Universität Hamburg
DER FORSCHUNG | DER LEHRE | DER BILDUNG

FACULTY
OF MATHEMATICS, INFORMATICS AND NATURAL SCIENCES

Bachelor's Thesis in Informatics

Conformal Surface Printing

on a 5-Axis 3D Printing System

Paul Bartel

DEPARTMENT
OF INFORMATICS

First Reviewer:	Dr. Florens Wasserfall
Second Reviewer:	Daniel Ahlers
Submission Date:	August 22, 2024
Program:	Informatics
Enrollment Number:	7428530

Abstract

5-axis 3D printing systems have a significant advantage over conventional 3-axis systems, as they can keep the extruder conforming to the object's surface. This greater freedom in extruder positioning theoretically allows for a multitude of new possibilities in 3D printing, such as printing overhangs without support structures or printing of non-planar layers, to mitigate stair-stepping. However, unlike 3-axis systems, general software support for 5-axis tool-path generation is still in its infancy.

This thesis aims to leverage the capabilities of a 5-axis system to conformally print a filled, user-defined user-defined freeform layer on the surfaces of printed parts. To achieve this Neo5x, an early-stage 5-axis slicer plugin for the CAD software Rhino, will be extended to support this functionality. The plugin can generate 5-axis tool-paths using an ordered list of points and surface normals as input.

The proposed solution will provide the necessary input by generating a space-filling pattern in a 2-dimensional plane, which is then projected onto the selected surface area. Due to the inevitable inaccuracies in the projection process a projection error is introduced, which causes the distance between curves to increase, depending on the curvature of the surface. This can result in tool-paths that does not fill the bounded area like intended. To reduce the effects of the projection error, a simple post-processing compensation is introduced, that iteratively adjusts the extrusion widths of the pattern to fill any gaps that may have formed.

Contents

Abstract i

Contents i

List of Figures iii

1 Introduction 1
- 1.1 Problem Statement . . . 1
- 1.2 Goal of this Thesis . . . 2
- 1.3 Outline . . . 2

2 Fundamentals 3
- 2.1 General 3D printing process . . . 3
  - 2.1.1 Modeling . . . 3
  - 2.1.2 Slicing . . . 3
  - 2.1.3 Printing . . . 3
- 2.2 Rhino . . . 4
- 2.3 Neo5x . . . 4
- 2.4 5-Axis Hardware . . . 6

3 Related Works 7
- 3.1 Conformal pattern printing . . . 7
- 3.2 Open5x . . . 7
- 3.3 s^3-Slicer . . . 7

4 Implementation 9
- 4.1 Neo5x Interface . . . 9
- 4.2 Implementation Preliminary . . . 11
  - 4.2.1 General Approach . . . 11
  - 4.2.2 3D Offsetting Along Surface . . . 12
  - 4.2.3 Projection on surface . . . 13
  - 4.2.4 Requirements . . . 14
- 4.3 Implementation Details . . . 15
  - 4.3.1 Projection to Plane . . . 15
  - 4.3.2 Pattern Generation in 2-dimensional Plane . . . 17
    - 4.3.2.1 Data Structures . . . 17
    - 4.3.2.2 Contour-Parallel Pattern . . . 19
    - 4.3.2.3 Zig-Zag Pattern . . . 20
  - 4.3.3 Projection back to Surface . . . 22
  - 4.3.4 Generation of Tool-Path Objects . . . 23
  - 4.3.5 Projection-Error Compensation . . . 24
  - 4.3.6 Compensation Approach . . . 25

5 Evaluation 28
- 5.1 Print Results . . . 28
  - 5.1.1 First Test Prints . . . 28

5.1.2 Projection Error & Compensation 30
5.2 Software Limitations 33
5.2.1 Surface Area Delimitation 33
5.2.2 Projection Approach 34
5.2.3 Compensation 35

6 Conclusion **37**
6.1 Future work 37

Bibliography **39**

List of Figures

Figure 1.1 3-axis vs 5-axis movement 1

Figure 2.1 Rhino Interface 4
Figure 2.2 Neo5x in Rhino 5
Figure 2.3 5-Axis hardware 6

Figure 4.1 Creation process of CAM-Blocks 9
Figure 4.2 General idea of the implementation 11
Figure 4.3 Self-intersection with 3D offsetting along surface 12
Figure 4.4 Example of projection error 13
Figure 4.5 Difference of projection angles 15
Figure 4.6 Projection of a closed-curve onto a plane 16
Figure 4.7 Supported pattern 17
Figure 4.8 Visual representation of curve data structure 18
Figure 4.9 Contour-Pattern can from individual islands 19
Figure 4.10 Contour-Parallel pattern 20
Figure 4.11 Zig-Zag pattern split 21
Figure 4.12 Zig-Zag pattern 22
Figure 4.13 Projection of curves back to surface 22
Figure 4.14 Visualizied tool-paths 24
Figure 4.15 Effects of projection error on tool-paths 25
Figure 4.16 Compensation process visualized 26
Figure 4.17 Compensation before and after 27

Figure 5.1 First test print: Contour-Parallel 29
Figure 5.2 First test print: Zig-Zag 29
Figure 5.3 Compensation print comparison 30
Figure 5.4 Side-view of compensation print comparison 31
Figure 5.5 Projection error vs surface slope 32
Figure 5.6 Surface delimitation with multiple boundary curves 33
Figure 5.7 Extrem projection error 34
Figure 5.8 Subdividing boundary curve for projection 35
Figure 5.9 Compensation artifacts 36

Introduction 1

5-axis additive manufacturing has gained significant interest due to its potential of overcoming the limitations of conventional 3-axis systems. A 3-axis system is simple and cost-effective, but is limited in its movement, as the printhead can not conform to the shape of object it is printing. As a result, objects are printed in a series of stacked planar layers along the z-axis that approximate the desired 3D model. This limited movement is reason for the characteristic issues associated with 3-axis systems, such as stair-stepping artifacts, the need for support structures for overhangs and limited mechanical properties.

5-axis systems have the potential to overcome these limitations and enable new possibilities in additive manufacturing, not possible with conventional means. The additional axes enable a higher degree of freedom, allowing the printhead to conform to the object's surface as seen in figure 1.1.

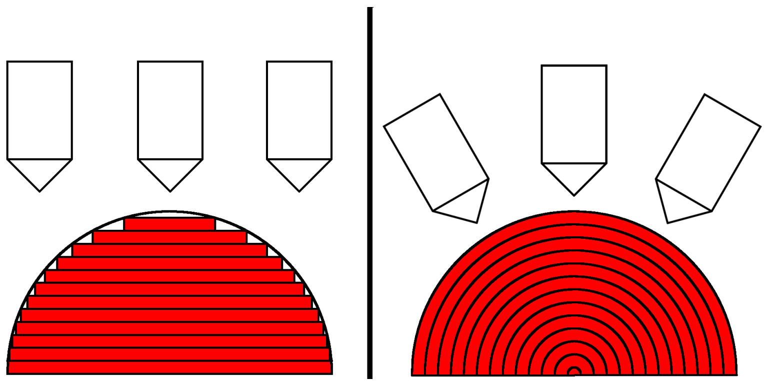


Figure 1.1: 3-axis vs 5-axis movement
The left image shows the movement of a 3-axis system, where the printhead can only be parallel to the build plate. The right image shows the movement of a 5-axis system, where the printhead can freely move around the object.

Making it possible to print non-planar layers, that can mitigate the stair-stepping artifacts [1, 2], which were possible with 3-axis systems [3, 4] but only to a certain degree. Print structures without support structures, as objects can be printed from any angle[2, 5, 6, 7] as well as potentially improve the strengh of printed parts [8].

The added freedom can also be used to print on surfaces of existing objects, which can be useful in printed electronics, where conductive traces may be printed directly onto surfaces [9, 10] or as way to post-process existing parts by adding additional structures.

1.1 Problem Statement

Unlike 3-axis 3D printing, 5-axis systems remain a niche in the current additive manufacturing space, this is due to the high cost and lack of general software support that is intutive to use for the average user. Currently there is no software solution available that allows users to intuitively generate 5-axis tool-paths that can conformally print a filled, user-defined freeform layer on surfaces of printed parts.

1.2 Goal of this Thesis

The main goal of this thesis is to leverage the capabilities of a 5-axis system to enable users to intuitively generate 5-axis tool-paths that can conformally print a filled, user-defined freeform layer on surfaces of printed parts. This feature would allow for new possibilities in additive manufacturing, such as the ability to improve the surface quality of printed parts, by confromally printing over stair-stepping artifacts, or aid in printed electronics where conductive SMD pads can be printed directly onto the surfaces of printed parts.

Since developing a 5-axis slicer from scratch is a monumental task and not feasible within the scope of a bachelor thesis, an early-stage 5-axis slicer plugin for the CAD software Rhino, called Neo5x, will be extended to support this functionality instead. Neo5x directly integrates into Rhino's workflow and is able to generate conformal 5-axis tool-paths, if provided with an appropriate input. Therefore, the proposed solution is mainly concerned with generating the neccessary input data, post-processing of the generated tool-paths and implementing the solution in a way that follows the intended workflow set by the Neo5x developers.

This thesis focuses on the software development aspect of 5-axis additive manufacturing, rather than the hardware side.

1.3 Outline

Chapter 1 introduces the potential of 5-axis additive manufacturing and outlines the goals of this thesis. Chapter 2 provides an overview of the software and hardware used. Chapter 3 showcases related works that have been motivated by the potential of 5-axis additive manufacturing. Chapter 4 provides an in-depth explaination of the proposed solution Chapter 5 present prints made with tool-paths generated by the implementation, as well as discuss current limitations of the implementation. Finally, chapter 6 will conclude this thesis with a summary of the work done and an outlook on future work.

Fundamentals 2

This chapter will provide a brief overview over the software and hardware used in this thesis.

2.1 General 3D printing process

Although 5-axis additive manufacturing lacks general software support, the underlying principles of 3D printing are still applicable. The general 3D printing process can be broken down into three steps: modeling, slicing and printing.

2.1.1 Modeling

In the modeling step, a 3D model is created using Computer-Aided Design (CAD) software. This model represents the object to be printed and is usually saved in a file format like STL, which can then be imported into other software for further processing.

2.1.2 Slicing

Slicing involves converting the 3D model into a series of steps that the printer can execute and resemble the desired object. In conventional 3D printing, this is typically achieved by importing the model into slicer software, which slices the model into a series of 2D layers along the z-axis and generates a toolpath for each layer that the printer executes individually to build the object across the z-axis.

In 5-axis additive manufacturing no single method has yet established itself as the defacto standard. Regardless, the general concept remains the same, the model has to be converted into a series of steps that, when exectued by the printer, will result in the desired object. This however, is more complex in 5-axis systems, as two additional axes have to be taken into account, as well as potential collisions between the printhead and the object.

In either case, the output of the slicing process will be a G-Code file, that contains step-by-step instruction for the printer to execute.

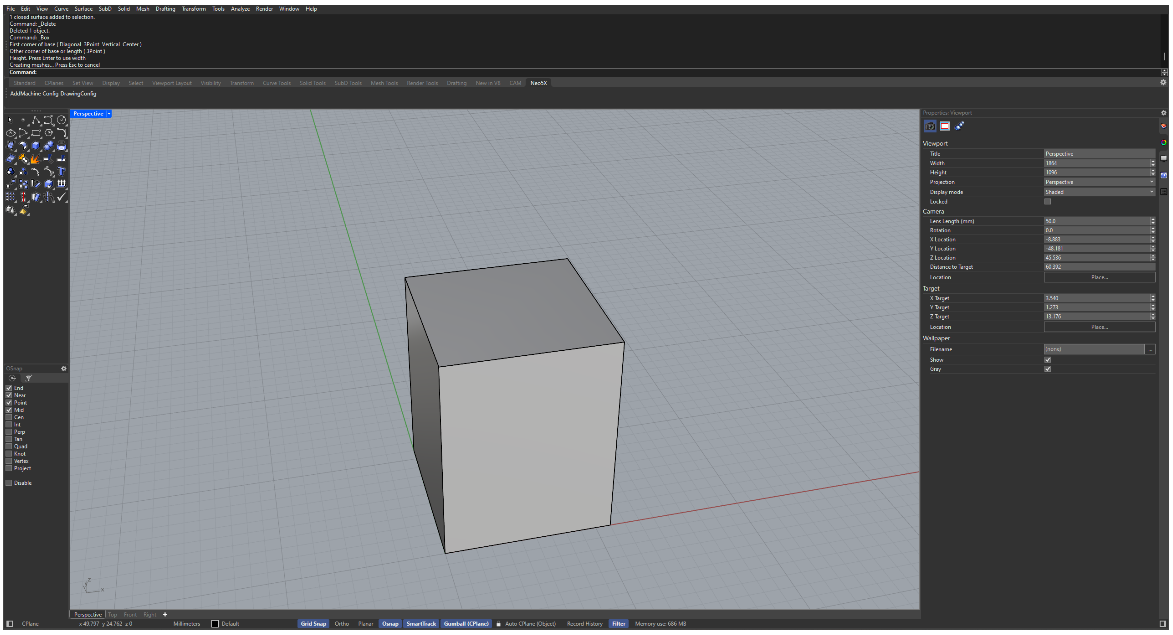
2.1.3 Printing

The final step is the actual printing process, the printer reads the provided G-Code file and executes the instructions one by one. Each instruction either moves the printhead to a specific loaction, extrudes material while moving or configure the printer in some way. When the last line was executed, the object has finished printing and can be removed from the print bed.

2.2 Rhino

Rhino [11] is a versatile CAD software developed by Robert McNeel & Associates, used widely across industries such as architecture, industrial design, jewelry design, and more. It is capable of accurately modeling 3D geometry with a high degree of precision using mainly freeform NURBS (Non-Uniform Rational B-Splines) curves and surfaces. To support the design process, Rhino provides a wide range of tools and features, that manipulate, create or analyze geometry.

Rhino also supports a wide range of file formats, increasing its compatibility with other software and tools.


File Edit View Curve Surface SubD Solid Mesh Drafting Transform Tools Analyze Render Window Help
Standard CPlanes Set View Display Select Viewport Layout Visibility Transform Curve Tools Solid Tools SubD Tools Mesh Tools Render Tools Drafting New in V8 CAM Neo5X
AddMachine Config DrawingConfig
Perspective
Properties: Viewport

```
1 closed surface added to selection.
Command: _Delete
Deleted 1 object.
Command: _Box
First corner of base ( Diagonal  3Point  Vertical  Center )
Other corner of base or length ( 3Point )
Height. Press Enter to use width
Creating meshes... Press Esc to cancel
Command:
```

Viewport	
Title	Perspective
Width	1864
Height	1096
Projection	Perspective
Display mode	Shaded
Locked	☐
Camera	
Lens Length (mm)	50.0
Rotation	0.0
X Location	-8.883
Y Location	-48.181
Z Location	45.536
Distance to Target	60.392
Location	Place...
Target	
X Target	3.540
Y Target	1.273
Z Target	13.176
Location	Place...
Wallpaper	
Filename	(none)
Show	✓
Gray	✓

Figure 2.1: Rhino Interface

In addition to all tools and features that are provided by default, Rhino has the ability to be extended by plugins. These can either be downloaded and installed or self-developed in C++, C#, Python, and other languages using the provided SDKs, granting developers the ability to create custom tools, commands or even entirely new workflows.

2.3 Neo5x

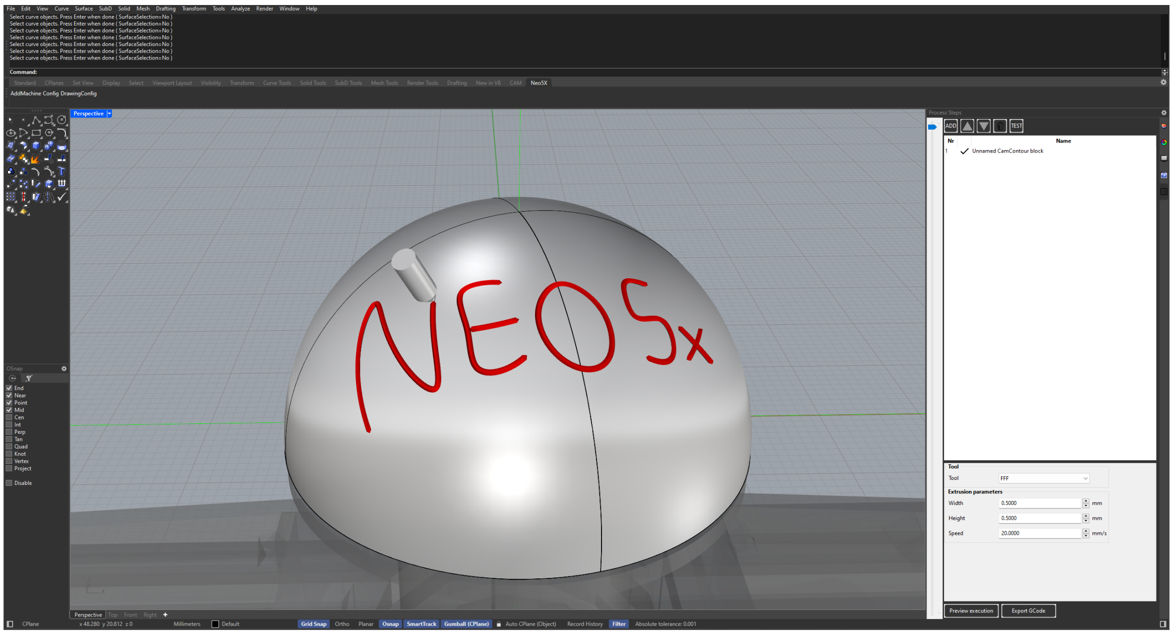
As mentioned in section 2.2, Rhino can be extended in many ways using plugins. Neo5x is one such plugin, aiming to provide a solution for 5-axis tool-path generation that directly integrates into Rhino's design process, turning it into a Computer-Aided-Manufacturing (CAM) tool for 5-axis additive manufacturing that handles the modeling process as well as the slicing process.

Even though the plugin is still in early-stage development, core features have already been implemented, that are essential to enable the generation of 5-axis tool-paths. These features currently include:

- Loading of URDF machine models
- Basic tool and material configurations

- Generation of 5-axis tool-paths from points and normal vectors using Inverse Kinematics
- Visualization of generated 5-axis tool-paths
- Support for exporting different types of G-Code flavors
- Saving and loading of created designs

The core design that builds of these fundamental features is the CAM-Block. In Neo5x, a CAM-Block represents a single process step in a greater manufacturing process. These can be, for example, creating a contour line on an object's surface, performing a P&P (Pick and Place) operation or in the case of this thesis, creating a conformal filled freeform layer on object surfaces.


```
Select curve objects. Press Enter when done ( SurfaceSelection=No )
Select curve objects. Press Enter when done ( SurfaceSelection=No )
Select curve objects. Press Enter when done ( SurfaceSelection=No )
Select curve objects. Press Enter when done ( SurfaceSelection=No )
Select curve objects. Press Enter when done ( SurfaceSelection=No )
Select curve objects. Press Enter when done ( SurfaceSelection=No )
Select curve objects. Press Enter when done ( SurfaceSelection=No )
Command:
```

Nr		Name
1	✓	Unnamed CamContour block

Tool

Tool: FFF

Extrusion parameters

Width: 0.5000 mm

Height: 0.5000 mm

Speed: 20.0000 mm/s

File Edit View Curve Surface SubD Solid Mesh Drafting Transform Tools Analyze Render Window Help
Standard CPlanes Set View Display Select Viewport Layout Visibility Transform Curve Tools Solid Tools SubD Tools Mesh Tools
AddMachine Config DrawingConfig
Perspective
Process Steps
ADD
TEST
Preview execution
Export GCode

Figure 2.2: Neo5x in Rhino

Since Neo5x is itegrated into Rhino, it is able to directly access, modify and create geometry in the Rhino enviroment, which makes aids enormously in creating the input data neccessary for the CAM-Blocks to generate 5-axis tool-paths.

2.4 5-Axis Hardware

To test and validate whether or not the proposed solution generated valid tool-paths, 5-axis hardware is required. The TAMS (Technical Aspects of Multimodal Systems) printer lab at the University of Hamburg, has recently modified an existing E3D Toolchanger [12] with two additional axes, to create a 5-axis printed based on the Open5x project [13, 14] (figure 2.3).

The printer uses a 3 + 2 configuration, with the usual cartesian x,y,z axes and 2 additional axes a and b, that can tilt and rotate the 45mm diameter print bed respectively. As the name suggests, the printer offers multiple tools that can be exchanged freely, including 2 custom made high clearence extruders that were specifically designed to reduce potential collisions in 5-axis printing, a pressure probe that is used for the calibration process developed by Tom Schmolzi's master thesis [15] and a syringe extruder that is used to extrude viscous materials such as conductive inks.

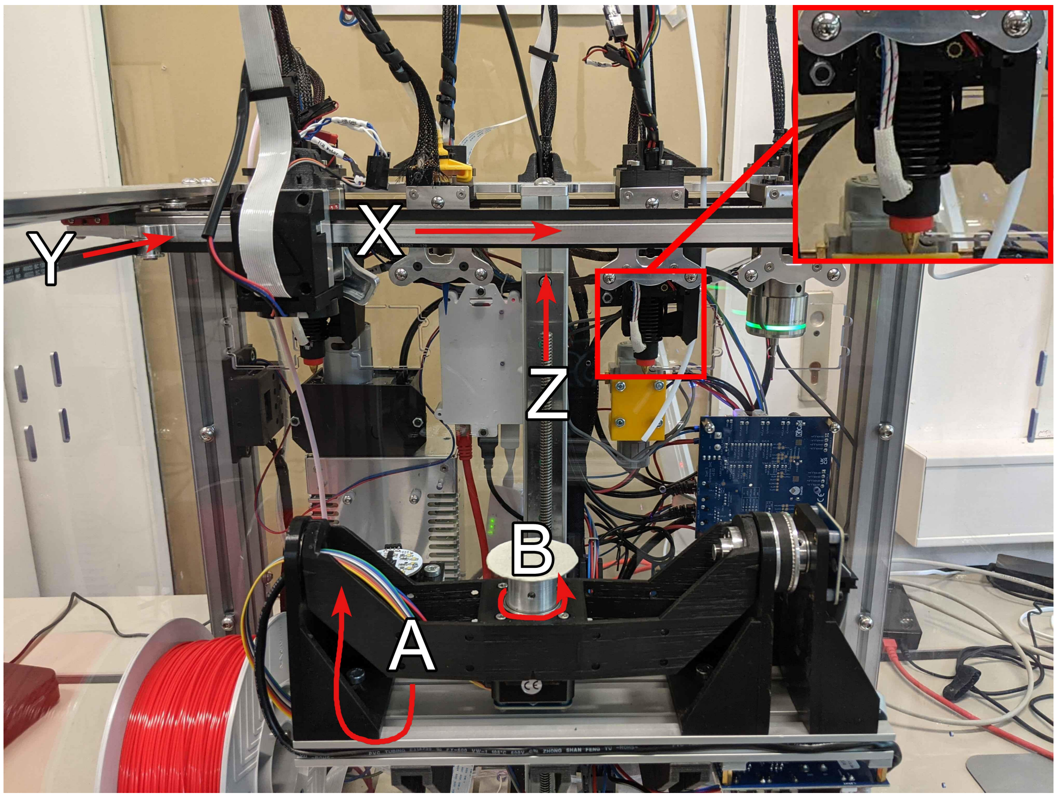


Figure 2.3: 5-Axis hardware
Modified E3D Toolchanger 5-axis printer setup based on the Open5x project [13, 14]. This setup uses a 3 + 2 configuration, with the usual x,y,z axes and 2 additional axes a and b. With the a-axis being attached to the z-axis and the b-axis being attached to the a-axis. The a-axis is for tilt and the b-axis for rotational movements of the print bed. The printer is equipped with 2 custom made high clearence extruder to reduce potential collisions in 5-axis printing. Additionally, the printer has a pressure probe for calibration purposes and an syringe extruder for extruding conductive inks.

Related Works 3

This chapter will introduce several approaches to 5-axis additive manufacturing, that have been proposed in recent years. These include proof-of-concept works, that showcase the potential of 5-axis systems by introducing ways to generate 5-axis toolpaths that satisfy certain criteria.

3.1 Conformal pattern printing

Rodriguez-Padilla et al. [16] introduced an algorithm to generate conformal tool-paths on non-planar surface for 5-axis additive manufacturing. The algorithm takes a tessellated surface and a set of ordered points, that represent the trajectory, as input and projects the points onto the surface. The projected points form a new conformal trajectory along the surface that can be used to generate a tool-path. Next, the algorithm calculates normal vectors at each point on the surface and uses them to generate a tool-path, where each vector represents the direction of the printhead at that point.

Conformal G-Code is generated using both the trajectory and the normal vectors, which can then be used to print the object. The proposed algorithm is able to project complex trajectory on any non-planar surface, to create a conformal tool-path that prints on the object's surface.

3.2 Open5x

The Open5x project [13, 14] is an open-source project that aims to make 5-axis additive manufacturing more acessible to the public. This is done by providing a way to convert existing 3-axis systems into 5-axis systems, by adding two additional axes to the printbed. To be able to generate 5-axis tool-paths, that can be used by the modified printer, a conformal slicer plugin for Rhino[11] Grasshopper has been developed.

It computes conformal tool-paths on the surfaces or geometry using various parameters, which influence the final tool-path. The Slicer uses a combination of forward and inverse kinematic to calculate the necessary axis configurations for any given extrusion point on the surface. Additionally several 5-axis movement optimizations, which enchance the efficiency of multi-axis movement during printing, are made.

Examples were shown to demonstrate the capabilities of the slicer, that include the ability to print support-less, create conformal extrusions on surfaces and improving mechanical properties with a conformally printed layer.

3.3 s^3-Slicer

Zhang et al. [17] introduces a general purpose 5-axis slicer framework called s^3-Slicer. The "s^3" stands for 'support free (SF)', 'strength reinforcement (SR)', 'and surface quality (SQ)' and refers to the three main objectives set by the authors.

The algorithm optimizes the 'Local Printing Direction' (LPD) at each point of the object to achieve the best balance between surface quality, mechanical strength, and support-free printing. Models are represented as a volumetric tetrahedral mesh, each tetrahedron in the model can be rotated to find the best print direction at that point. To find the optimal print direction an iterative quanternion-based optimization loop, that balances all three objectives, is used. Upon calculating the optimal tetrahedron rotations the algorithm creates a deformed model, that can then be used to create curved layers tool-paths.

The printed objects have shown to have improved surface quality, mechanical strength and could mostly be printed without support structures.

Implementation 4

In this Chapter, section 4.1 will provide a brief overview of the interface that Neo5x uses internally to generate 5-axis tool-paths. This is necessary to understand, as the implementation will build upon this interface to enable a user to automatically generate tool-paths that conformally print a filled freeform layer on an object's surface.

Section 4.2 will provide a prelimiary overview of the general approach of the implementation, while section 4.3 will go into the specifics of the implementation.

4.1 Neo5x Interface

CAM-Blocks are the core building blocks of Neo5x, as they serve as the main interface for the user to interact with the plugin within Rhino. Since the implementation will be integrated into Neo5x as a new CAM-Block type, it is important to get a general understanding how CAM-Blocks are created and used within Neo5x. Figure 4.1 shows a flowchart of the creation process of a CAM-Block and the steps are explained in detail below.

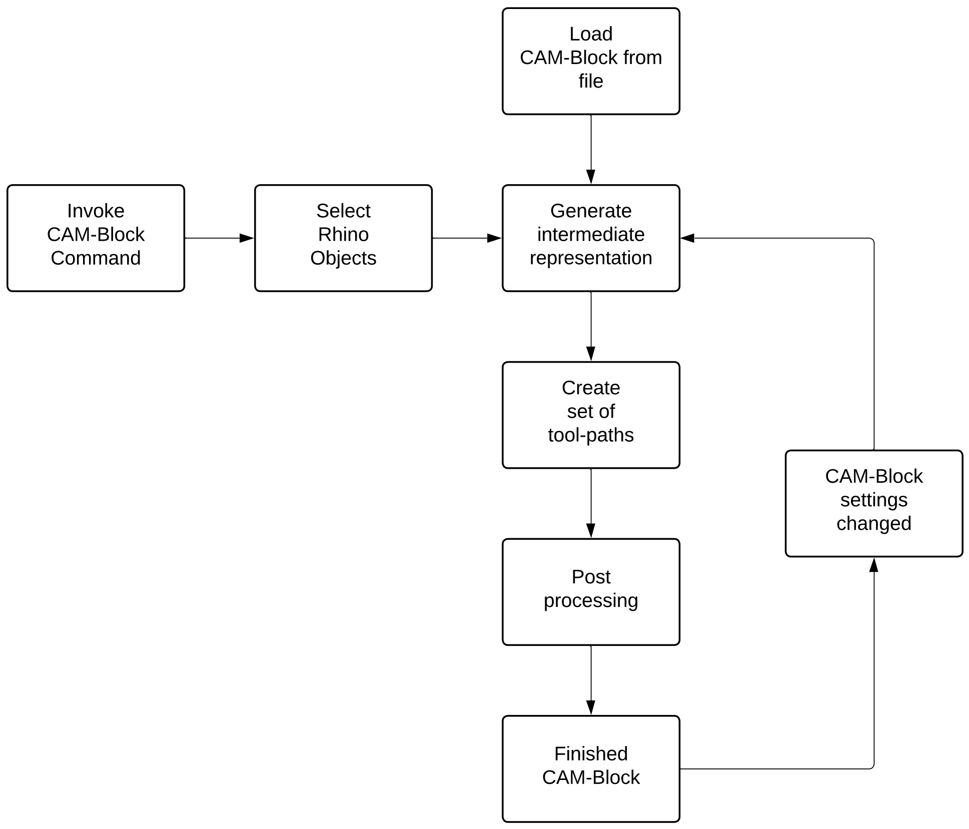


Figure 4.1: Creation process of CAM-Blocks

Figure 4.1 shows all necessary steps to that define the Neo5x CAM-Block interface. Starting with either 'Invoke CAM-Block Command' or 'Load CAM-Block from file' and ending with 'Finished CAM-Block'.

Invoke CAM-Block Command: For every new CAM-Block, a new Rhino command must be created. This command handles user input, which include selecting objects, setting parameters and attempting to create a new CAM-Block object. If the creation is successful, the inputs are passed to the CAM-Block, and the generateToolpath() method, inherited from the base class, is called. Afterward, the CAM-Block, now holding a valid tool-path, is handed over to the Neo5x internal data manager, allowing the user to interact with it further down the line.

Select Rhino Objects: During the CAM-Block command, depending on the specific CAM-Block, the user may be prompted to select certain objects in the Rhino scene. After selection is complete, the CAM-Block obtains a reference to the selected objects, which can then be used in the next step.

Load CAM-Block from file: If the user previously worked on a Rhino project with Neo5x, CAM-Blocks may also be created by automatically loading all necessary information from a YAML file created when saving the project. In this scenario, the CAM-Blocks are directly created after loading the project file, without the user needing to invoke a Rhino command.

Generate intermediate representation: Before a tool-path is created a CAM-Block may generate additional information that is necessary for either the tool-path generation or the post-processing of those generated tool-paths.

Create set of tool-paths: Each CAM-Block holds a set of tool-path objects, that have yet to be populated with actual tool-path data. This is done in the generateToolpath() method, it uses either the initially selected Rhino objects, the immediate representation calculated in the previous step or both to generate the tool-paths. Firstly as Rhino mostly works with NURBS curves and, by extension the user as well, the input will most likely be a set of NURBS curves. Although NURBS curves are very powerful, they also increase the complexity of the tool-path generation process. Therefore, the input curves are converted into a set of poly-lines that approximate the original curves, making it easier to work with them.

Secondly, for every point on the poly-line, a normal vector is calculated using Rhino's SDK functionality. This normal vector is calculated from the object's surface at each point. It may also be necessary to further subdivide the poly-line into smaller segments to accommodate the object's surface curvature. This ensures that the extruder will conform to the object's surface between any two points. This doesn't take general collision detection into account, as it is currently not implemented in Neo5x.

For every point in the tool-path extrusion width, layer height and speed is initially set to a default value. These values can be adjusted in the post-processing step.

Post processing: After the tool-path objects have been generated, a CAM-Block may perform additional post-processing steps, to edit or optimize the tool-paths. For this every single extrusion points extrusion width, layer height or speed can be adjusted, affecting the extrusion till the next point.

Finished CAM-Block: After completing the generateToolpath() call, the CAM-Block is considered finished and can be handed over to the Neo5x internal data manager. This also concludes the CAM-Block Rhino command.

CAM-Block settings changed: Every CAM-Block holds a set of config options that can now be adjusted in the Rhino interface. These usually include extrusion width, layer height, speed as well as the tool being used. Depending on the CAM-Block, there may also be additional config options. When the user changes one or multiple config option, the CAM-Block deletes all previously generated tool-paths and recalculates them with the new configurations set by the user.

4.2 Implementation Preliminary

Before going into detail in Section 4.3, it is important to exactly define what the implementation has to do on its own and what is done in conjunction with Neo5x. Section 4.2.1 provides a brief overview of the general approach of the implementation, discussing two different approaches that could be used to generate the required input for Neo5x's tool-path generation functionality. Section 4.2.4 then concludes with the choosen approach and the requirements needed to properly integrade the implementation into Neo5x.

4.2.1 General Approach

The general goal of the implementation is to provide users, intending to create designs for 5-axis manufacturing, with tools to automatically generate a tool-path for printing a freeform conformal layer on an object's surface. The freeform area on the surface should be definable by the user, using a closed-curve.

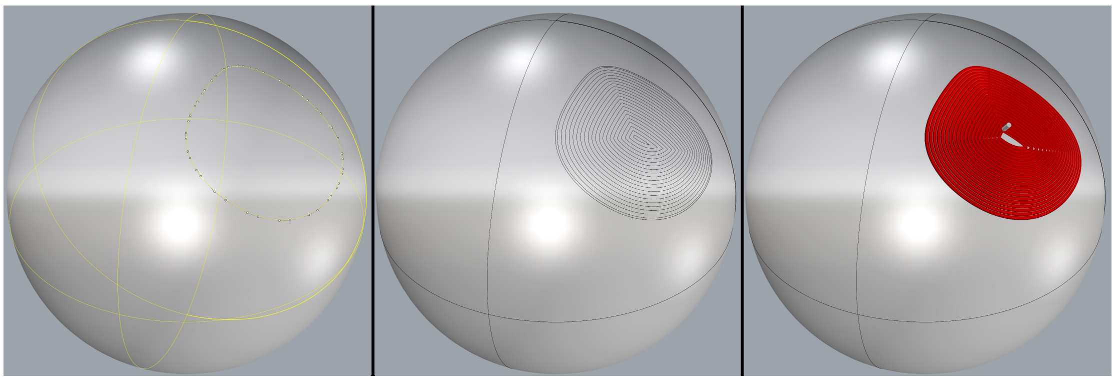


Figure 4.2: General idea of the implementation
In the first step (left) the user selects a surface and a closed-curve on it. In the second step (middle) a space-filling pattern is generated to fill the bounded surface area. In the last step (right) the Neo5x functionality is used to generate a 5-axis tool-path from the curves previously generated

The Neo5x interface described in section 4.1, takes care of the tool-path generation, as long as it is provided with the necessary input. That means the most important part of the implementation is the generation of the space-filling curves on the surface, as those directly affect the tool-path generation process. Optimally, every curve in the generated pattern should be x_{mm} apart from each other, with x_{mm} being the extrusion width. Depending on the pattern, this can not always be guaranteed, but the goal is to get as close to this as possible to ensure that the space will be properly filled when printed. So, the core question is how to effectively generate curves within the bounded area on the surface to ensure the space is filled, while also supporting various fill patterns.

In the following two different approaches will be discussed, and one of the two will be chosen for the implementation. As a baseline the only thing available for the calculation are the previously selected Rhino objects. Additional objects that aid in the calculation may be generated from the selected objects.

4.2.2 3D Offsetting Along Surface

The first approach involves using the initially selected closed curve and generating inward-facing parallel curves along the surface. This method is directly supported by Rhino and can be easily accessed through a built-in function, which requires a curve, surface, offset direction, and distance. It has the big advantage of ensuring that each parallel curve filling the bounded area is exactly x_{mm} apart from each other.

However, a characteristic drawback of this approach, also highlighted by [18], is that self-intersections on offset curves may occur depending on the surface and the curve used. These issues can be seen in figure 4.3 and must be addressed before the curves could be properly used as input for the tool-path generation.

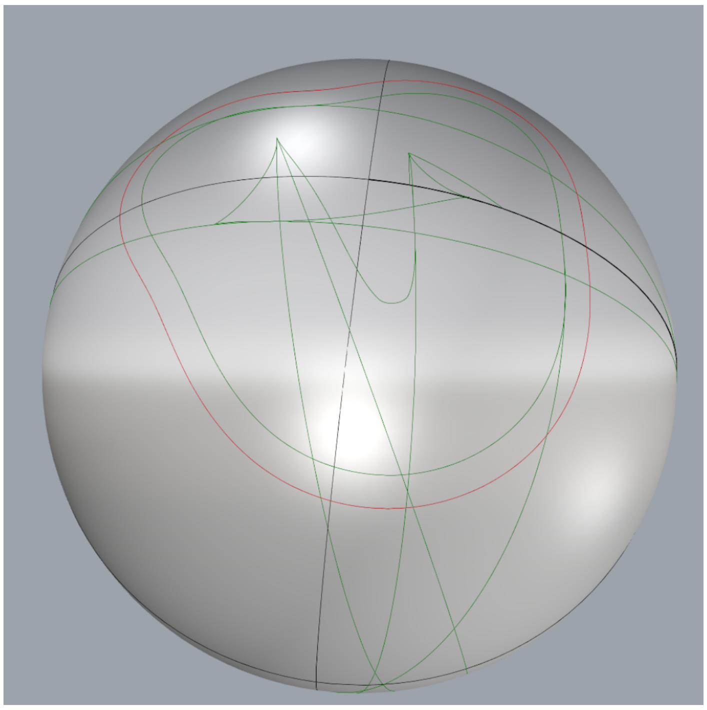


Figure 4.3: Self-intersection with 3D offsetting along surface
The base curve (red) is offset inward twice along a spherical surface (gray) at different distances. The resulting curves (green) show self-intersections, with increasing severity depending on the offset distance. As such they cannot be used directly as input for the tool-path generation.

Ignoring the self-intersections, this approach also offers limited flexibility in generating different space-filling patterns, as the only way to create a pattern is by offsetting curves that lie on the surface, whether that be the intially selected one or additionally generated ones. This makes it rather challenging to achieve patterns made of multiple building blocks, such as a zig-zag pattern, as there is no simple way to connect the building blocks together. Because the connections would also have to lie on the surface, to ensure a smooth tool-path generation.

Regardless, for patterns that can be generated this way it is an optimal approach, as the generated curves have the guarantee of being x_{mm} apart from each other. Resulting in consistently spaced tool-paths, as long as the self-intersections are resolved, benefiting the final print quality.

4.2.3 Projection on surface

The second approach involves using projection as a means to get a space-filling pattern onto the bounded area on the surface. Projection is a simple and versatile method that is also used by several other works concerned with tool-path generation [2, 4, 19, 16]. Rhino itself offers a variety of built-in functions to support such an approach. In addition to being simple to implement, this approach has the advantage of generating a variety of space-filling patterns without being constrained by the surface structure. The generation of a space-filling pattern in 2-dimensional space is easier than in 3-dimensional space or directly on a surface, allowing for greater freedom in the pattern generation process.

Despite its simplicity and versatility projection comes with a major drawback, that being the projection error. When projecting the 2-dimensional pattern back onto the surface, two points may hit the surface at different heights, causing the points to be further apart than intially intended.

The severity of this error depends on the curvature of the surface at a given projection angle and the intial distance between the points. Figure 4.4 demonstrates this issue, with the surface curvature being 45° the projection error can be calculated using the pythagorean theorem.

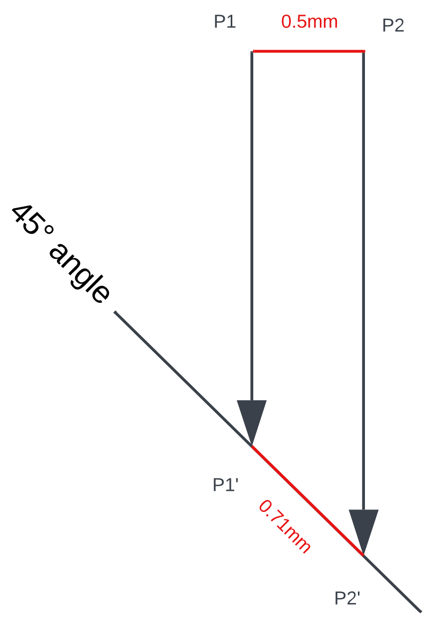


Figure 4.4: Example of projection error
$P1$ and $P2$ are two points with a distance of 0.5mm and are being projected on a surface with a slope of 45°. The distance between the projected points $P1'$ and $P2'$ has increased to 0.71mm in accordance with the pythagorean theorem. As the projection vector hit the surface at different heights, for each point.

This error introduces irregularities in the final tool-path, as distances between points differ from the intial pattern in 2-dimensional space. To ensure a good quality tool-path, these errors have to be compensated for.

4.2.4 Requirements

Both approaches shown in the previous sections have their advantages and disadvantages. Ideally both approaches would be implemented, as some patterns may be better suited for one approach over the other. However, given the scope of this thesis, only one approach will be implemented. The projection method has been chosen for its simplicity, versatility for different surface structures and flexibility to support a wider range of patterns. The only drawback being the projection error that needs a compensation method to reduce the effects of the error.

To achieve the process shown in figure 4.2, the implementation has to implement the following things in Neo5x:

A new CAM-Block type
A new CAM-Block needs to be created to handle the entire functionality of the implementation. As well as that, all features provided by the Neo5X interface must be utilized to ensure that the new CAM-Block seamlessly integrates into the intended workflow set by the Neo5x developers. This includes to make sure that the CAM-Blocks can properly be saved and loaded and providing configuration options, that a user can adjust in the Rhino interface.

A new Rhino command
A new Rhino command needs to be created, allowing the user to create and utilize the new CAM-Block type in their design. In this command, the user should be prompted to first select the surface on which a pattern should be generated, then select a closed curve on the surface that encloses an area, and finally choose one of the available space-filling patterns.

4.3 Implementation Details

In this section, the implementation details will be provided. The following steps can all be categorized under a stage in the creation process of a CAM-Block, as shown previously in Figure 4.1. With sections 4.3.1, 4.3.2 and 4.3.3 belonging to the ‘Generate intermediate representation’ stage. And section 4.3.4 and section 4.3.5 belonging to the ‘Create set of tool-paths’ and ‘Post processing’ stages respectively.

4.3.1 Projection to Plane

Before a space-filling pattern can be generated and projected to the surface, a 2-dimensional closed-curve is required to represents the bounded area on the surface. A simple way to achieve this is by projecting the initially selected closed-curve onto a plane in 3-dimensional space, effectively flattening it in the process. Ideally, the plane should be parallel to the general direction of the bounded surface area. This will help minimize the projection error when the pattern is projected back onto the surface and prevent distortion of the flattened curve, as shown in figure 4.5. The distortion of the flattend curve causes addtional issues, as the area the curve should represent no longer match, leading to a less dense fill space-filling pattern than required.

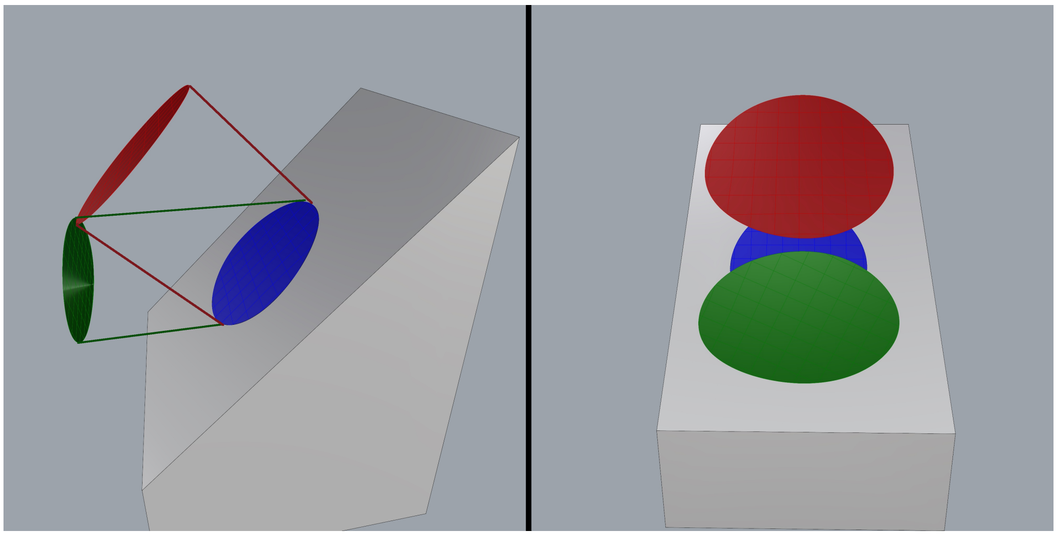


Figure 4.5: Projection of initial closed-curve (blue) onto two different planes in 3-dimensional space, shown in two different perspectives. The surface (gray) has a curvature of 45°, the Blue curve is projected onto a parallel plane (red) and a plane parallel to the x-axis (green). The green curve is clearly more distorted, resulting in a curve with less fillable space, reducing the fill density pattern when projected back (gray).

Rhino’s SDK functionality provides support for both creating planes and projecting curves onto them, enabling a simple and efficient implementation of this step. To create a plane, Rhino requires a normal vector to define the plane’s orientation and an origin point to define the plane’s position in 3-dimensional space.

As previously mentioned, the plane should be parallel to the general direction of the bounded surface area. Therefore, a method is required to calculate an appropriate normal vector that represents this direction.

Shown in Figure 4.6, the normal vector is calculated by averaging many normal vectors along the closed-curve, overall it proved to be a good approximation of the general direction and can be used to construct a plane, there are however some edge cases, that will be highlighted later in section 5.2.

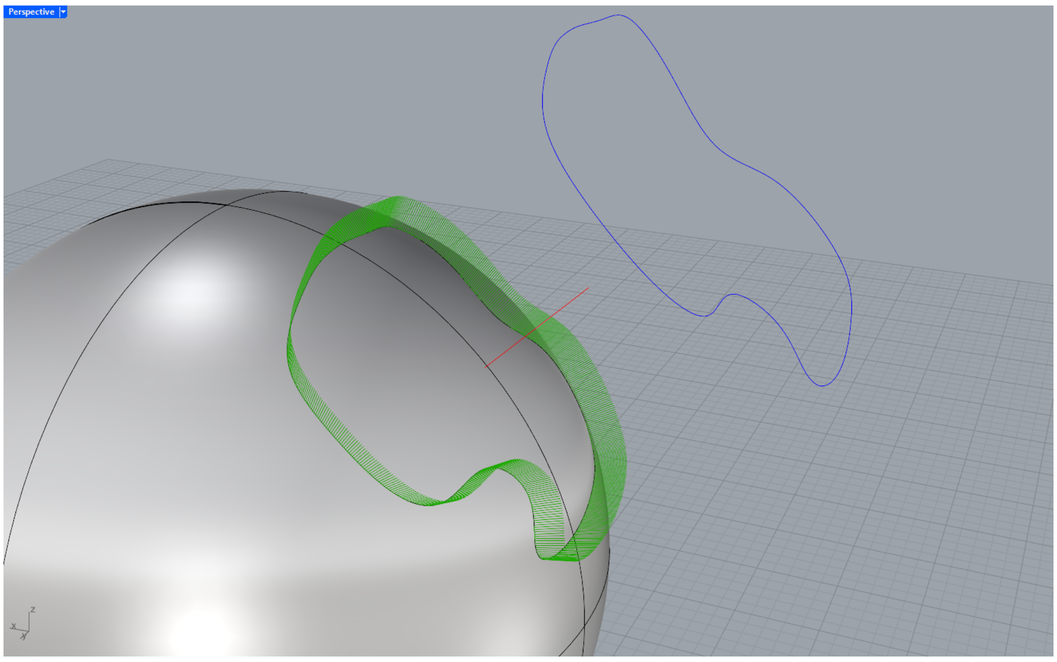


Figure 4.6: Projection of a closed-curve onto a plane
Surface normal vectors are evenly calculated along the closed curve on it (green). All calculated normal vectors are averaged to get an approximate direction vector (red) of the general direction. The averaged normal vector and a point above the surface is used to create a plane in 3-dimensional space, where the closed-curve on the surface is projected to with a built-in Rhino function (blue).

With the normal vector calculated, an origin point is required next to define the plane's position in 3-dimensional space. The origin point is calculated in a way that ensures the plane's positioning to always be above the selected surface. As this will prevent complications down the line with how Rhino hadles surface and projections. It is calculated using the center of the bounding box that surrounds the object, the diagonal length of the bounding box and the normalized normal vector. With the plane constructed, an in-built function can be used to project the closed-curve onto it, resulting in the blue curve seen in figure 4.6.

Additionally, the normal vector will be saved, as it will be used later to project the generated space-filling pattern back onto the object's surface, as explained in section 4.3.3.

4.3.2 Pattern Generation in 2-dimensional Plane

Since the goal is to have a completly filled conformal layer on the surface, the space in-between the planar boundary curve must be filled, before it will be projected back onto the surface. For this, space-filling patterns will be employed [20]. The implementation supports two common patterns widely used in additive manufacturing. The first is the **Contour-Parallel** pattern, which fills the area by incrementally offsetting the boundary curve inwards. The second is the **Zig-Zag** pattern, which fills the area by connecting a series of straight lines. Both patterns are shown in figure 4.7.

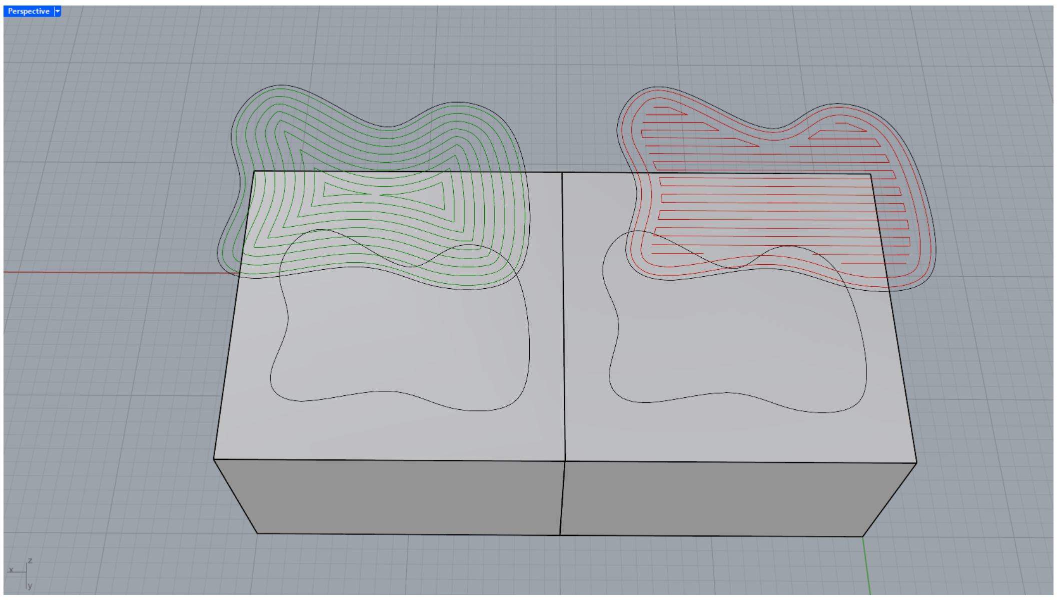


Figure 4.7: The two patterns supported by this implementation: The Contour-Parallel pattern (left) and the Zig-Zag pattern (right).

As previously mentioned in section 4.2, the CAM-Block holds different config objects one of which controls the base extrusion width for the tool-path generation. Regardless of the chosen pattern, the extrusion width has to be considered while generating the input curves. With the base extrusion width set to x_{mm}, curves generated by the pattern should optimally be x_{mm} apart from each other. Although the spacing can only be guaranteed for the pattern on the plane, the projection error will cause the distance between the curves to vary. As this will be compensated for later explained in section 4.3.5, this variable is not considered in the pattern generation step itself.

However, in addition to the generated curves data structure, every pattern generator also generates an additional data structure that holds local neighbourhood information between the generated curves, which are invaluable for the projection error compensation step. The exact usage of this data structure will be explained in section 4.3.5.

4.3.2.1 Data Structures

For the entire implementation to work seamlessly, a pattern generator must calculate two sets of data, first being the curves that make up the pattern and the second being a local neighbourhood information.

The data structures are defined as follows:

- **Curve Data Structure**
  The data structure used to store the generated curves is a simple array of arrays of Rhino curve objects. With the outer array representing a single segments of the pattern, and the inner array representing the curves that make up the segment as seen in listing 4.3.2.1 or as a visual reference in figure 4.8.

```
//Segments[i][j]: i being the i’th segment and j being the j’th curve in
   segment i
typedef std::vector<std::vector<ON_Curve*>> Segments;
```

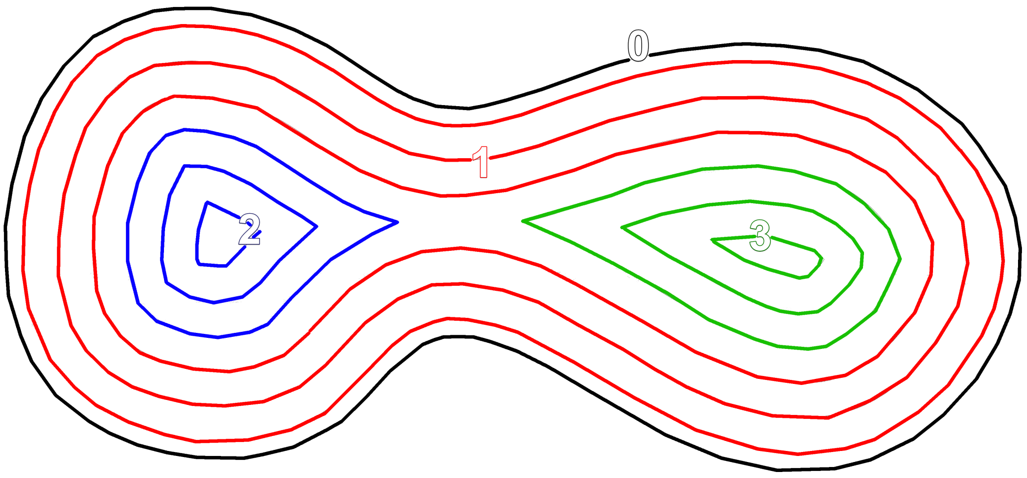


Figure 4.8: Visual representation of curve data structure
Each same colored curves belong to the same segment, denoted by the numbers. With segment 0 begin the boundary curve.

  This grouping is done as a small optimization for the tool-path generation in a later step, as each segment represents a group of curves that can be used to create a continuous sub tool-path. Additionally, it keeps the curves better organized, making them easier to access later on. The first segment is always the boundary curve.

- **Neighbourhood Data Structure**
  The neighbourhood data structure builds upon the curve data structure by holding additional information for every generated curve. Every curve in the pattern has a group of curves that were generated before it, these previous curve can be in the same segment or in a previous segment.

  These curves are defined as the local neighbourhood of each curve and are stored in an array with n entries, with n being the number of segments in the curve data structure. Because the first segment is always the boundary curve, it is kept empty, as it has no defined neighbours.

Each subsequent array entry *i* holds a map where the key *j* is the index of the curve in the *i'th* segment and the value is an array of pairs. Each pair consists of a segment index and a curve index that is inside the segment. With every pair in the array representing one neighbour of the current curve.

```
// For every segment i, there is a map with j entries, map(j) = all
    neighbours of curve j
typedef std::map<int, std::vector<std::pair<int, int>>> NeighbourhoodMap;
// Neighbourhood has n entries, with n being the number of segments
typedef std::vector<NeighbourhoodMap> Neighbourhood;
```

4.3.2.2 Contour-Parallel Pattern

The Contour-Parallel pattern fills the bounded area by iteratively offsetting the boundary curve inwards by the extrusion width until no smaller curve be created this way. Except the first offset curve, which is offset by half the extrusion width instead of the full extrusion width, this ensures that entire area the user selected is filled.

Rhino offers a built-in function to offset planar curves, which will mainly be used to generate this pattern. The most important inputs are the curve to offset, the offset distance, the direction in which the curve should be offset and an array to store the results in. With the curve to offset being the planar boundary curve, the offset distance being the extrusion width and the direction being inwards. Additionally, the function returns an integer value that indicates how many curves were generated. This is an important value, as it helps to sort the curves into different segments. Depending on the boundary curve, a single offset can yield multiple curves, forming individual islands (Figure 4.9), requiring new segments for each formed island.

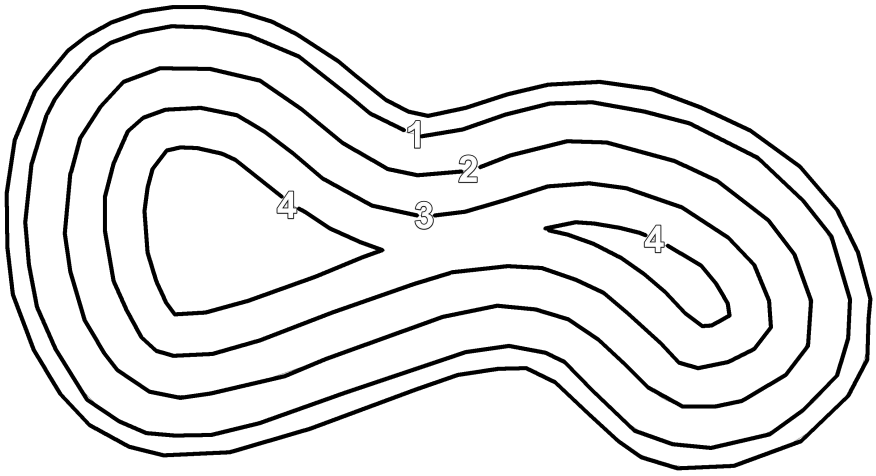


Figure 4.9: Contour-Pattern can from individual islands
A Contour-Parallel pattern in the making, the number denotes the iteration. The fourth iteration yielded two curves, each forming an individual island. Each island must be stored in a separate segment.

In each iteration, the boundary curve is first offset by the current offset value, initially set to half the extrusion width, using the built-in Rhino function. The return value *n* is then checked, if it is equal to zero, no new curves were generated, ending the loop. Otherwise, *n* is compared against the value of the previous iteration. If it is different, *n* new segments will be created, housing the newly formed islands. If *n* is equal, the new curves are instead stored in the last *n* corresponding segments. Additionally, the variable for the previous iteration is updated.

For each generated curve, additional neighbourhood information are calculated next. For the Contour-Parallel pattern, a curve may only have one neighbor: the last curve of the previous segment that surrounds the current one, if new segments have been created, or otherwise, the previous curve in the same segment.

Finally, the current offset value is incremented by the extrusion width and the loop continuous with the next iteration.

Figure 4.10 shows the final result for several boundary curves.

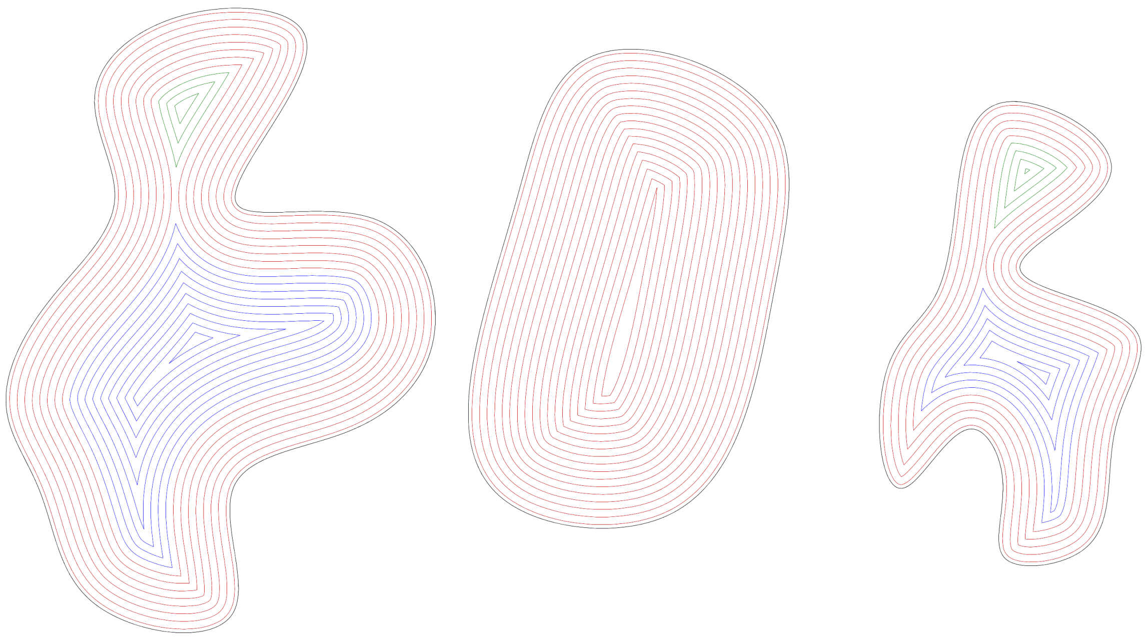


Figure 4.10: Contour-Parallel pattern
Results for different boundary curves (black), segments are highlighted in different colors.

4.3.2.3 Zig-Zag Pattern

The Zig-Zag pattern fills the bounded area by first generating a number of perimeter curves. Afterward a series of straight parallel scan-lines, inside the last perimeter curve are created at equal distances from each other, while connecting consecutive scan-lines alternately at the start and end points.

The perimeter curves are generated like the Contour-Parallel pattern, but limiting the number of iteration to a set parameter. An additional perimeter curve is generated, that will only be used for intersecting the scan-lines, ensuring that only the parts inside the bounded area are kept.

The series of scan-lines are generated similarly to the perimeter curves, in that they are offset with an increasing distance, which depends on the extrusion width. To start this process, an initial line has to be calculated, for that an edge of the bounding box surrounding the bounding curve is used. This approach unfortunately only allows for a few fill directions, but is considered sufficient for now.

Similarly to the islands that form in a Contour-Parallel pattern, an offset of the straight line can yield multiple sub-lines when multiple intersections occur with the perimeter curves, as seen in Figure 4.11. For each sub-line a new segment is created. This is done because a continuous tool-path between the previously generated scan-lines cannot be guaranteed, without additional optimization work.

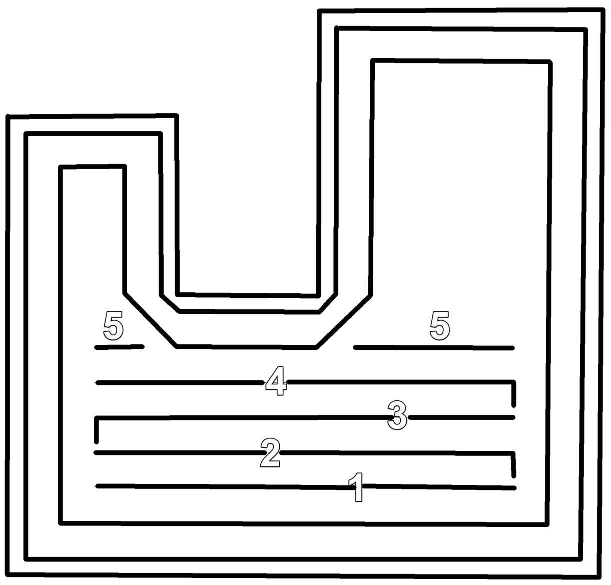


Figure 4.11: Zig-Zag pattern split
The Zig-Zag pattern in the making, the number denotes the scan-line iteration. The fifth iteration yielded two lines after intersection, a continuous tool-path cannot be guaranteed, thus two additional segments are created.

After generating the perimeter curves, the intersection curve and a scan-line, a loop is used to generate the remaining pattern. In each iteration, the scan-line is offset by the extrusion width, creating exactly one new scan-line. Next, intersections are calculated. If the number of intersections n is equal to zero, the curve is discarded, ending the loop. Otherwise, n is compared against the intersections of the previous iteration. If different, n new segments will be created, each housing part of the trimmed scan-line. If n is equal, the newly trimmed scan-lines are instead stored in the last n corresponding segments.

For the perimeter curves, the neighborhood information is calculated exactly the same as for the Contour-Parallel pattern. For the scan-lines, the calculation is similar, but with the difference that a scan-line may have multiple neighbors, as it can be parallel to several scan-lines created in the previous iteration. Since every scan-line connection is parallel to the last perimeter curve, it is additionally added as a neighbor for every scan-line. Finally, the offset value is incremented by the extrusion width and the loop continuous with the next iteration.

Figure 4.12 shows the final result for several boundary curves.

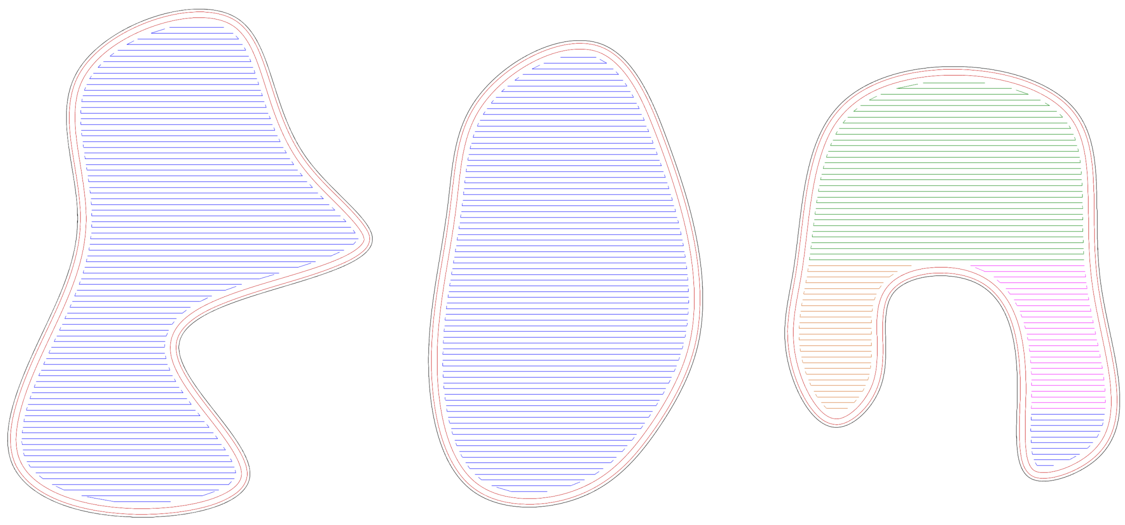


Figure 4.12: Zig-Zag pattern
Results for different boundary curves (black), segments are highlighted in different colors.

4.3.3 Projection back to Surface

In order to use the generated pattern as input for the tool-path generation, it has to be projected back onto the object's surface. Just like projecting curves to planes, Rhino offers multiple built-in function to project curves onto surfaces. All of these function require the curves to project, the surface to project onto and a projection vector. The projection vector calculated in Section 4.3.1 will be used. Before it can be used, it first must be negated so that it will project the curves in the surface's direction.

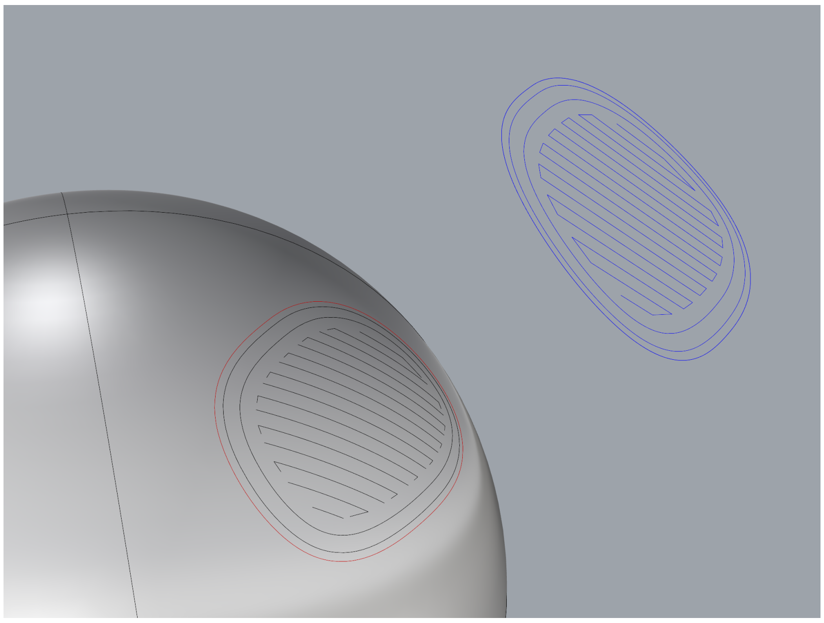


Figure 4.13: Projection of curves back to surface
The blue curves are projected back onto the base object (gray), resulting in the black curves.

The curves shown in figure 4.13 are the projected curves, which were already subdivided and converted into poly-lines. These can now properly be utilized in the next step.

4.3.4 Generation of Tool-Path Objects

Once the input curves have been projected onto the surface, they can be used to generate tool-path objects. These are necessary for further interface with the Neo5X functionality, as the data they provide will be used by the Neo5x inverse kinematic solver to generate G-Code.

Tool-path objects consist of an ordered list of points that represent a continuous line extrusion, moving from point to point, with each point being defined by a position, extruder orientation vector, extrusion width, layer height, and speed.

For each previously generated curve segment, a new tool-path object is created, that represents a part of the pattern that can be printed as a continuous extrusion. Poly-lines are used to represent the curves, which is particularly useful since they are also made up of an ordered list of 3-dimensional points that can be directly used to append points to the tool-path object. Additionally, for each appended point, the nearest normal vector of the object's surface to the point is calculated and used as the corresponding extruder orientation. The extrusion width is set to the same base value that was used in the pattern generation. The height and speed are also set to default values of 0.2 and 20 respectively, but can be later adjusted by the user.

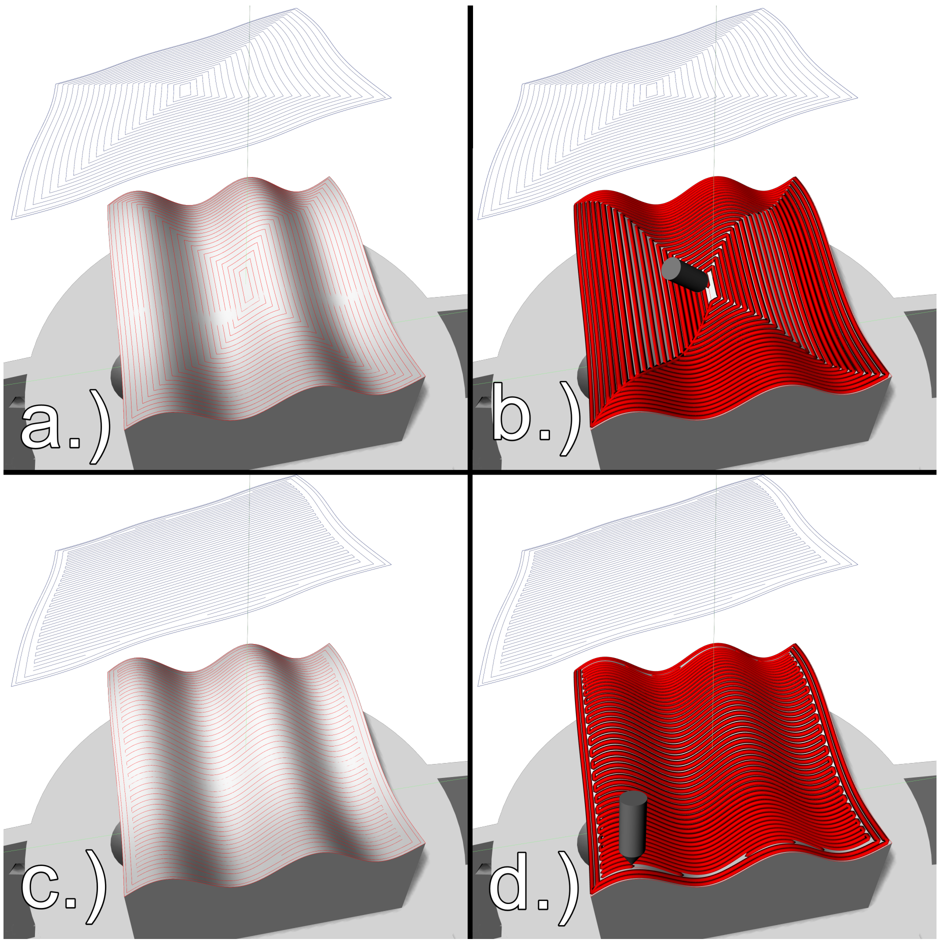


Figure 4.14: Visualizied tool-paths
(a) and (c) show the input curves (red) with a contour-parallel and zig-zag pattern respectively. While (b) and (d) show the visualizied tool-paths that were generated from the input curves.

By appending all points of each curve in a segment to the tool-path object, all neighboring curves are connected at their start and end points, forming a continuous extrusion. Repeating this process iteratively for n functional segments yields exactly n tool-path objects, each containing m points, where m is the sum of the points from all curves in the segment. Each of which can be adjusted to fine tune parts of the extrusions if necessary.

Once all n tool-path objects have been created, the user can view the visualized tool-path in the Rhino interface and choose to export it as G-Code or run a compensation pass before exporting it.

This will result in a tool-path that prints a conformal layer on the bounded surface area, which will be filled with the chosen space-filling pattern. A visualization of such tool-paths for both the Contour-Parallel and Zig-Zag patterns can be seen in figure 4.14.

4.3.5 Projection-Error Compensation

Although the generated tool-path is valid and could be exported as is, it is far from optimal. The projection error, as explained in section 4.2.3, can cause visible gaps between extrusions. Such an example can be seen in figure 4.15.

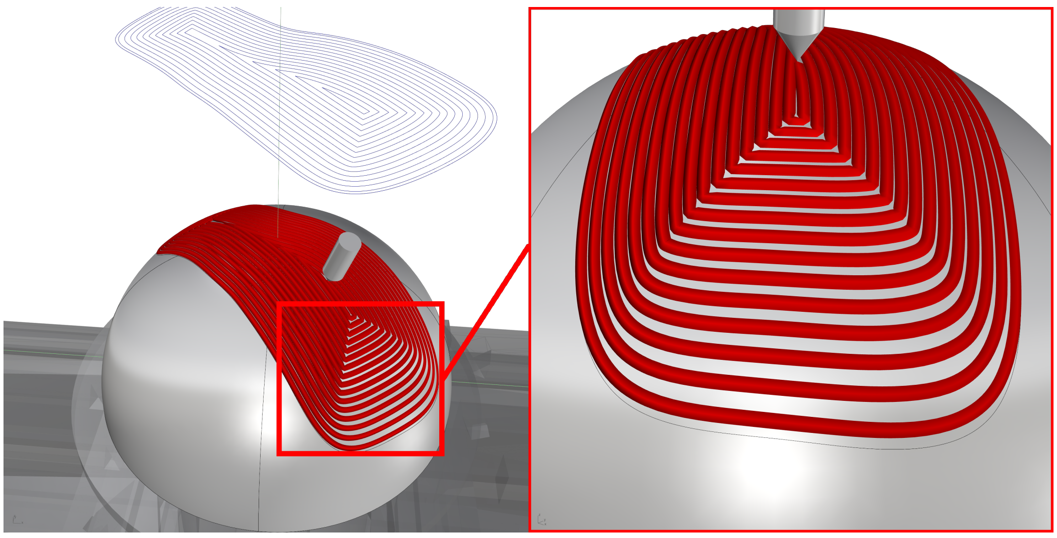


Figure 4.15: Effects of projection error on tool-paths
Due to the projection error, the final tool-path (red) exibits visible gaps between extrusion lines. Without compensation, the print quality would be negatively affected.

In turn, it fails to meet the expectations of a fully filled area. As the projection error is neither considered in the pattern nor in the tool-path generation, this post-processing step is required to reduce the effects of the projection error.

4.3.6 Compensation Approach

There are multiple ways one can tackle this issue. One approach would be adding additional extrusions between the existing ones, when the distance between two neighbouring extrusions exceeds a certain threshold. Ensuring a fully filled area, at the cost of increased print time. However, effectively integrating the new extrusions into the existing tool-path is a challenging task.

Another approach, and the one chosen for this implementation, involves dynamically adjusting the extrusion widths at each point. This way parts of the extrusion can be made wider or narrower, effectively compensating for the projecting error. This however can only be done to a certain degree, as extrusions that are too wide or too narrow can not be printed properly. Thus the maximum and minimum extrusion width adjustments are limited to a $1.5 \cdot extrusion\_width$ times and $0.75 \cdot extrusion\_width$ respectively. The evaluation chapter 5 will address the cut-off point where the compensation method becomes ineffective due to this limitation.

The compensation process is carried out in a greedy, iterative manner, where each extrusion point of a curve has its extrusion width adjusted based on the closest distance to its locally defined neighbour, defined in section 4.3.2, and the extrusion width of the neighbour at that point. This maximizes the extrusion width to fill the gap between extrusions. In order to work properly, the order of adjustement is important, as the extrusion width at each extrusion point of a curve depends on the local neighbours already adjusted extrusion width.

The process begins with the outermost curve, whose neighbor is the boundary curve, and continues with the next curve in the current segment. Once all curves in a segment have been adjusted, the process moves on to the next segment.

The new extrusion width is calculated with a simple formula:

$$new\_extrusion\_width = (distance - \frac{width}{2}) \cdot 2 \tag{4.1}$$

Where $distance$ is the gap between the extrusion points and the closest parallel point of the neighboring curve and $width$ being the extrusion width of the neighboring curve at that point. The term in parentheses calculates the remaining gap between the extrusions that needs to be filled by the current extrusion. Then the calculated term is multiplied by two to ensures that the extrusion width is double this remaining gap, as extrusions are composed of two halves. This formula ensures that the extrusion width is adjusted in a way that fills the gap between extrusions as much as possible without causing overlapping.

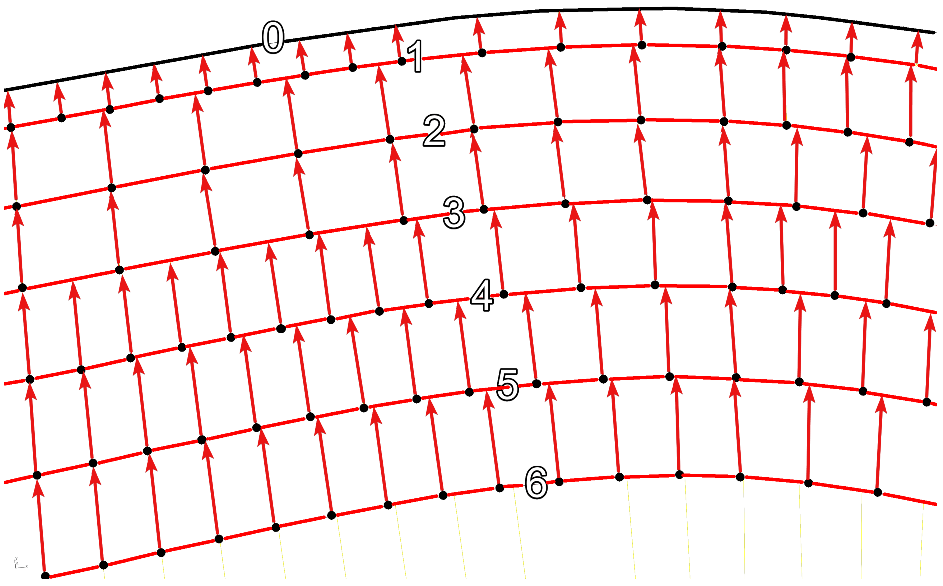


Figure 4.16: Compensation process visualized
Compensation starts with the outermost, non boundary curve, and adjusts the extrusion width of each extrusion point based on the distance to its neighbour and the extrusion width at this point, marked by the red arrows. The process continues inwards with the next curve, when all extrusion points have been adjusted this way.

Since the local neighbourhood information used, only refrences the segments and curves in the curve data structure, an additional map has to be calculated that maps a segment and curve index to the corresponding tool-path object and point index. This is done by calculating a culmulative index array made of the sum of points in each segment, where the index of curve i in segment j return an index that is the sum of all points in segment j up until curve i. This index can then be used to acces the corresponding point in the tool-path object, that are needed for the calculation of the new extrusion width.

Figure 4.16 visualizes the compensation process for the Contour-Parallel pattern. Where the neighbour of curve i is the curve $i - 1$. The compensation process begins with the outermost curve (1) and adjusts every extrusion point along the curve, using the distance and extrusion width of the neighbouring curve at that point, highlighted by the red arrows. When every extrusion point has been adjusted for the current curve, the process continues inwards with the next curve $i + 1$

This greedy compensation can be applied to any pattern, as long as the pattern provides local neighborhood information for each generated curve. Since the compensation of a curve's extrusion width depends on its neighbour, determining the neighbours is crucial and not always trivally determined as a curve may have multiple neighbours for different extrusions points, as seen in the Zig-Zag pattern.

The final result of the compensation process visualized in figure 4.17.

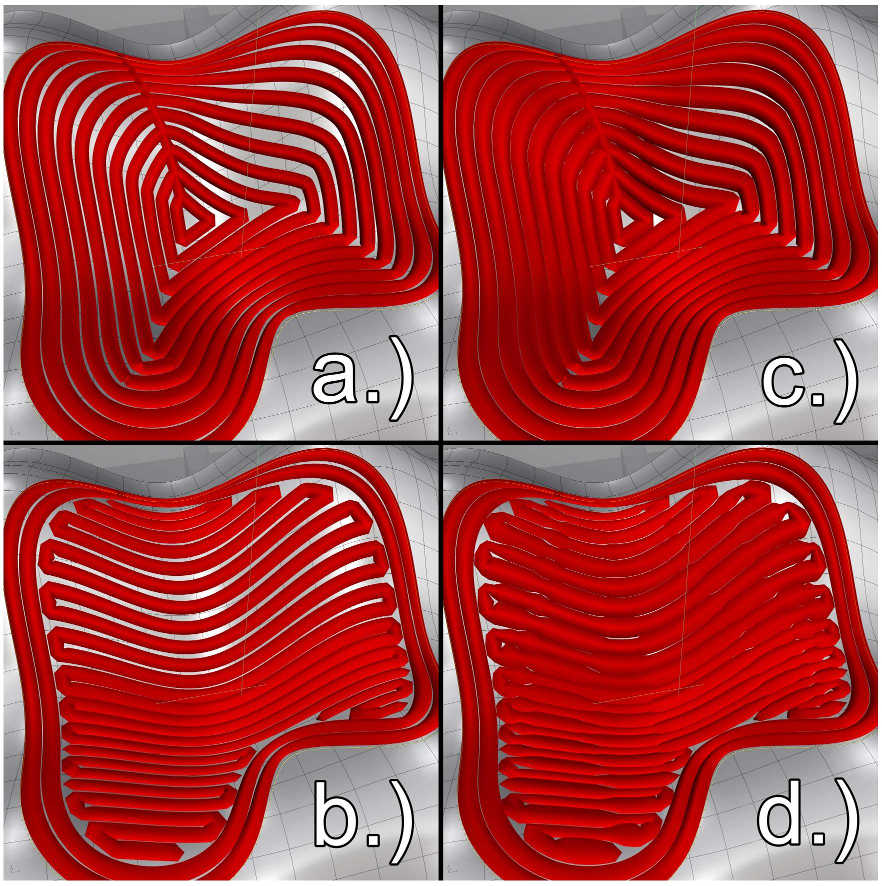


Figure 4.17: Compensation before and after
(a) and (b) show close-ups of the tool-paths for the Contour-Parallel and Zig-Zag patterns before compensation, where clear gaps between extrusions are visible. After compensation, shown in (c) and (d), the gaps between extrusions are significantly reduced.

Evaluation 5

This Chapter focusses on the evaluation of the implementation, starting with section 5.1. The generated G-Code produced by the newly implemented CAM-Block is used to conformally print a filled layer on top of the enclosed surface area of a simple object. First general results are presented, that showcase both the Contour-Parallel and Zig-Zag space-filling patterns. Then, focus is put on the effects of the projection error and compensation on the printed results.

Furthermore, key limitations of the current implementation, that negatively affect the usability, are discussed in section 5.2. For each limitation, the problems are described, and in some cases possible solutions are proposed. That can be considered as potential future work to improve upon the current implementation.

5.1 Print Results

Printing is done on the modified E3D Toolchanger described in section 2.4. The printing process is conducted in three steps:

- First, the printer must be calibrated, this is done using the procedure and guide developed in Tom Schmolzi's master thesis [15].
- Next, the base object is sliced with PrusaSlicer and printed without using the additional 2 axes, as full model 5-axis slicing is not yet supported.
- Finally, the conformal layer on the object's surface, sliced by Neo5x using the associated CAM-Block, is printed will all 5-axes.

Each of the printing steps uses one of two avaiable extruder, both of which are equipped with a $0.5mm$ nozzle, but are loaded with different colored PLA filament to distinguish between the steps. The base object is printed with white filament, while the conformal layer is printed with red filament.

5.1.1 First Test Prints

First test prints are shown, showcasing the Contour-Parallel and Zig-Zag space-filling patterns. The base object is a simple cylinder with an elliptical top, reminiscent of a buzzer. For the boundary curve, a simple circle was placed on the elliptical top, which will cover most of the elliptical surface. Each created CAM-Block had its extrusion width set to $0.5mm$, layer height set to $0.2mm$ and compensation turned off.

Due to a bug in Neo5x's volume calculation, more material was extruded than intended, resulting in slight over-extrusion in the printed parts. These prints can only be used to evaluate, if the general functionality of the implementation works as intended. That is, whether tool-paths were generated that generally fullfill the goals set in section 1.2. Since the over-extrusion masks the effects of the projection error, no real insight than the general functionality can be gained from these prints.

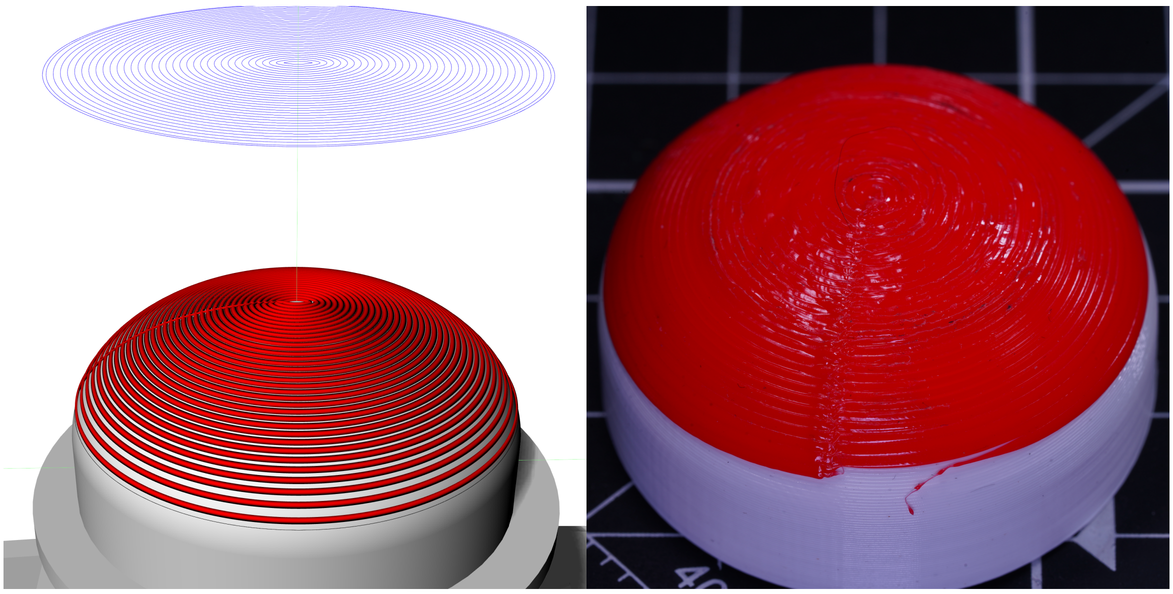


Figure 5.1: First test print: Contour-Parallel
The left image shows the visualization of the tool-paths generated by the CAM-Block. Settings used: Contour-Parallel pattern, $0.5mm$ extrusion width and $0.2mm$ layer height. The right image show the printed result, that differs from the visualization due to slight over-extrusion.

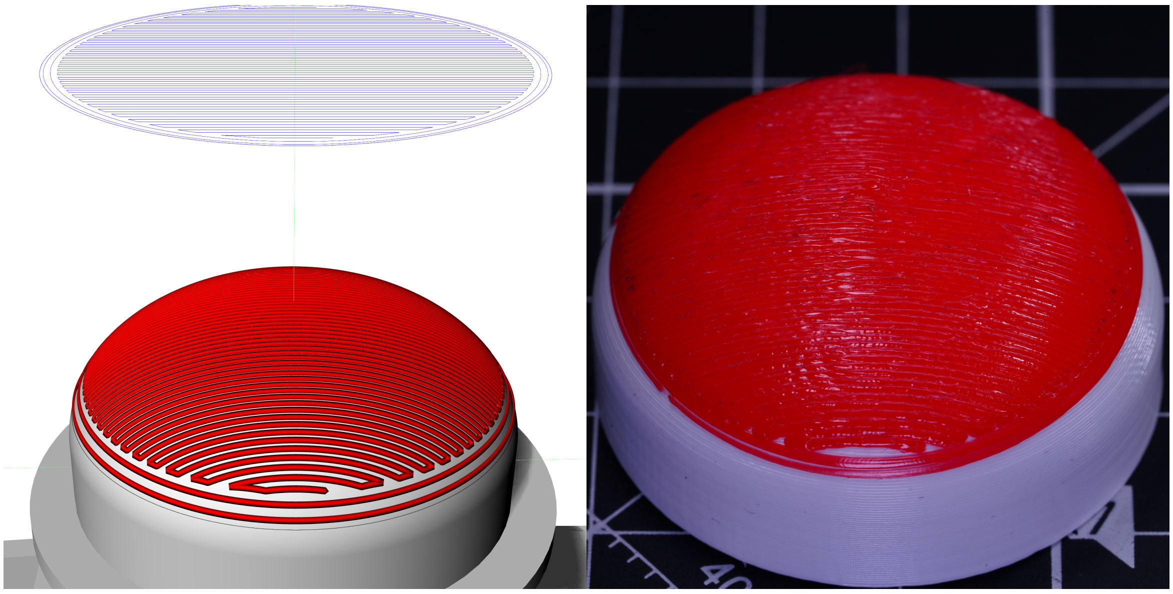


Figure 5.2: First test print: Zig-Zag
The left image shows the visualization of the tool-paths generated by the CAM-Block. Settings used: Zig-Zag pattern, $0.5mm$ extrusion width and $0.2mm$ layer height. The right image show the printed result, that differs from the visualization due to slight over-extrusion.

Disregarding the slight over-extrusion, both test prints were completed without issues, conformally printing along the surface using all five available axes. The surface area bounded by the circle was filled as expected, in both cases resulting in a conformal layer that eliminates the stair-stepping effect caused by planar printing of the base object. Both space-filling patterns can easily be distinguished by a visual inspection of the printed parts.

Since a convex boundary curve was chosen, the resulting conformal layer could be printed as single continuous extrusion for the Contour-Parallel pattern (figure 5.1) and with one travel move for the Zig-Zag pattern (figure 5.2).

Overall the first test prints were sucessful and showed that the general functionality of the implementation works as intended and can be used to conformally print on surfaces of already printed parts.

5.1.2 Projection Error & Compensation

To evaluate the effects of projection error and compensation on printed objects, a second round of test print were conducted. The same base object and boundary curve as in section 5.1.1 were used. Since the boundary curve lies at the very edge of the elliptical top, the projection error is most pronounced at the edges, because the surface slope will approaches 90° near the very edge with the calculated projection vector. As such, this test represents one type of worst-case scenario and is ideal to observe both the projection error and the potential benefits and limits of the used compensation. Additionally, the Contour-Parallel pattern was chosen, as all extrusions are parallel to each other, further emphasizing the projection error at the edges.

Although the volume calculation bug mentioned in section 5.1.1 was still present, flow rates were adjusted to address the issue and achieve extrusion widths as close to the intended values as possible.

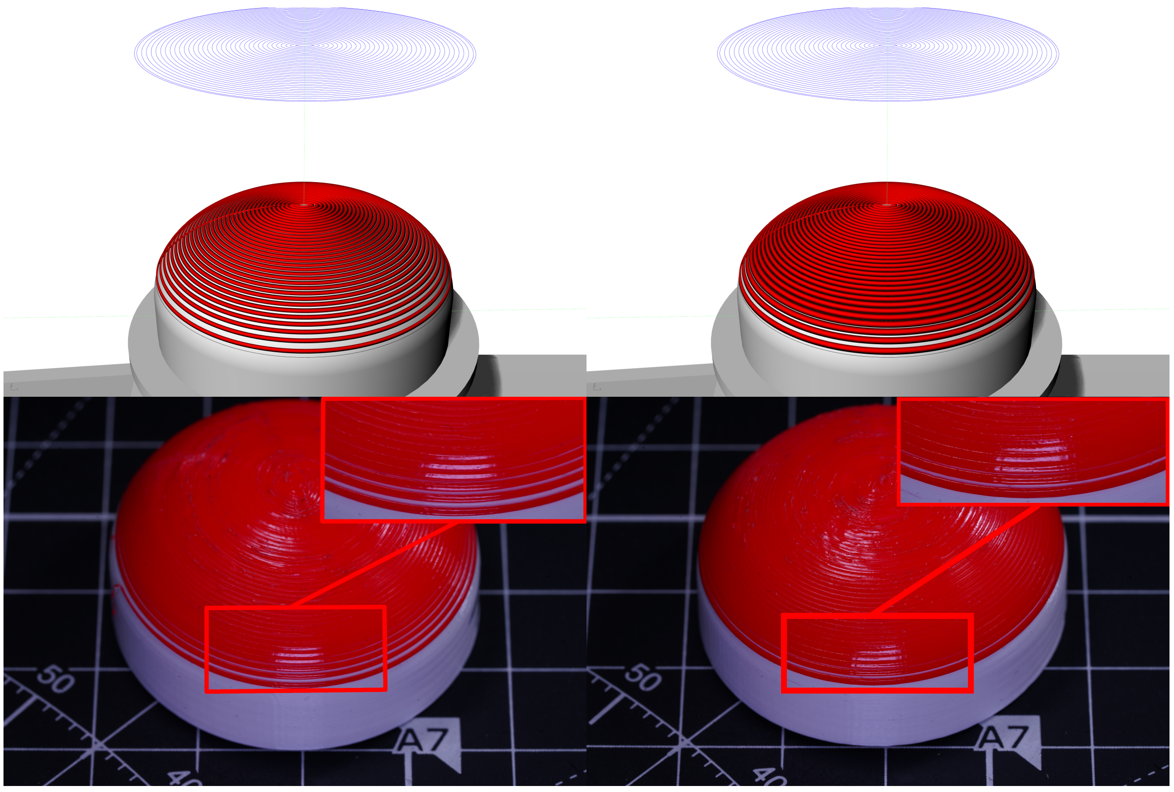


Figure 5.3: Compensation print comparison
Both sides show the visualizied tool-paths generated in the top view and the printed result on the bottom. compensation disabled (left) and enabled (right). A clear reduction in visible gaps between extrusion at the edge of the elliptical top can be observed on the right part.

Figure 5.3 and 5.4 show a comparison between the printed object with compensation disabled and enabled, from a slightly angled view and a side view, respectively. The projection error in the left part is clearly visible between the last four extrusions at the edge, with gaps as small as $0.1mm$ still visible in the printed part.

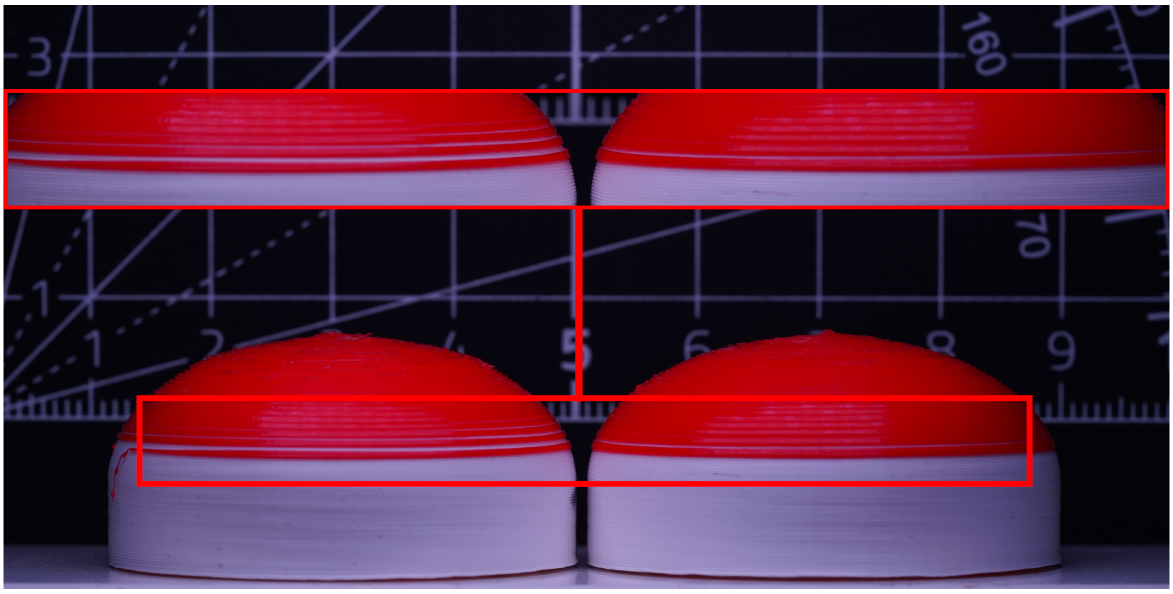


Figure 5.4: Side-view of compensation print comparison
Both printed object are shown in a side-view: compensation disabled (left) and enabled (right). A clear reduction in visible gaps between extrusion at the edge of the elliptical top can be observed on the right part.

When compensation is enabled, the gaps between extrusions at the edge are significantly reduced, as seen in the side view in Figure 5.4, with only a tiny gap remaining between the last two extrusions. However, because the compensation process has a limited adjustment range, it couldn't completely eliminate the projection error, as the initial gap between the extrusions was too large. As mentioned in section 4.3.5 the compensation caps the maximum extrusion width at $1.5 \cdot base\_extrusion\_width = 1.5 \cdot 0.5mm = 0.75mm$, meaning that the maximum gap that can be compensated for is $0.25mm$, with the total distance between extrusion points being $0.75mm$.

The projection error of two points, with a distance of $0.5mm$, with a fixed projection vector can be plotted against an increasing surface slope, as shown in Figure 5.5. The plot shows that the projection error increases exponentially with the surface slope, doubling the distance between points at around $60°$. Since a base extrusion width of $0.5mm$ is used in the test prints, the maximum gap that can be compensated for is $0.25mm$. A projection error of $0.25mm$ occurs at a slope of approximately $45°$, meaning any steeper slope will result in gaps between extrusions, even with compensation enabled.

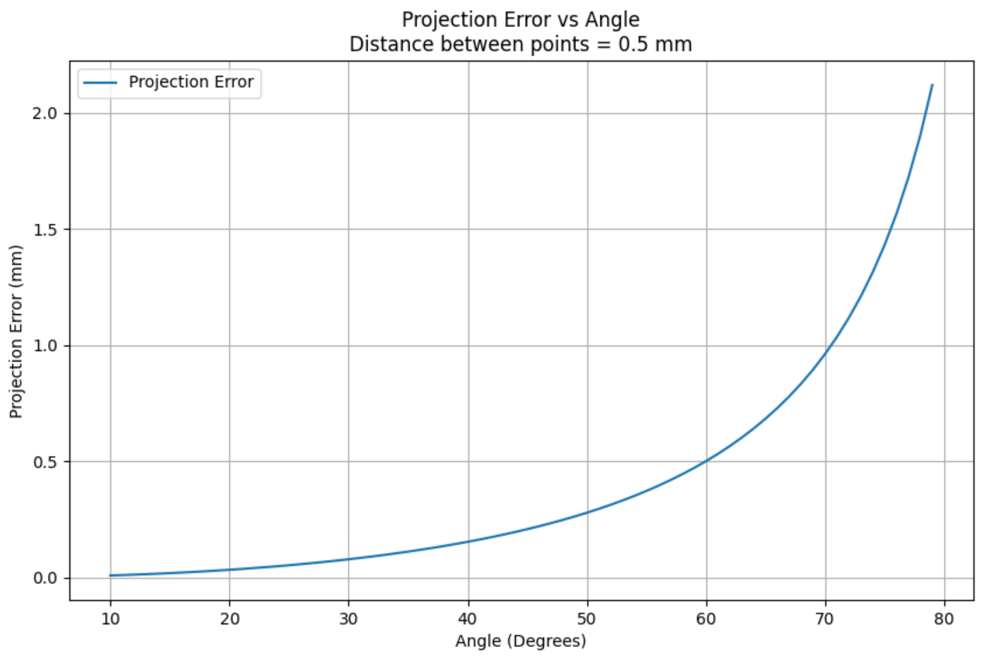


Figure 5.5: Projection error vs surface slope
This plot shows the projection error of two points with a fixed projection vector, spaced 0.5 mm apart, measured in mm in relation to the surface slope in degrees. The slope is modeled as a linear function, with $\tan(\theta) = m$, representing a small section of the surface. The projection error increases exponentially with increased steepness of the surface. With 60° marking the point where the distance between the points is doubled.

The results have shown that the compensation process can visibly reduce the projection error, being most effective on surface slopes up to 45° and less effective on steeper slopes. Therefore, selecting an optimal projection vector is crucial for an effective compensation process.

However, choosing an optimal projection vector depending on the boundary curve and surface is not always possible, in some cases leading to severe projection errors. That would require more work in the general projection approach and is further discussed in section 5.2.2.

Additionally, due to the way the compensation is handled, the tool-paths may exibit a form of uneven uniformity in parallel extrusions. Although this issue wasn't visible in the test prints, it remains a limitation, that will be highlighted in section 5.2.3.

5.2 Software Limitations

While the current implementation is a functional proof-of-concept and can be used as such, there are key limitations that negatively affect the usability. These stem from several design choices made during the thesis, that were necessary to keep the scope manageable. Each of the following limitation should be addressed in future work to improve the current implemenation and increase its usability.

5.2.1 Surface Area Delimitation

In the current implementation, the user can only select a single boundary curve to define the surface area for printing. While sufficient for simple designs, it lacks the flexibility required for more complex designs, such as surfaces with holes. Ideally, the user should be able to select one outer-boundary curve to define the general surface area and multiple inner boundary curves, which results in the generation of a space-filling pattern that fills the area in between the boundary curves. Figure 5.6 shows an example with two boundary curves and contrasts this with the current implementation.

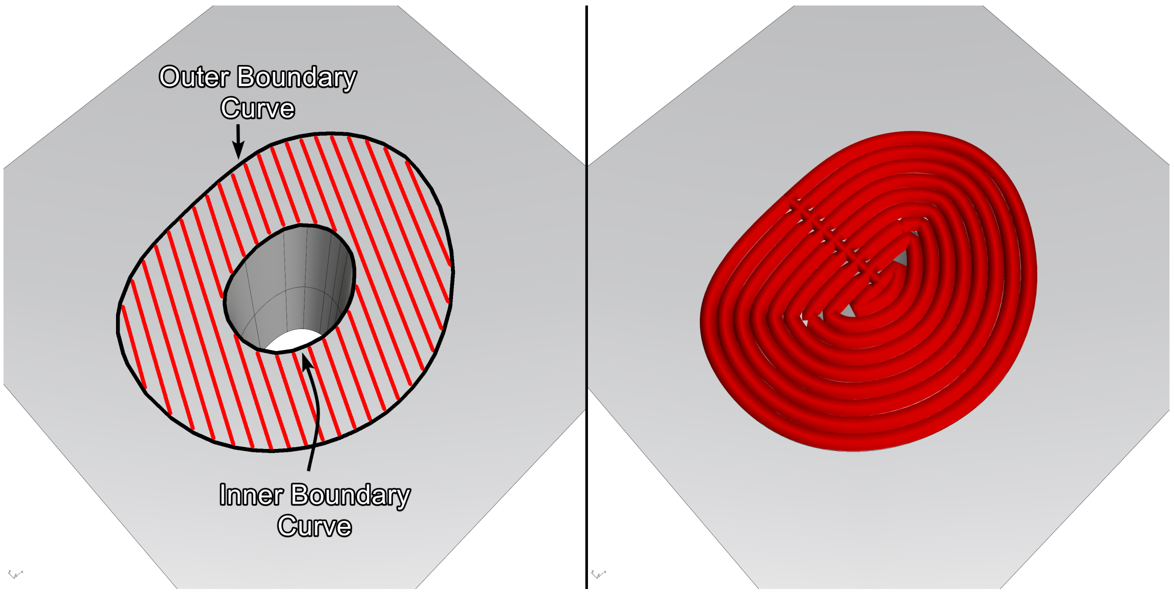


Figure 5.6: Surface delimitation with multiple boundary curves
Ideally, the user should be able to select multiple boundary curves to define the surface area for printing, allowing for more complex designs, such as surfaces with holes. The **left image** shows two boundary curves (black) and the area to be filled (red). The **right image** highlights the current limitation, where tool-paths that circumvent the hole cannot be generated, as the implementation is built with only one boundary curve in mind.

Implementing this feature would require more sophisticated algorithms capable of generating space-filling patterns that can handle multiple boundary curves at once. This could be achieved by using third-party libraries, such as Clipper [21], that are specifically designed for polygon clipping and offsetting. Rather than relying on in-built Rhino functions, that were not initially designed for the purpose of generating space-filling patterns.

5.2.2 Projection Approach

The current implementation uses a simple projection approach that first projects the entire boundary curve onto a plane, with the projection vector calculated based on surface normals along the boundary curve as explained in section 4.3.1.

While this approach is simple and effective for cases, where the general direction of the boundary curve is generally uniform, it fails when parts of the area enclosed by the boundary curve face in vastly different directions. This results in a projection plane that cannot be parallel to the entire enclosed surface area, exacerbating the projection error at certain parts, as shown in figure 5.7, that can no longer be compensated for by simply adjusting the extrusion widths and needs a more complex solution.

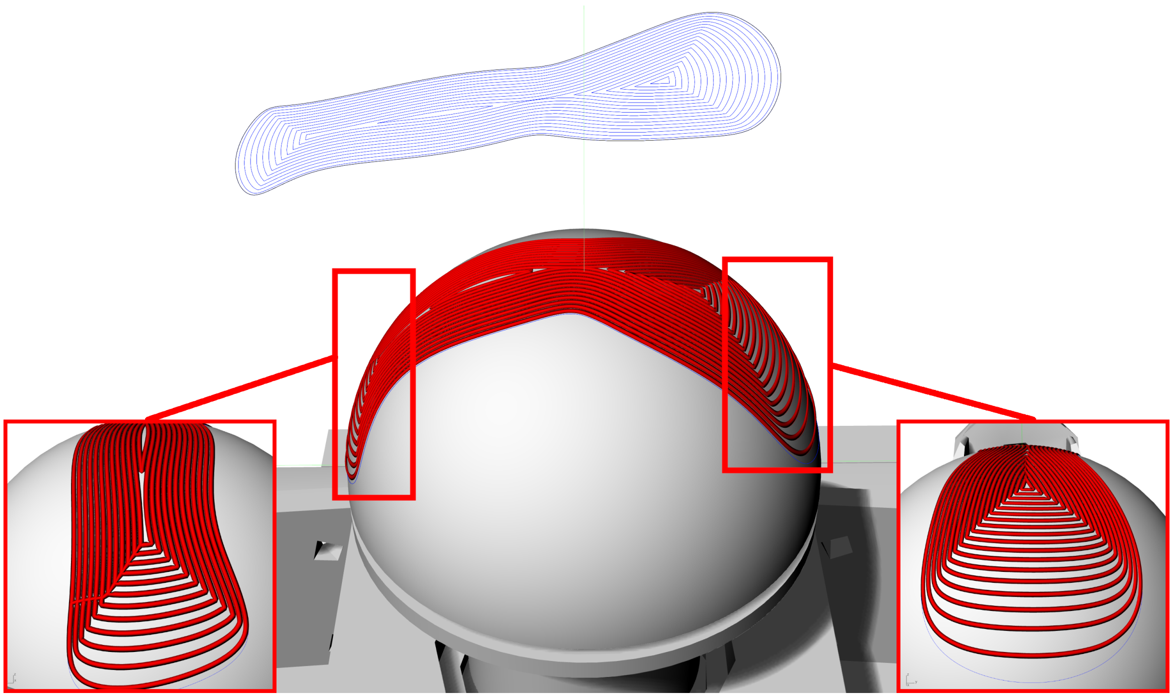


Figure 5.7: Extrem projection error
The projection error becomes more pronounced, when the boundary curve spans over a large surface area, where the general direction for certain parts of the curve go in completely different directions. Since the calculated projection plane is cannot be parallel to the entire surface area enclosed by the curve at once.

A possible solution to this problem would be to automatically subdivide the original boundary curve into smaller section, so that each section faces in a more uniform direction. Then for each subdivided section, the entire implementation pipeline can be run to generate several tool-paths that combinded fill the orginally enclosed surface area more accurately, as shown in figure 5.8.

While this approach is applicable, it is also more complex and several challenging problems must be solved. These include determining how to subdivide the original boundary curve and how to effectively combine the generated patterns into a single tool-path that minimize visible artifacts, that occur as a result of the subdivision.

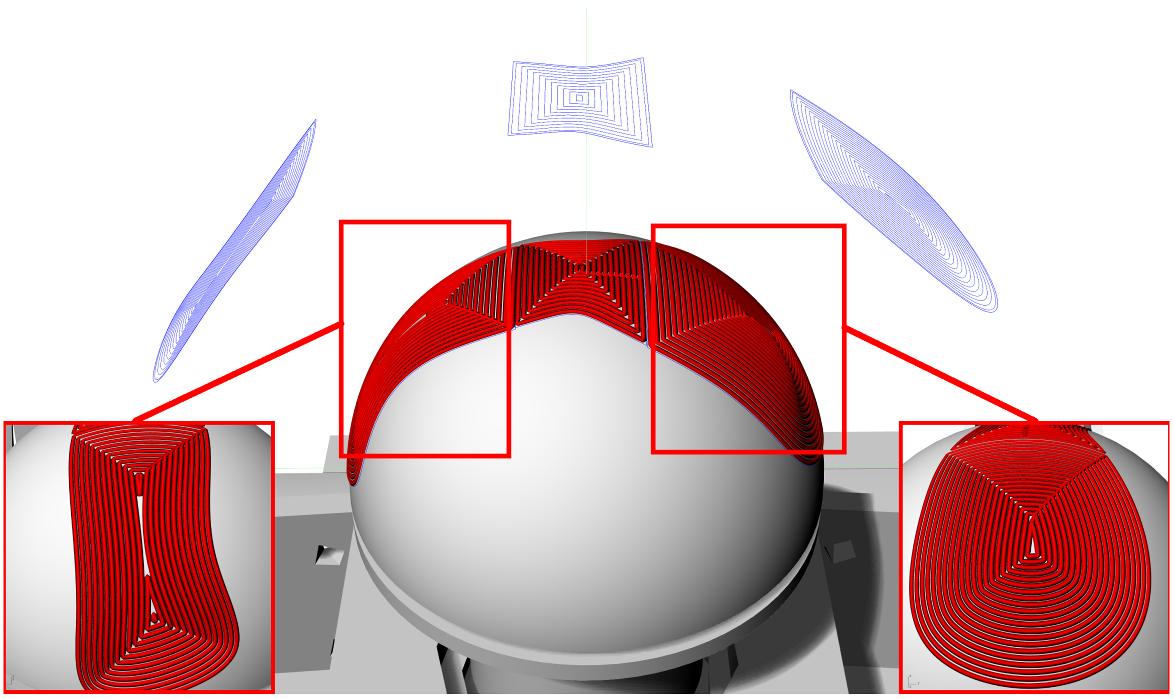


Figure 5.8: Subdividing boundary curve for projection
As a possible solution to the problem shown in Figure 5.7. The original boundary curve is subdivided into three smaller sections. Each section faces in a more uniform direction, leading to better results when applying the implementation pipeline to each subdived section individually, and significantly reducing the projection error.

5.2.3 Compensation

The current implementation uses a simple variable extrusion width compensation approach to counteract projection errors. This is done by iteratively adjusting each curve, with each one relying on its locally defined neighbour to determine its extrusion width, trying to maximize it to fill any gaps. While this greedy approach works, as shown in section 5.1.2, This approach can lead to visibly unpleasing uniformity in the printed part, being especially undesirable when the goal is a good surface finish.

During the compensation process, when the previous curve increased its extrusion width to the maximum allowed, three outcomes are possible for the next curve: The curve is far enough from its neighbour that even when increasing its extrusion width to the maximum, it cannot fill the gap. The curve is just close enough to its neighbour, that the extrusion width must be increased to fill the gap. Or the curve is so close to its neighbour, that the gap can only be filled by decreasing the extrusion width. Ideally, the compensation process alternates between the second and third outcome, as this achieves the highest fill rate.

However, in some cases, this can cause extrusions to alternate between very wide and very thin extrusions, negatively impacting the uniformity of the tool-path and, as such the final surface finish. The effects of this can be seen in figure 5.9.

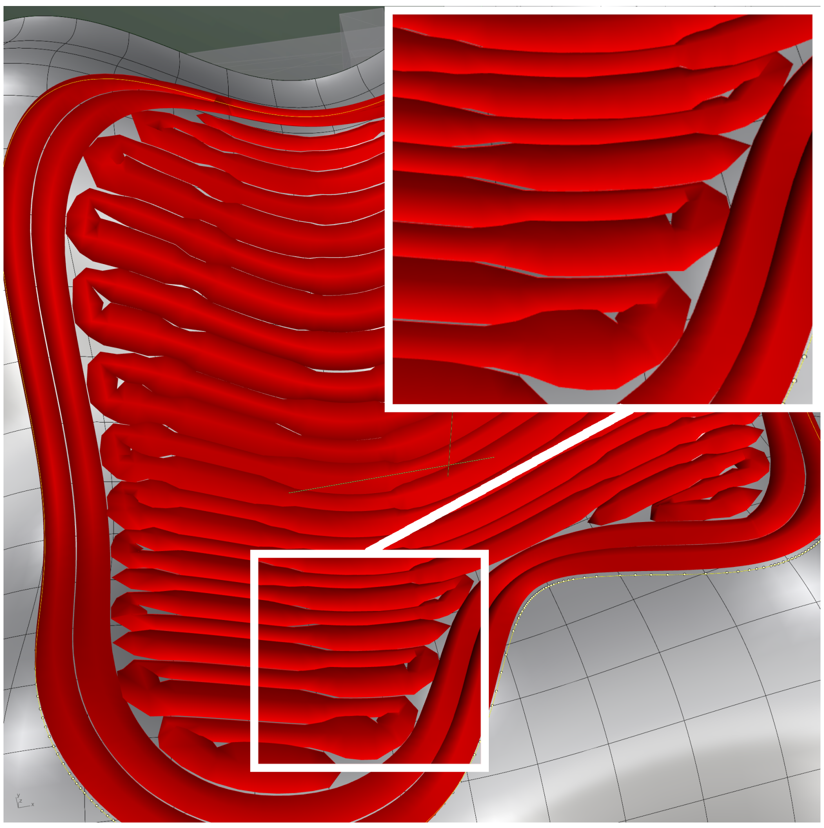


Figure 5.9: Compensation artifacts
Due to the greedy approach used in the compensation process, parallel extrusions may alternate between very wide and very thin extrusions, leading to undesirable uniformity in the printed part.

Conclusion 6

Neo5x has successfully been extended to support the generation of tool-paths for conformally filled, freeform layers on surfaces of printed parts.

To achieve this, a new CAM-Block type was introduced, allowing users the to be define printed area by first selecting the surface and a closed curve on it. The implementation uses projection to generate the necessary input curves on the surface, which the Neo5x interface requires to generate 5-axis tool-paths.

The boundary curve is first projected onto a plane that is constructed by the averaged normal vectors of the surface along the boundary curve. This flattens the boundary curve and represents the enclosed surface area in 2D and makes it easier to generate the required space-filling pattern. Currently supports two types of space-filling patterns that a user can choose from, a Contour-Parallel or a Zig-Zag pattern. The generated pattern is then projected back onto the surface, in a space-filling pattern that fills the original bounded area on the surface. Due to the projection process, projection errors are introduced that causes the distance between the curves to increase, depending on the curvature of the surface. This will cause the tool-path to not correctly fill the bounded area. To reduce the effects of this, a variable extrusion width compensation is introduced, that iteratively adjusts the extrusion width of every curve. This is done in a greedy manner, where the extrusion width is increased or decreased based remaining gap between the curve and its neighbour.

6.1 Future work

The results presented in section 5.1 have shown promising results as a proof of concept. Key limitations mentioned in section 5.2 still need to be addressed to increase the usability of the implementation. The most critical limitation is the projection error, which is the primary source of issues in the implementation. One proposed solution, discussed in section 5.2.2, involves subdividing the boundary curve into smaller segments, allowing each segment to be projected more favorably. Depending on the subdivision, this approach could significantly reduce the projection error for each individual segment. Reducing the projection error would also increase the effectiveness of the compensation, as now only smaller gaps, that are within the adjustment range, need to be closed. If such an approach would be implemented, it would necessitate a more sophisticated way of generating space-filling pattern, as the sub-divided parts would need to be connected in the final tool-path to guarantee smooth transitions between segments.

The compensation used is also a limiting factor that could be improved, currently the range of adjustment is limited to 45° surface slopes at a fixed projection angle. Depending on the surface and boundary curve, even with compensation enabled, tool-paths may still fail to completely fill the bounded area. The effective adjustment range can not be increased due to physical limitations of extrudable material, which other unwanted artifacts. To address this, an added feature in the compensation process would be desirable. That additionally to adjusting the extrusion width, could close gaps by adding new extrusions, in areas where the the extrusion width cannot further be increased.

Additionally, the current compensation method, which greedily focuses only on the local gap between two curves, can introduce undesirable artifacts in the tool-path. These artifacts manifest as alternating wide and thin parallel extrusions. To address this, a new approach would need to be implemented, that considers the global scope of the tool-path. Which ensures that gaps are closed effectively without introducing such artifacts.

Bibliography

[1] Li, A., Li, L., Liu, Y., Cui, B., Li, Y., and Wang, X.: Research on Curved Layer Slicing and Spatial Path Generation Method in Five-axis Material Extrusion Process. *Journal of Physics: Conference Series* 1884, 012013 (2021)

[2] Jensen, M., Mahshid, R., D'Angelo, G., Walther, J., Kiewning, M., Spangenberg, J., Hansen, H., and Pedersen, D.: Toolpath Strategies for 5DOF and 6DOF Extrusion-Based Additive Manufacturing. *Applied Sciences* 9, 4168 (2019)

[3] Ottonello, E., Hugron, P.-A., Parmiggiani, A., and Lefebvre, S.: QuickCurve: revisiting slightly non-planar 3D printing. *test* (2024)

[4] Ahlers, D., Wasserfall, F., Hendrich, N., and Zhang, J.: 3D Printing of Nonplanar Layers for Smooth Surface Generation. In *2019 IEEE 15th International Conference on Automation Science and Engineering (CASE)*, 1737–1743 (2019)

[5] Ramos, B., Pinho, D., Martins, D., Vaz, A. I. F., and Vicente, L. N.: Optimal 3D printing of complex objects in a 5–axis printer. *Optimization and Engineering* 23, 1085–1116 (2022)

[6] Yamin, L., He, D., Wang, X., and Tang, K.: Geodesic Distance Field-based Curved Layer Volume Decomposition for Multi-Axis Support-free Printing. *test* (2020)

[7] Dai, C., Wang, C. C. L., Wu, C., Lefebvre, S., Fang, G., and Liu, Y.-J.: Support-free volume printing by multi-axis motion. *ACM Trans Graph* (2018)

[8] Elkaseer, A., Müller, T., Rabsch, D., and Scholz, S.: Impact of Nonplanar 3D Printing on Surface Roughness and Build Time in Fused Filament Fabrication. *test* 285–295 (2020)

[9] Emon, O. F., Alkadi, F., Kiki, M., and Choi, J.-W.: Conformal 3D printing of a polymeric tactile sensor. *Additive Manufacturing Letters* 2, 100027 (2022)

[10] Hong, F., Lampret, B., Myant, C., Hodges, S., and Boyle, D.: 5-axis multi-material 3D printing of curved electrical traces. *Additive Manufacturing* 70, 103546 (2023)

[11] Associates, R. M. .: Rhinoceros 3D. https://www.rhino3d.com/ (2024)

[12] e3d online: Research and Development: Motion System and Tool-Changer. https://e3d-online.com/blogs/news/research-and-development-motion-system-and-tool-changer (2018)

[13] Hong, F., Hodges, S., Myant, C., and Boyle, D.: Open5x: Accessible 5-axis 3D printing and conformal slicing (2022)

[14] FreddieHong19: Open5x Github Repository. https://github.com/FreddieHong19/Open5x (2024)

[15] Schmolzi, T.: Calibrating a Low-cost, 5 Axis 3D Printer (2024)

[16] Rodriguez-Padilla, C., Cuan-Urquizo, E., Roman-Flores, A., Gordillo, J. L., and Vázquez-Hurtado, C.: Algorithm for the Conformal 3D Printing on Non-Planar Tessellated Surfaces: Applicability in Patterns and Lattices. *Applied Sciences* 11 (2021)

[17] Zhang, T., Fang, G., Huang, Y., Dutta, N., Lefebvre, S., Kilic, Z. M., and Wang, C. C. L.: S3-Slicer: A General Slicing Framework for Multi-Axis 3D Printing. *ACM Trans Graph* 41 (2022)

[18] Gálvez, A., Iglesias, A., and Puig-Pey, J.: Computing parallel curves on parametric surfaces. *Applied Mathematical Modelling* 38, 2398–2413 (2014)

[19] Can, A. and Ünüvar, A.: Five-axis tool path generation for 3D curves created by projection on B-spline surfaces. *The International Journal of Advanced Manufacturing Technology* 49, 1047–1057 (2010)

[20] Cox, J. J., Takezaki, Y., Ferguson, H. R., Kohkonen, K. E., and Mulkay, E. L.: Space-filling curves in tool-path applications. *Computer-Aided Design* 26, 215–224 (1994). Special Issue:NC machining and cutter-path generation

[21] AngusJohnson: Clipper2 Github Repository. https://github.com/AngusJohnson/Clipper2 (2024)

[22] Isa, M. A. and Lazoglu, I.: Five-axis additive manufacturing of freeform models through buildup of transition layers. *Journal of Manufacturing Systems* 50, 69–80 (2019)

[23] Wang, X., Bai, Q., Gao, S., Zhao, L., and Cheng, K.: A Toolpath Planning Method for Optical Freeform Surface Ultra-Precision Turning Based on NURBS Surface Curvature. *Machines* 11 (2023)

[24] Liu, T., Zhang, T., Chen, Y., Huang, Y., and Wang, C. C. L.: Neural Slicer for Multi-Axis 3D Printing. *ACM Trans Graph* 43 (2024)

Eidesstattliche Versicherung

Hiermit versichere ich an Eides statt, dass ich die vorliegende Arbeit im Bachelorstudiengang Informatik selbstständig verfasst und keine anderen als die angegebenen Hilfsmittel – insbesondere keine im Quellenverzeichnis nicht benannten Internet-Quellen – benutzt habe. Alle Stellen, die wörtlich oder sinngemäß aus Veröffentlichungen entnommen wurden, sind als solche kenntlich gemacht. Ich versichere weiterhin, dass ich die Arbeit vorher nicht in einem anderen Prüfungsverfahren eingereicht habe.

22.08.24, Hamburg	P. Bartel
Ort, Datum	Unterschrift