta * theta));
                        if (theta < 0.0) { t = -t; }
                    }

                    c = 1.0 / Math.Sqrt(1.0 + t * t);
                    s = t * c;
                    tau = s / (1.0 + c);
                    h = t * a[ip, iq];
                    z[ip] -= h;
                    z[iq] += h;
                    d[ip] -= h;
                    d[iq] += h;
                    a[ip, iq] = 0.0;

                    for (j = 0; j <= ip - 1; j++) {
                        g = a[j, ip];
                        h = a[j, iq];
                        a[j, ip] = g - s * (h + g * tau);
                        a[j, iq] = h + s * (g - h * tau);
                    }

                    for (j = ip + 1; j <= iq - 1; j++) {
                        g = a[ip, j];
                        h = a[j, iq];
                        a[ip, j] = g - s * (h + g * tau);
                        a[j, iq] = h + s * (g - h * tau);
                    }

                    for (j = iq + 1; j < n; j++) {
                        g = a[ip, j];
                        h = a[iq, j];
                        a[ip, j] = g - s * (h + g * tau);
                        a[iq, j] = h + s * (g - h * tau);
                    }

                    for (j = 0; j < n; j++) {
                        g = v[j, ip];
                        h = v[j, iq];
                        v[j, ip] = g - s * (h + g * tau);
                        v[j, iq] = h + s * (g - h * tau);
                    }
                    nrot++;
                }
            }
```

```
        }
        for (ip = 0; ip < n; ip++) {
            b[ip] += z[ip];
            d[ip] = b[ip];
            z[ip] = 0.0;
        }
    }
    throw new Exception("Too many iterations in routine Eigen!");
}
```

Appendix B

The Lead-Through Server GUI

This appendix contains an image of the Lead-Through servers graphical user interface.

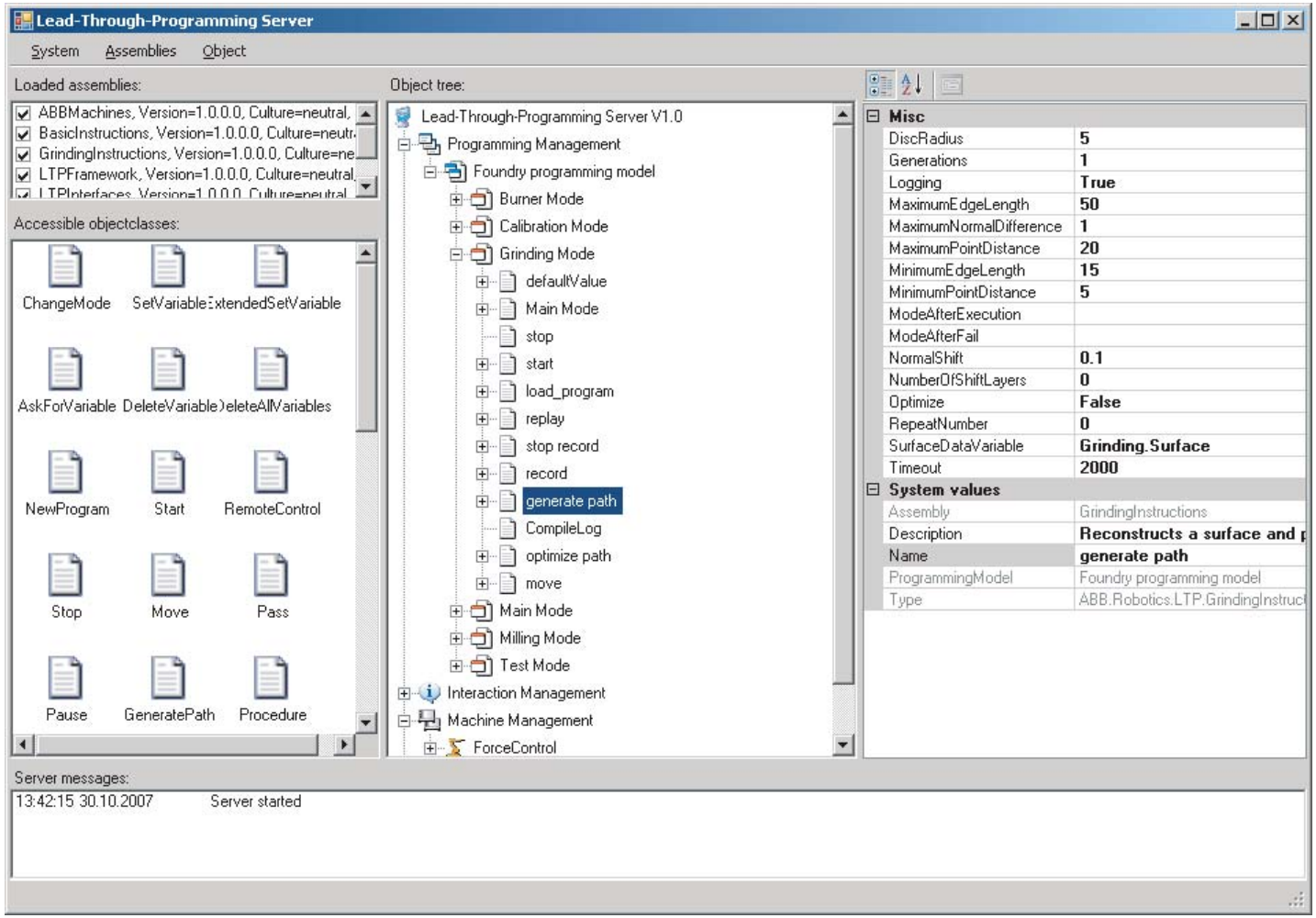


Figure B.1: The Lead-Through server window. The left lower box shows the template instructions. The middle box show, among other things, the instruction modes. The grinding mode is expanded. The left box shows the current selected instructions properties.

Appendix C

Robtarget

The `robtarget` is a data structure used by the RAPID language. RAPID is the robot language used by ABB. It is used to represent a robot position in cartesian coordinates and its tool orientation. Additional data concerning the axis configuration is also provided which is important to identify the position in the robot work space.

```
robtarget := [position, orientation, axisconf, extaxisconf]
```

The different elements of the `robtarget` will be explained further.

position The robot position in cartesian coordinates.

orientation The robot tool orientation expressed as a quaternion.

axisconf The current axis configuration of the robot.

extaxisconf The configuration of some external axis.

Appendix D

Grinding Algorithm: User Manual

This user manual gives a short introduction of how to use the executable `ABBGrinding.exe`. It is the executable version of the grinding library `ABBGrinding.dll`. The program accepts two optional flags and a file containing predefined parameters and a file containing robtargets.

Program Execution

Running the program is done by typing the following in the console.

```
.\LTPGrinding.exe [-opt] [-path] <paramterfile> <pointdata file>
```

Running the program without any flags results in a reconstructed mesh from the points in the point data file. Adding the `-opt` tells the program to also generate an optimized mesh. The `-path` option turns on the path planner which generates a path for the non optimized mesh and the optimized, if the `-opt` flag is specified.

During the program execution there will be printed some information to the console, telling which step is currently executing and how long the last step took to run. When running the path planner there will be printed some additional information, concerning the timeout and restarting of the path planner.

File Formats

The parameter file and point data file should be formatted as pure text. The parameter file must contain the following lines with specified values, the order of the lines is not important.

```
SELECT_NEIGHBORHOOD_RADIUS=<double value>
SELECT_REDUCTION_RADIUS=<double value>
MESH_NEIGHBORHOOD_RADIUS=<double value>
MESH_REDUCTION_RADIUS=<double value>
MESH_ANGLE_DIFFERENCE_TRESHOLD=<double value>
MESH_OPTIMIZATION_MAX_EDGE_LENGTH=<double value>
```

Each line in the point data file must contain one `robtarget`. Each `robtarget` should match the following regular expression or it wont be correctly parsed by the program.

```
robtarget = @".*\[␣*[.*\],\[.*\],\[.*\],\[.*\]␣*\].*";
```

Output

The program generates one or two files, two if `-opt` is set. They are named `mesh_<surfacedata>.m` and `optmesh_<surfacedata>.m`. Each file contains the points, edges and paths of the non optimized and the optimized meshes and some additional information for debugging purposes.

The files contains the following variables with specified row formatting. The names corresponds to the vertex attributes with same name, id is a scalar, point, orientation and normal are three dimensional vectors.

```
Mesh_Points : {[<id>] [<point>] [<orientation>] [<normal>]}
Mesh_Edges : [<id1> <id2>]
Mesh_HalfEdges : [<id1> <id2>]
Mesh_Faces : [<id1> <id2> <id3> <point1> <point2> <point3><]
Path<Number> : [<id>]
PathExt<Number> : [<point>]
```

Matlab Scripts

Reading the output can be done with the Matlab©-script `plotsurface.m`. Typing `help plotsurface` at the command line will generate the following printout. It describes the use of the function togehter with the variables in the generated Matlab files.

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

  PLOTSURFACE plots data from the grinding algorithm "ABBGrinding".
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

  PLOTSURFACE(Edges, Points, Path, varargin) plots the edges, point
      data and
  path according to the flags in varargin.

  The input variables must have the following format.
     Edges          n x 2 matrix containning vertex id:s.
     Points         n x 10 cell array with the following formatting.
                    {[id] [point (1 x 3)] [normal (1 x 3)] [orient (1
                        x 3)]}
     Path           1 x n array containing vertex id:s OR n x 3 matrix
                    containing coordinate points. When the latter is
                        used
                    the ExtPath flag must be specified.

  The following flags are available.
     PointColor     followed by a color string e.g. 'r' sets the color
          of the
                    points.
     NoEdges        hides the edges.
     Degree         displays the degree of each vertex.
     NoPoints       hides the point data.
     VertexNbr      displays the id of the vertex.
     Normals        displays the normals of the vertices.
     Orientations   displays the tool orientation at the vertices.
     NoPath         hides the path.
     PathNbr        displays the path nodes id:s.
     SoftPath       displays the path with smooth angles (second order
          Bezier
                    curves).
     ExtPath        tells the algorithm that Path is an extended path.

     It is also possible to supply a string containing plot
         formatting
     characters. This affect the edges appearance.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

Bibliography

[1] T. Bodenmueller and G. Hirzinger. Online surface reconstruction from unorganized 3D-Points for the DLR hand-guided scanner system. In *3DPVT '04: Proceedings of the 3D Data Processing, Visualization, and Transmission, 2nd International Symposium on (3DPVT'04)*, pages 285–292, Washington, DC, USA, 2004. IEEE Computer Society.

[2] U. Dogrusöz and M. S. Krishnamoorthy. Hamiltonian cycle problem for triangle graphs. Technical report, Rensselaer Polytechnic Institute, Troy, NY 12180, USA, 1995.

[3] Chen Fei. Design of a safe operator communication interface for lead through programming of industrial robots. Technical report, Hamburg University of Technology, 2007.

[4] William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery. *Numerical Recipes in C: The Art of Scientific Computing*. Cambridge University Press, New York, NY, USA, 1992.

[5] L. Pósa. Hamiltonian circuits in random graphs. *Discrete Mathematics*, 14:359–364, 1976.

[6] Sanparit Puanaiyaka. Intuitive lead-through-programming of robots considering feeder removing of injection-formed plastic parts as example. Master's thesis, Hamburg University of Technology, 2007.

[7] H. Samet. *Applications of spatial data structures: Computer graphics, image processing, and GIS*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1990.

[8] V. Surazhsky and C. Gotsman. Explicit surface remeshing. The Eurographics Association, 2003.

[9] B. Vandegriend. Finding hamiltonian cycles: Algorithms, graphs and performance. Master's thesis, University of Alberta, 1998.

[10] Weining Yue, Qingwei Guo, Jie Zhang, and Guoping Wang. 3D triangular mesh optimization in geometry processing for CAD. In *SPM*

'07: Proceedings of the 2007 ACM symposium on Solid and physical modeling, pages 23–33, New York, NY, USA, 2007. ACM Press.

[11] A. Zelinsky, R. Jarvis, J. Byrne, and S. Yuta. Planning paths of complete coverage of an unstructured environment by a mobile robot, 1993.