Department of Mathematics and Computer Science
Algorithms

An Improved Partitioning Algorithm for 3D Printing

Master's thesis

Pieter van Beerendonk

August 28th, 2024

Supervision:
Prof. Dr. Mark de Berg

Assessment committee:
Prof. Dr. Mark de Berg
Dr. Thomas van Dijk
Maxime Chamberland

Credits: 30

This is a public Master's thesis.

This Master's thesis has been carried out in accordance with the rules of the TU/e Code of Scientific Conduct.

Abstract

The growing use of 3D printing, both in the consumer market and in industry is greatly changing traditional manufacturing, creating new opportunities for geometry-focused algorithmic research. Large models cannot always be produced in one part using 3D printing techniques due to the limited size of the printing volume of 3D printers, thus it is necessary to partition a model into pieces that each fit into the printing volume. Previous work used a recursive partitioning strategy, where in each step a whole plane is used to partition the model. This makes certain local cuts impossible to perform, possibly leading to many (small) parts. To overcome this restriction and maintain an acceptable runtime, we present a novel technique that partitions a model using local cuts. To assess possible use cases of the proposed technique, the quality of the cut, and runtime performance of both the original and proposed partitioning technique were measured on several models. Because the new technique proposed in this paper provides more freedom when cutting models, this additional freedom improves the quality of the final result for certain models.

Contents

Chapter 1

Introduction

1.1 Background and motivation

In recent years 3D printing (3DP), also called additive manufacturing (AM), has become more accessible and commonplace for end-users like consumers, industry, and healthcare providers. The price of consumer-grade 3D printers has dropped significantly while the quantity sold has increased dramatically [50]. Concurrently, consumer-grade printers have become more advanced, with improvements in speed and precision [51]. In industry, additive manufacturing has gained prevalence where it is, among other things, used for quick prototyping, manufacturing speciality parts, and constructing buildings [28, 43, 49]. Meanwhile, in healthcare, a 3D printed representation of a patient's teeth is used as a visual model during complex orthopaedic treatments [16], and much research is being done in bio-printing where human tissue is printed [24, 56], like for the corneal (thin outer layer of the eye) [30].

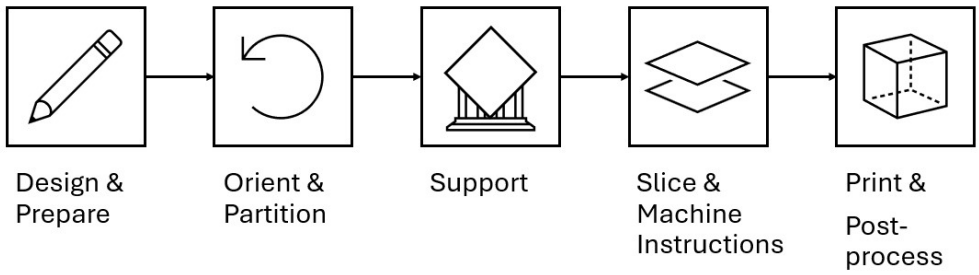


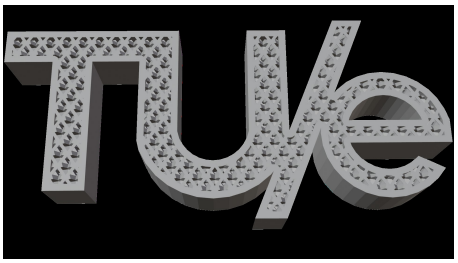
Figure 1.1: The 3D printing pipeline, as described below. A 3D model starts as a design, it is then oriented and partitioned. Next, the model is supported after which it is made ready to be printed. Finally, it is printed and post-processing steps are executed.

Additive Manufacturing Process. The additive manufacturing process [37], which creates a product from a set of design specifications, is quite involved as can be seen in the overview in Figure 1.1. Starting with a set of design specifications, a design has to be made using computer-aided design (CAD) software. Oftentimes, these designs have to be converted from their original format to the triangle-based STL format, which is the conventional format for 3D print-

ing applications. Models in STL format can have degeneracies that prohibit them from being printed and that need to be repaired. Such degeneracies include gaps in the topology [9, 44], faces without area, overlapping vertices or faces, non-manifold topology conditions [7, 12, 36], or thin features [13]. When the model is repaired, it can be hollowed by creating an inner cavity at a set distance from the model's exterior, aiming to strike a balance between material usage and the strength of the final product. Often, drain holes need to be made for material to escape these cavities.

The next step to printing the model is partitioning and orienting. Since 3D printers have a limited volume in which parts can be printed, called the *print volume*, they are unable to print certain models. To overcome this restriction or to reduce the amount of external supports required (see below), a model can be *partitioned* into parts such that each part fits in the build volume, the volume in which the printer can produce a part [33, 64]. Several factors affect the quality of the final model when it is reassembled from its parts. Firstly, the model will have less strength at the seams and needs to be reattached there so a large seam-surface area is advantageous. In addition, the placement of seams can affect the mechanical and aesthetic properties of the final model. Of course, it is crucial to reduce the number of parts used, as a single part can take several hours to produce. Next, *orienting* a model is a multi-objective optimization problem as a model's orientation has a great effect, including on the build time, volume of (sacrificial) supports used, chance of print failures [53, 67], location of the supports, surface finish, mechanical properties of the final model and finally on the accuracy of the model.

(a) Example of external supports.



(b) Example of internal supports (Body-centered cubic lattice structure [23]), the model was cut open to show them.

Figure 1.2: Example of internal and external supports. In these examples, specialized in resin 3D printing. Supports generated in Anycubic Photon Workshop v3.1.4 [4].

After the model has been partitioned and oriented, it can be placed in the build volume and supports can be added. Supports can be distinguished into two categories: internal, and external. *internal supports*, also called infill, aim to increase rigidity and change the local density of hollowed models. Infills can be divided into three categories: dense, sparse, and microstructures. Dense infills follow a pattern, for example, zig-zag [41], contour parallel pattern [26, 32, 66], are more likely to trap materials (like resin or powder) due to its density and are prominently used to support overhangs. Meanwhile, sparse infills use less material than their dense counterpart and are less likely to trap materials. Examples include a hatching pattern [18], a pattern optimized for mechanical stress [54], a pattern based on rhombic cells [65], and

a frame structures [62]. Finally, microstructure infills affect the rigidity of the printed part by using a complex internal structure [46]. Next, *external supports* are used to enable material deposition, to avoid movement, and deformation. Most 3D printing techniques [6, 2, 37] are unable to print parts that are not supported by either the printer or previously printed layers, this is particularly the case for techniques that rely on material deposition. For most techniques, it is important to support the weight of the part or provide an escape for access heat. The placement of external supports is relevant to the surface finish; support structures can damage the surface of the final part where the supports and the part touch. Generating supports requires detecting where supports are required, for example, places where steep angles, overhangs, and points are unsupported [3, 10, 29].

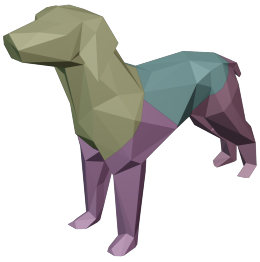
After supports have been placed, the 3D model can be converted to slices. A *slice* is defined by the contour of the model at some horizontal plane, where the contour is given a set thickness. A representation of the model is made, using such slices at regular intervals over the vertical length of the model. The thickness of the slice influences the accuracy of the final model. This thickness is often constant for the whole model, however, techniques allow to vary the thickness of separate slices [19, 55]. It is even possible to vary thickness within a given slice [27, 60, 61, 63]. A more detailed overview of slicing is provided by Pandey et al. [42]. Several techniques have been proposed to find the contour of a given slice: using simple plane intersection [34], using the connectivity of the mesh [40], using incremental construction by traversing the triangles [68], or using a ray-based approach [35, 45, 69]. Finally, machine instructions are generated from the slices and the part is printed. Then, external supports may need to be removed and other post-processing steps may need to be performed.

For a more detailed overview of the Additive Manufacturing Process, please refer to the excellent overview paper by Livesu et al. [37].

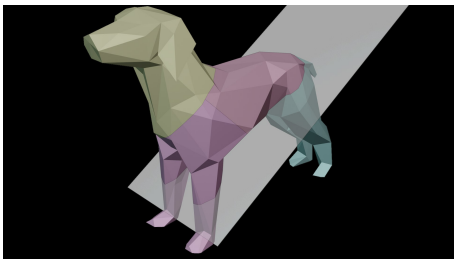
Figure 1.3: Example of the build volume for a resin-based 3D printer. The printing volume is usually a rectangular box, defined by given dimensions $w \times d \times h$ (image from 3dgadgets [1]).

The Partitioning Problem. As just discussed, the whole 3D printing process involves several steps, which each involve algorithmic challenges. In this thesis, we focus on the partitioning step, which is necessary when the model is too large to fit into the print volume. Next, we define this *partitioning problem* more formally. Given a 3D model M and a printing volume V, subdivide M into parts such that each part P_i fits in V and such that $\cup_i P_i = M$ and $\cap_i \text{Int}(\text{P}_\text{i}) = \emptyset$, where $\text{Int}(P_i)$ denotes the interior of part P_i, i.e. the parts can be printed and the parts exactly combine into the model. Such a subdivision of M is then called a partition (denoted with $\mathcal{P}$). Often, a set of objectives needs to be optimized for. Such objectives include: the number of parts used, objectives regarding the aesthetics of the final partition, objectives regarding structural integrity, and objectives to guide the size of parts and the size of connections between parts. Here, let a *3D model* be a 3-dimensional, watertight mesh defined by vertices, edges and faces that encloses a volume, and let a *(3D) printing volume* be the volume in which a 3D printer can print a 3D model. Typically, the printing volume is a rectangular volume of given dimensions $w \times d \times h$.

Given these definitions, we can define the partitioning problem more formally as: Given a model M, a print volume V, and objective function f, find a valid partition $\mathcal{P} = \{P_1, \ldots, p_k\}$ of the model M such that each P_i fits into V and $f(\mathcal{P})$ is minimized. Often, we require a partition to optimize for a set $\mathcal{F}$ of objective functions with weights for each f_i in F, thus we can define the objective function as $f(\mathcal{P}) = \sum_{f_i \in F} \alpha_i \cdot f_i(\mathcal{P})$.



(a) The partition using the original algorithm by Luo et al.



(b) The partition using the proposed algorithm.

Figure 1.4: The algorithm by Luo et al. cannot cut the hind legs of the dog without cutting both legs front, while the technique proposed in this thesis is able to perform such a cut.

1.2 Contributions and organization

While much research has been done into other algorithmic problems concerning 3D printing, the problem of partitioning a model has been less discussed in literature. Of particular interest here is the paper by Luo et al. [38], which provides a solution based on greedy planar cuts, optimizing for several desirable objectives. However, these planar cuts can force undesirable cuts like in Figure 1.4 where cutting the hind legs of the dog would force a cut through its front legs. Using our proposed technique, this is no longer necessary.

The proposed technique will iteratively partition a 3D model into parts that can be printed. In each iteration, the algorithm will generate a selection of planar cuts and proceed with a

subset of the best cuts. Contrary to the technique by Luo et al., the proposed technique will not cut at every intersection of the cutting plane and the model. Instead, the proposed technique will cut only on a subset of intersections such that the resulting partition contains exactly two parts. The technique by Luo et al. is laid out in more detail in Chapter 2, while the proposed approach is detailed in Chapter 3.

Next, this paper will compare both techniques in Chapter 4 and highlight cases, such as the one shown above in Figure 1.4, where the proposed technique works well; or cases where the original technique by Luo et al. performs better. During testing, it was concluded that the proposed technique although slower has use cases where cuts are performed that were not possible using the original algorithm, leading to slightly better results in a limited amount of cases. Finally in Chapter 5, concluding remarks and possible further improvements will be discussed.

Chapter 2

Partitioning Technique by Luo et al.

In this chapter, we will discuss the partitioning technique by Luo et al. [38]. This paper introduces a framework based on a BSP (Binary Space Partition) tree and objective functions to partition a 3D model, such that the resulting parts fit in the printing volume. The technique employs a set of objective functions to guide the quality of the final product, as described in the problem definition in Section 1.1. These objective functions will be discussed in Section 2.1.

The greedy algorithm introduced by Luo et al. iteratively partitions a 3D model, each iteration considering many possible cuts and then reducing the number of options to the top b best partitions until each resulting part fits in the build volume. The algorithm will be discussed in detail in Section 2.2, after which limitations of the technique will be discussed in Section 2.3.

2.1 Objective Functions

The algorithm uses several metrics to determine the quality of a partition during calculation (based on a BSP tree, see Section 2.2). For each possible cut the algorithm will consider the objective score, which is calculated as a weighted sum of the objective functions. This objective score is then used to determine which partitions will be used in the next round. The objective functions can also be employed to quantify the quality of a completed partition.

Let us define $f(\mathcal{P}) = \sum_{f \in F} \alpha_f \cdot f(\mathcal{P})$ as the weighted sum of all objective functions f with weight α_f for some partition $\mathcal{P}$. If some objective function $f_i(\mathcal{P}')$ equals infinity, then this partition $\mathcal{P}'$ is considered infeasible and should be discarded. Let us briefly discuss each objective function used; for more details, we refer you to section 3.2 of the paper by Luo et al. For each objective function, its formula is given and explained, and some reasoning is provided.

- First, we consider the *Number of Parts*. To estimate the quality of a given partition $\mathcal{P}$, for which not all parts fit in the build volume, the algorithm will calculate an upper bound $\mathcal{O}(P)$ on the number of build volumes needed to print the part P. Exactly calculating the number of build volumes needed is NP-hard as the 2D version of this problem, namely determining the number of squares required to cover a bigger square, is NP-hard as well. This upper bound $\mathcal{O}(P)$ is given for a part P in partition $\mathcal{P}$, by the number of printing volumes needed to contain that part, when these volumes are simply arranged in a grid. To normalize the $\mathcal{O}(P_0)$ denotes the upper-bound estimate for the initial model P_0.

$$f_{\text{part}}(\mathcal{P}) = \frac{1}{\mathcal{O}(P_0)} \sum_{P \in \mathcal{P}} \mathcal{O}(P) \tag{2.1}$$

- Second, the *Utilization* will force parts to maximize usage of the printing volume of the 3D printer, a cube described by non-negative height, width, and depth with non-negative volume V. Here $\text{Vol}(P)$ is the volume of a part P in partition $\mathcal{P}$. This objective function will penalize the smallest part of the partition and will favour large parts.

$$f_{\text{util}}(\mathcal{P}) = 1 - \min_{P \in \mathcal{P}} \frac{\text{Vol}(P)}{\mathcal{O}(P)V} \tag{2.2}$$

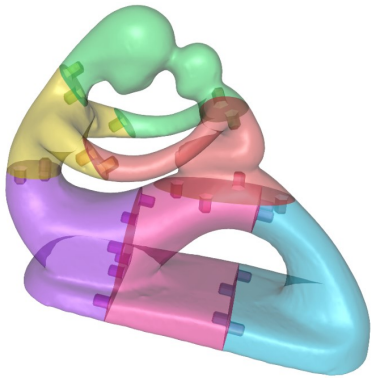


Figure 2.1: Example of connectors which aim to connect two parts of a model, keeping it together. The connectors are the small protrusions on the seams (taken from Luo et al. [38]).

- Next, connectors should be added to keep the parts together as shown in Figure 2.1, for this *Connector Feasibility* is determined. For each connected component G in the cross-section of a cut C, we compare the ratio between the area where connectors can be placed, a_G, and the total area of the connected component A_G. As the area where connectors can be placed approaches zero, the objective function will approach infinity. The rationale behind this measure is that large areas of connected parts make these parts more likely to adhere to each other. $c_{connector}$ is set to 10, to reduce the influence of this objective function.

$$f_{\text{connector}}(\mathcal{P}) = \max\left(\max_{G \in C \in \mathcal{P}} (A_G/a_G - c_{\text{connector}}), 0\right) \tag{2.3}$$

- For *Structural Soundness*, two separate statistics can be considered, namely Structural Analysis and Fragility. First, if a user supplies the gravity direction of the object and its density, then *structural analysis* can be done on the input model based on tension T_t and shear T_s stress on the object, which are calculated from a pre-calculated stress tensor. The objective will then encourage cuts away from areas of high stress. A_C is the area of some cut C, α_t and α_s are constants.

$$f_{\text{structure}}(\mathcal{P}) = \sum_{C \in \mathcal{P}} \frac{1}{A_C} \int_C (\alpha_t T_t(x) + \alpha_s T_s(x)) dA \tag{2.4}$$

Second, the *fragility* objective aims to avoid cuts that leave thin parts of the model that connect two larger parts. To find such parts, first the algorithm finds a set S with vertices v from the mesh of the part whose normal n_v is nearly parallel to the cut plane π with normal n_π. More precisely, if $n_v \cdot n_\pi > c_{fragility\_n}$ where $c_{fragility\_n} = 95$. Next, a ray tangent to π is cast from v to π. If the distance from v to π is less than $c_{fragility\_d}$ and the ray does not intersect the mesh, then v is added to $S_{fragility}$. $c_{fragility\_d}$ is chosen to be 1.5 times the connector depth, thus ensuring that connector placement is possible.

$$f_{\text{fragility}}(\mathcal{P}) = \begin{cases} 0 & \text{if } S_{\text{fragility}} = \emptyset \\ \infty & \text{otherwise} \end{cases} \tag{2.5}$$

- For the *Aesthetics* objectives we consider seam obstructiveness and symmetry. *Seam Obstructiveness* aims to place seams on texture edges, in less visible places or away from user-indicated areas. The penalty is computed based on the cost $\epsilon(C)$ of the given cut C as defined below and on the expected cost for the parts that still need to be cut. $\epsilon(C)$ is expressed as the average penalty over the seam, divided by the length d_0 of the diagonal of the bounding box of the initial model. $\rho(p)$ is the penalty (for example ambient occlusion) for a point p on the seam of C.

$$\epsilon(C) = \frac{length(C) \cdot \sum_{p \in C} \rho(p)}{|C| \cdot d_0} \tag{2.6}$$

The final objective is then the sum of the cost of all cuts C in partition $\mathcal{P}$, plus the expected cost for all parts P that do not yet fit in the printing volume, i.e. where $\mathcal{O}(P) > 1$. For the expected cost $\hat{\epsilon}(P)$ for a part P, the paper found that the cost of the most recent seam on the part is a good estimate.

$$f_{\text{seam}}(\mathcal{P}) = \frac{1}{\mathcal{O}_0} \left(\sum_{C \in \mathcal{P}} \epsilon(C) + \sum_{P \in U \subseteq \mathcal{P}} (\mathcal{O}(P) - 1) \hat{\epsilon}(P) \right) \tag{2.7}$$

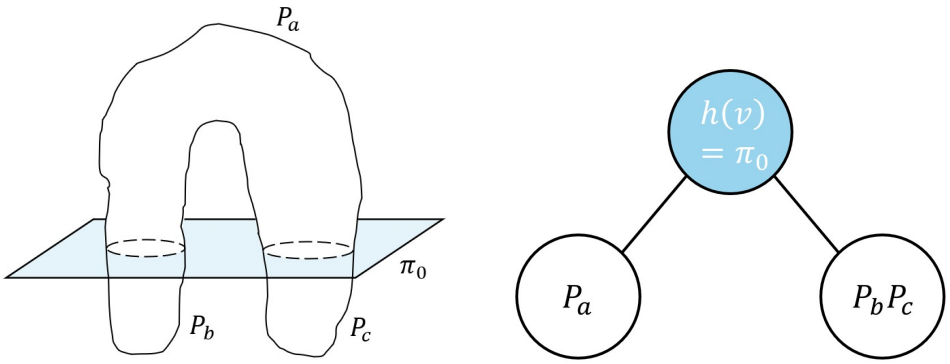
Finally, the *Symmetry* objective is based on how well the cut follows the symmetry of the original model. For this, we initially calculate the symmetry plane of the model, π_{symm}. Next, for each cut plane C of a given partition $\mathcal{P}$ points p are uniformly sampled. Next, the Root-Mean-Square (RMS) distance of that point and the closest point on the symmetry plane π_{symm} is calculated. This is then normalized by d_0, the diagonal length of the original bounding box.

$$f_{\text{symmetry}}(\mathcal{P}) = \frac{1}{d_0} \cdot \underset{pC \in \mathcal{P}}{\text{RMS}} \left(\min_{q \in C \in \mathcal{P}} \|p - \text{reflect}(q, \pi_{symm})\| \right) \tag{2.8}$$

2.2 The Partitioning Algorithm

In this part, we will discuss the data structure underlying the algorithm by Luo et al., the BSP tree, to then discuss the actual algorithm.

BSP trees. Let us first consider the underlying data structure used in the algorithm, a *BSP tree*. A (3-dimensional) *binary space partition* (or *BSP* for short), as first introduced by Fuchs et al. [22], is a recursive partitioning of $\mathbb{R}^3$ using planes to represent a partition of a 3D model. A BSP can be described by a *BSP tree*. Each internal node v stores a splitting plane $h(v)$. The splitting planes recursively divide $\mathbb{R}^3$ into regions, where the root v_r divides all of $\mathbb{R}^3$ while the left and right children of the root respectively divide the region on the positive and negative side of the plane $h(v_r)$. Thus each leaf of the tree defines a single part P, which can be found by traversing the tree.



(a) A model cut into three parts (P_a, P_b, and P_c) using plane π_0.

(b) The BSP tree as a result of cutting the model using plane π_0.

Figure 2.2: Cutting the model shown in the image using plane π_0 results in three parts, resulting in two internal regions. The BSP hence contains one internal node with $h(v) = \pi_0$ and two leaves representing the parts P_a, P_b, and P_c.

The algorithm uses a BSP tree since it is more time-efficient to only consider planar cuts, instead of arbitrary cuts. In addition, a BSP tree satisfies two objectives by construction, namely printability and assemblability. Printability is maintained in the outer loop of the main algorithm, where a part is divided into two parts and eventually the loop terminates if all parts fit in the volume. Assemblability is maintained because the model can be assembled in reverse order in which it was cut.

The Algorithm. As we have introduced BSP trees, the algorithm by Luo et al. will be discussed. The algorithm will maintain a collection $\mathcal{B}_{\text{curr}}$ of at most b BSP trees, representing different options for partitioning the model. More precisely, each BSP represents a (partial) partitioning, which may still need to be refined further as not all parts fit in the printing volume yet. The pseudocode for the algorithm can be found in Algorithm 1, the steps of this algorithm will be discussed below.

The algorithm aims to iteratively improve BSP trees in $\mathcal{B}_{\text{curr}}$ that contain parts that do not yet fit in the build volume (line 3), here $\mathit{parts}(\mathcal{P})$ for some partition $\mathcal{P}$ indicates the parts defined by the partition. To do this, it at most b (partial) BSP trees, in $\mathcal{B}_{\text{curr}}$. This b is called the beam width and limits the amount of BSPs that are considered at one time.

To improve a BSP tree $\mathcal{P}$, the algorithm finds the part P which is expected to require the most build volumes to be printed, according to the upper bound as given by $\mathcal{O}(P)$. $\mathcal{P}$ is removed from $\mathcal{B}_{\text{curr}}$ (line 6), and then a set of BSP trees are proposed which cut P (line 7,8), using the function EvalCuts. All such sets of improved BSPs are combined in $\mathcal{B}_{\text{new}}$, and the at most b of the best BSP trees are selected from $\mathcal{B}_{\text{new}}$, where first the already completed BSP trees are selected for the next round of $\mathcal{B}_{\text{curr}}$. Once all BSP trees partition the object into parts that fit in the printing volume, the best BSP tree is selected.

Algorithm 1: Chopper search, taken from [38]

Input: O is the object, b is beam width
Output: T is the BSP tree that partitions the object

```
1  Function BeamSearch(O, b):
2      B_curr ← ∅;
3      while not all parts P' ∈ parts(P) in some P ∈ B_curr fit in a build volume do
4          B_new ← ∅;
5          foreach P ∈ B_curr if some part P ∈ P does not fit in a build volume do
6              B_curr ← B_curr \ {P};
7              P ← LargestPart(O, P);
8              B_new ← B_new ∪ EvalCuts(P, P);
9          end
10         while |B_curr| < b do
11             B_curr ← B_curr ∪ HighestRanked(B_new);
12         end
13     end
14     return HighestRanked(B_curr);
```

The helper algorithm, EvalCuts, for improving a given BSP tree $\mathcal{P}$ on a part $P \in \mathcal{P}$ works as shown in Algorithm 2. First, the algorithm selects a set of normals n_i by combining the normals of a three-times subdivided octahedron (line 1) with the axes of the minimum oriented bounding box of part P (line 2). Next the algorithm uniformly samples points p_j over the range where n_j intersects P, to get a set of uniformly sampled planes $\pi_{ij} = (n_i, p_j)$. It then creates new BSPs $\mathcal{P}'$ by adding the cut plane π_{ij} to $\mathcal{P}$ at the leave of part P. Then $\mathcal{P}'$ is added to a list of candidates, $\mathcal{B}_{\text{can}}$, only if its calculated objective score $f(\mathcal{P}')$ is not infinity (the cut is feasible).

Finally, it returns the best (lowest) scoring candidates from $\mathcal{B}_{\text{can}}$, based on the objective scores, that are sufficiently different, i.e. not too close to each other.

Algorithm 2: Evaluation of Cuts, taken from [38]

Input: $\mathcal{P}$ is some BSP tree, where part $P \in \mathcal{P}$ does not yet fit in the build volume

Output: $\mathcal{B}_{\text{result}}$ is a set of the best, sufficiently different, BSP trees, which are $\mathcal{P}$ with a cut added partitioning P

Function EvalCuts($\mathcal{P}$, P):
  $N \leftarrow$ UniformNormals() $\cup$ AuxiliaryNormals(P);
  $\mathcal{B}_{\text{can}} \leftarrow \emptyset$;
  foreach $n_i \in N$ **do**
    // Uniformly sample points p_j over the range of n_i where it intersects P
    foreach *plane* $\pi_{ij} = (n_i, p_j)$ *intersecting* P **do**
      $\mathcal{P}' \leftarrow$ AddToBsp($\mathcal{P}$, P, π_{ij});
      $f(\mathcal{P}') \leftarrow \sum_{i=1}^{i} \alpha_i f_i(\mathcal{P}')$;
      if $f(\mathcal{P}') \neq \infty$ **then**
        $\mathcal{B}_{\text{can}} \leftarrow \mathcal{B}_{\text{can}} \cup \{\mathcal{P}'\}$;
      end
    end
  end
  $\mathcal{B}_{\text{result}} \leftarrow \emptyset$;
  foreach $\mathcal{P} \in \mathcal{B}_{\text{can}}$ *sorted by* $f(\mathcal{P})$ **do**
    if SufficientlyDifferent($\mathcal{P}$, $\mathcal{B}_{\text{result}}$) **then**
      $\mathcal{B}_{\text{result}} \leftarrow \mathcal{B}_{\text{result}} \cup \{\mathcal{P}\}$;
    end
  end
  return $\mathcal{B}_{\text{result}}$;

2.3 Limitations and Advantages

As described in Section 6 of the paper by Luo et al., the partitioning technique as described in the paper has some limitations. In particular, we will focus on the limitation imposed by BSP to make the cuts, which means that in each of the recursive steps, the part is cut by a full plane.

As discussed above, the algorithm by Luo et al. cuts an object using a plane. This can lead to cases where the algorithm is forced to partition the model in multiple places to achieve a favourable cut. This is shown in Figure 1.4, here cutting the hind legs of the dog would force the original algorithm to cut its front legs. In this case, the original algorithm chooses another cut.

Cutting using a full plane also has advantages, it is noted that cutting using a plane will always result in a model that can be assembled. Assemblability is present by construction in the technique by Luo et al., however, this may not be the case for other approaches.

Chapter 3

A New Partitioning Technique

This chapter will discuss the approach taken to improve the quality of partitions, based on the objectives as discussed in Section 3.1. First, relevant objective functions will be discussed which aim to evaluate the quality of a given partition. These functions will be used in Chapter 4 to compare the quality of both the original and proposed methods.

Second, a novel method of dealing with the limitation of planar cuts used in the technique by Luo et al. will be discussed. This proposed improvement will cut parts using a plane, but will only cut part of the model intersecting the plane such that the cut results in exactly two parts. This will allow for more freedom when cutting, as previously impossible cuts can be performed as exemplified in Figure 3.3.

3.1 Relevant Objective Functions

As discussed in Section 2.1, Luo et al. [38] proposed several objective functions in their paper, which are used to evaluate the quality of a partition while either the original or the proposed algorithm is running. To enable assessment of the quality of a final partition, this section will simplify the functions to evaluate the quality of a partition where each of its parts fits in the build volume.

Number of Parts. To evaluate the final solution, the number of parts of the final partition will be counted. This number is expected to be lower for the proposed solution since it will have more freedom in choosing cuts.

$$f_{\text{part}}(\mathcal{P}) = |\mathcal{P}| \tag{3.1}$$

Utilization. The utilization objective aims to maximize the utilization of the printing volume by penalizing a partition on the part with the least volume. The utilization objective is expected to be slightly lower, and thus better, for the proposed method due to a higher freedom of cuts. The utilization function can be used as discussed in Section 2.1 with the expected number of build volumes needed $\mathcal{O}(P)$ is 1 for each part P in the partition of the final product.

$$f_{\text{util}}(\mathcal{P}) = 1 - \min_{P \in \mathcal{P}} \frac{\text{Vol}(P)}{V} \tag{3.2}$$

Seam Unobtrusiveness. The proposed algorithm will likely improve the placement of seams to make them less obtrusive. Allowing for more freedom in choosing cuts will allow the algorithm to place the seams in a better spot. As for the seam objective, we can use the original function. The function is repeated below, where $\epsilon(C)$ is again the cost of a cut C given by the ambient occlusion p of points on that seam, and $\hat{\epsilon}(P)$ is the same as the cost of the latest cut.

$$\epsilon(C) = \frac{1}{d_0} \int_{\partial C} p dx \tag{3.3}$$

$$f_{\text{seam}}(\mathcal{P}) = \frac{1}{\mathcal{O}_0} \left(\sum_{C \in \mathcal{P}} \epsilon(C) + \sum_{P \in U \subseteq \mathcal{P}} (\mathcal{O}(P) - 1) \hat{\epsilon}(P) \right) \tag{3.4}$$

The Objective Function. The objectives of *Connector Feasibility*, *Structural Analysis*, *Fragility* and *Symmetry* will not be included for several reasons. First of all, implementing these objective functions requires time and effort which we choose to spend on developing the proposed algorithm instead. Second, these objectives are not expected to show relevant improvements for the proposed algorithm over the original algorithm.

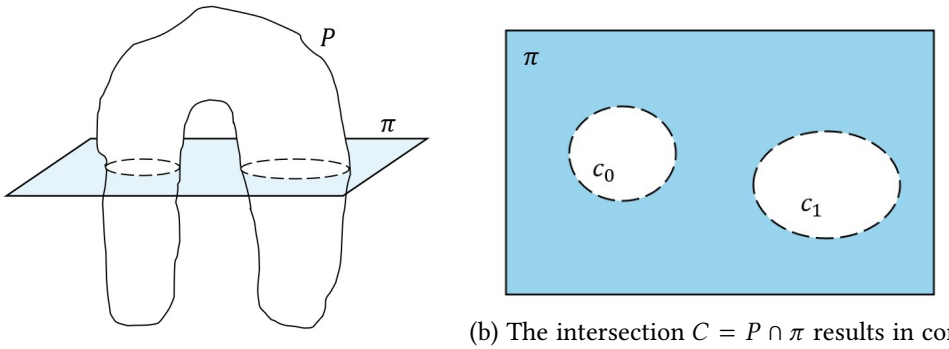
$$f(\mathcal{P}) = \alpha_{\text{part}} \cdot f_{\text{part}}(\mathcal{P}) + \alpha_{\text{util}} \cdot f_{\text{util}}(\mathcal{P}) + \alpha_{\text{seam}} \cdot f_{\text{seam}}(\mathcal{P}) \tag{3.5}$$

Given these objective functions, the final objective function for a given partition $\mathcal{P}$ for which each part fits in the build volume is shown in Equation 3.5. The objectives imposed by these functions are expected to interact with each other. When the weight for the part function (α_{part}) is relatively high compared to the other weights the algorithm is expected to produce partitions with fewer parts, but worse more visible seams. The utility function and part function are expected to enhance each other, where both aim for large and few parts in the final partition.

3.2 Local Planar Cuts

This part will discuss a proposed method of partitioning a model based on the method by Luo et al. This novel method aims to relax the constraint imposed by the algorithm of Luo et al., which requires the model to be cut using a cutting plane and might result in more than two parts after cutting. This method will still cut using a plane, but it will "sow" parts back together such that the result will contain exactly two parts. Note that a part or object mesh M may be defined as $M = (V, F)$ or $M = (V_M, F_M)$ o where V or V_M are the vertices, and F or F_M are the faces of this mesh.

More precisely, given a part P and a cut plane π which intersects P, let us consider the intersection $C = P \cap \pi$ of the part and the plane. Note that C can have one or more connected components. The method by Luo et al. will cut the model at each such component, possibly partitioning the model into more than two pieces. Instead, we will be using a cut $C' \subseteq C$ such



(a) Plane π intersects part P.

(b) The intersection $C = P \cap \pi$ results in connected components c_0 and c_1.

Figure 3.1: Plane π intersects part P, resulting in set $C = \{c_0, c_1\}$. Clearly picking either $C' = \{c_0\}$, or $C' = \{c_1\}$ will lead to a cut resulting in exactly two parts.

that cutting the model only at the components in C' will partition the model into exactly two parts.

Consider the example in Figure 3.1, if the original algorithm by Luo et al. uses the plane π this cut will result in three parts. Instead, the proposed method will find $C = \{c_0, c_1\}$ and can pick $C' = \{c_0\}$ or $C' = \{c_1\}$ to be cut, resulting in two parts. Thus, if we would like to cut off one of the "legs" of the model in Figure 3.1, the new method is not forced to cut off the other leg.

Note that assemblability is no longer guaranteed if we only make a local cut, so a separate test should be added for this. In all models we tested, however, assemblability was never an issue. In the remainder, we will ignore this assemblability check.

Pseudocode. More formally, the base algorithms are the same, so Algorithm 1 and Algorithm 2 have not been changed. However the method AddToBsp in Algorithm 2 is replaced by TwinCut for the proposed algorithm. Do note that TwinCut returns a list of partitions, instead of just one. Thus instead of calculating the objective score for a single partition, EvalCuts will calculate the objective score for each partition.

Next, we describe the method TwinCut($\mathcal{P}$, P, π), which replaces AddToBsp($\mathcal{P}$, P, π) which cuts part P in partition $\mathcal{P}$ using plane π, see Algorithm 3 for the pseudocode. This method will return all a set $\mathcal{B}_{\text{cut}}$ of partitions with part P cut into exactly two parts. It will initialize $\mathcal{B}_{\text{cut}}$ and then the method will create a temporary partition $\mathcal{P}'$ by removing P from the partition $\mathcal{P}$. Next, the set F of faces intersecting part P at plane π will be found. The method GetConnectedFaces(F) will then find the list of all connected components C, where the union of all connected sets in C is F; the intersection of C is empty; and if two faces $f_a, f_b \in F$ share at least one vertex, they are in the same component $c \in C$. Once these connected components have been found, all possible subsets of C, except the empty set, are traversed. If cutting the faces listed in such a subset results in exactly two parts, then these are added to a copy of $\mathcal{P}'$ and added to the results. If all subsets have been considered, all generated partitions are returned.

Algorithm 3: Cut part P into two parts, replaces AddToBsp in Algorithm 2.

Input: $\mathcal{P}$ is some partition, where part $P \in \mathcal{P}$ does not yet fit in the build volume and plane π intersects P.

Output: $\mathcal{B}_{\text{cut}}$ is a list of partitions, which are $\mathcal{P}$ with P cut into exactly two parts through.

```
1  Function TwinCut(P, P, π):
2      B_cut ← ∅;
3      P_sub ← P − P;
4      F ← IntersectingFaces(P, π);
5      C ← GetConnectedFaces(F);
6      foreach C′ ∈ PowerSet(C) \ ∅ do
7          parts ← cutAtComponent(P, C′, π);
8          if |parts| == 2 then
9              P′ ← P_sub ∪ parts;
10             B_cut ← B_cut ∪ {P′};
11         end
12     end
13     return B_cut;
```

Finally, let us discuss the function which performs the local cut through part the largest part P, cutAtComponent(P, C, π). This function cuts a part P into two at plane π, but only through faces in C. The function is adapted from the function slice_mesh_plane(...) as implemented in Trimesh [17]. This function splits a mesh $M = (V, F)$ defined by a set of vertices V and faces F on a plane, and only maintains the mesh on the positive side of the plane. A face in F is defined as a triple of vertex indices.

The modified version of the function in Trimesh, cutAtComponent(P, C, π), which maintains the mesh on both sides of the plane π and cuts only faces in C, works as follows. Given a part P as a triangulated mesh, the intersection points of P and π are determined, where each intersection point must be on an edge of a face in C. Faces in P are removed if they occur in C. Next, these new vertices are ordered and combined with the vertices above the plane of the edges of faces in C. Given this ordering, new faces are created. These faces can be divided into two categories, quadrilaterals and triangles, as shown in Figure 3.2. After determining the vertices and faces for the quadrilaterals and triangles, they are added to the mesh. Next, the quadrilaterals are triangulated and duplicate vertices are combined.

Once the quadrilaterals and triangles above the plane are handled, the quadrilaterals below the plane must also be handled. This works the same way, apart from merging duplicate vertices. Duplicate vertices are only merged if they are below the plane or if they are connected to a face determined to be below the plane.

After adding all triangles and quadrilaterals, and merging duplicate vertices; the algorithm will traverse the mesh and separate the model into parts. Next, if the number of parts is exactly 2 all vertices that are on the plane π are collected, and a face is added which is triangulated.

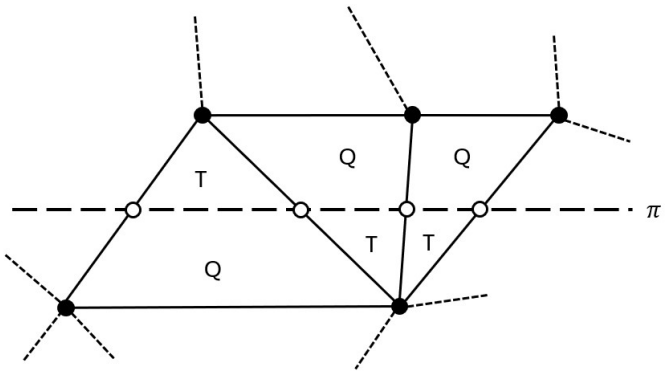


Figure 3.2: 2D representation of cutting a mesh using plane π. At the intersection of a part P and plane π vertices are added. The newly created faces are divided into two categories: Quadrilaterals (Q) and Triangles (T).

3.3 Analysis of the Runtime

Let us briefly compare the runtime of this proposed algorithm to the original, by determining the worst-case runtime for both. In terms of runtime, the main difference is that the AddToBSP returns exactly one BSP, while the proposed variant returns at most $2^{|C|} - 1$ partitions for some intersection C of part P with plane π, where $|C|$ is the number of connected components in this intersection using the function TwinCut which replaces AddToBSP in EvalCuts. As the differing factor between the two algorithms is the runtime of AddToBSP or TwinCut, let us first express the worst-case runtime of BeamSearch as described in Algorithm 1 with respect to an unknown cutting algorithm CutMethod.

The worst-case runtime of BeamSearch can be expressed as follows, given an object $O = (V, F)$ and beam width b. In each iteration of the outer while-loop on line 3, a single cut is added to each partition in $\mathcal{B}_{\text{curr}}$, while the loop on line 5 goes over every partition in $\mathcal{B}_{\text{curr}}$ cutting its largest part. This while-loop lasts until b partitions fit in the building volume. Let us define the number of iterations as #parts as this value depends on the size and shape of the model O and the weights and parameters of the algorithm.

Finding the part with the largest bounding box, in $LargestPart(O, \mathcal{P})$ will find the minimum and maximum values of each of the vertices, this takes $\mathcal{O}(|V|)$ time. The inner while-loop on line 10 iterates over all partitions in $\mathcal{B}_{\text{new}}$ and the score of each partition as determined inside the function AddToBSP or twinCut, extracting the b highest scoring partitions. This inner while-loop takes at most $\mathcal{O}(|\mathcal{B}_{\text{new}}| \cdot b)$ time and thus, the worst-case runtime of BeamSearch can be expressed as shown in Equation 3.6. Let us now determine the runtime for EvalCuts.

$$\text{running time of BeamSearch} = \mathcal{O}(\#\text{parts} \cdot (b \cdot \text{running time of EvalCuts} + |\mathcal{B}_{\text{new}}| \cdot b) \quad (3.6)$$

The runtime of EvalCuts$(\mathcal{P}, P)$ for partition $\mathcal{P}$ and part $P = (V_P, F_P)$ of this partition depends on all planes in the loops on lines 5 and 6, let us define the number of planes here as $\#planes$. Each iteration takes $\mathcal{O}(\text{CutMethod})$ time, where CutMethod is the cutting method

used in either of the algorithms. Finding the number of parts and utility can respectively be done in constant and $\mathcal{O}(|V_P|)$ time. However, finding the seam score requires us to calculate the ambient occlusion, which takes $\mathcal{O}(|F_P| * |V_P| * \#rays)$ time, as for each vertex on the seam, which are at most $|V_P|$ vertices, $\#rays =$ `SEAM_OCCLUSION_RAY_COUNT` number of rays are shot. If any face in F_P blocks this ray, the ambient occlusion score is increased.

The for loop on line 14 takes at most $\mathcal{O}(\#planes)$ time, as SufficientlyDifferent only compares the planes and runs in constant time. Thus EvalCuts takes $\mathcal{O}(\#planes \cdot (\mathcal{O}(\text{CutMethod}) + |F_P| * |V_P| * \#rays))$ time. As the $|\mathcal{B}_{\text{new}}|$ is at most $\#planes$ the total runtime of BeamSearch can be expressed as shown in Equation 3.7.

$$\text{running time of BeamSearch} = \mathcal{O}(\#\text{parts} \cdot b \cdot \#planes \cdot (\text{running time of CutMethod} + |F_P| * |V_P| * \#rays))) \tag{3.7}$$

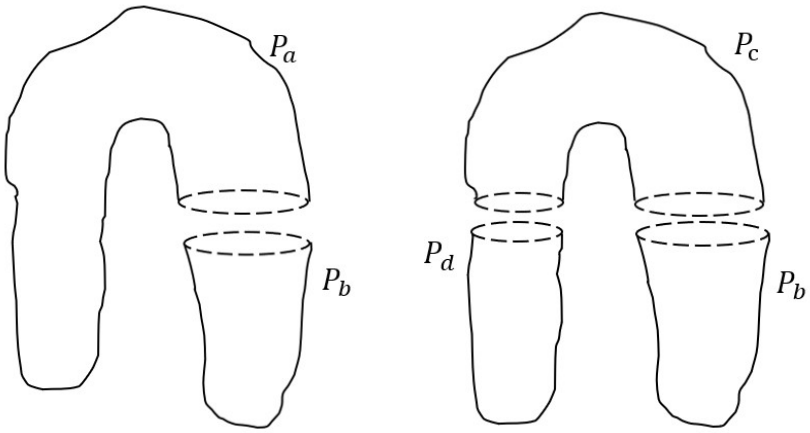
Original Algorithm. Let us determine $\mathcal{O}(\text{CutMethod})$ for the original algorithm, which is equal to the worst-case runtime of AddToBSP. This function works similarly to $\text{CutAtComponent}(P, C, \pi)$, as described above, but it cuts at all planes intersecting the plane π. Thus, AddToBSP finds intersection of part $P = (V_P, F_P)$ and plane π in $\mathcal{O}(|V_P|)$ time. The algorithm then adds faces. Next, it will merge vertices and update the faces; as the faces are triples of indices of vertices these have to be updated as well. To this end, the algorithm will find the unique vertices in V_P and calculate an inverse to apply to the faces. This is done using the Unique method in Numpy [25] which takes $\mathcal{O}(|V_P| \cdot \log(|V_P|))$ time. Hence, the worst-case runtime of AddToBSP is as given in Equation 3.8.

$$\text{running time of AddToBSP} = \mathcal{O}(|V_P| \cdot \log(|V_P|)) \tag{3.8}$$

Proposed Algorithm. Now, let us determine the runtime for the proposed algorithm. The runtime of $\text{CutAtComponent}(P, C, \pi)$ for a part $P = (V_P, F_P)$ with vertices V_P and faces F_P is as above, running time of CutAtComponent $= \mathcal{O}(|V_P| \cdot \log(|V_P|))$. For TwinCut, the method CutAtComponent is run at most $|2^{|C|} - 1|$ times, where C is the set of connected components on the intersection of the plane π and part P. Finding this set C requires us to check for each face in F_P if it intersects π, and then traverse this set to determine which faces share at least one vertex. This takes $\mathcal{O}(|F_P|)$ time. Thus we have the worst-case runtime of TwinCut as given in Equation 3.9.

$$\text{running time of TwinCut} = \mathcal{O}(|F_P| + |2^{|C|}| \cdot |V_P| \cdot \log(|V_P|)) \tag{3.9}$$

Runtime Conclusion. Concluding, the difference between the runtime of both algorithms stems from the fact that TwinCut considered all possible cuts of connected components on the intersection of the plane π and part P, while AddToBSP only performs a single cut. This is exemplified in Figure 3.3, where P_a and P_b are the result of cutting only at intersection c_1 and P_b, P_c and P_d are the result of cutting using the original algorithm.



(a) Part P cut at intersection c_1, using the proposed algorithm.

(b) Part P cut at plane π, using the original algorithm.

Figure 3.3: Comparison between the original algorithm and the proposed algorithm, cutting part P using the plane π, or the component c_1 of part π, as shown in Figure 3.1.

Chapter 4

Testing and Results

This chapter will provide a comparison of the original technique by Luo et al. with the proposed technique as presented in Chapter 3, aiming to highlight cases in which the proposed technique performs differently from the original. The comparison aims to highlight differences in score and partition results, and thus emphasize use cases in which either technique performs better.

First, Section 4.1 will discuss the resources used to compare the two techniques. In particular, it will highlight which models, test setup, and implementation were used. In Section 4.2, the objective function weights will be determined. Then in Section 4.3, suitable parameters will be determined for the new partitioning technique by comparing the effect of each parameter on the final scores. Finally, Section 4.4 will compare the original and proposed techniques using the parameters and models from the first three sections.

4.1 Experimental Setup

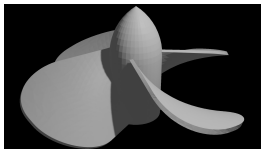
This section will discuss the models, setup, and implementation used in the rest of this chapter. All experiments use the setup as described below, except if noted otherwise.

Models. All models used for tuning and testing are included Appendix A, as well as below. For each model, some relevant statistics pertaining to its size and number of primitives (vertices, edges, and faces) are included in Table A.1. The models aim to represent the wide variety of models found in real-world applications. Additionally, certain models have been selected based on their anticipated variations in performance between the different techniques. For example, Model A.3 (pikachu) and Model A.9 (dog) are likely to have cuts resulting in more than 2 parts for the original algorithm. Meanwhile, other models are less likely to be cut into more than two parts, including models like Model A.7 (vase) and Model A.8 (nut).

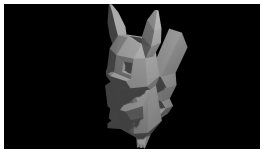
Sections 4.2 and 4.3 will discuss tuning the weights and parameters of the algorithm. This will be done using only Models A.1 (logo) and A.2 (propeller), as tuning takes a considerable amount of time.

Setup. All tests are run on machines with an Intel Core i5-6640HQ CPU (2.60GHz) and 8.00 GB RAM. In addition, only one test is run on a machine at a time. The size of the build volume

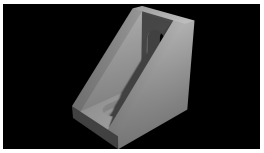
A.1 *Logo*



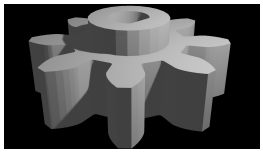
A.2 *Propeller*



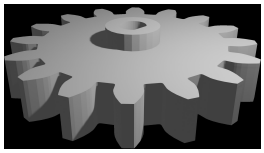
A.3 *Pikachu*



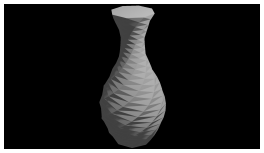
A.4 *Bracket*



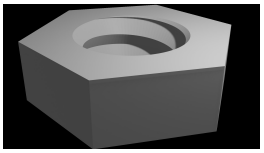
A.5 *Gear Spur 1*



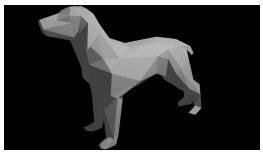
A.6 *Gear Spur 2*



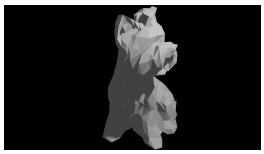
A.7 *Vase*



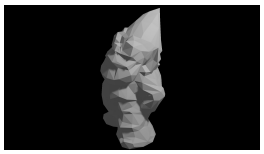
A.8 *Nut*



A.9 *Dog*

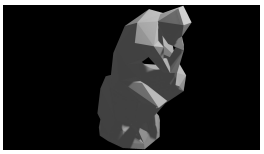


A.10 *Yorkie*



A.11 *Gnome*

A.12 *Moai*



A.13 *Thinker*

is set to the build volume of the Anycubic Photon Mono 4k, see figure Figure 1.3, as this printer is available to the author. Its build volume measures 165 x 132 x 80 mm (HWD) [5].

Implementation. Both the algorithm for the original and proposed technique have been implemented in Python and can be found on Github `https://github.com/PFvanBeerendonk/final_thesis__partitioning/`. Several libraries have been used to implement the algorithms, these uses of third-party code are noted here.

Part of the implementation, particularly the objective functions, was inspired by a partial Python implementation of the original algorithm, pychop3d [11]. However, to allow for easy adaptation of the original, the implementation differs significantly from pychop3d.

Meanwhile, most geometric operations, such as finding the intersection of a mesh with a plane or the volume of a mesh, employ methods from the package Trimesh [17]. In addition, the method developed for the proposed technique, which allows local cuts on a plane, is adapted from the `slice_mesh_plane` method and related methods in Trimesh. To calculate the ambient occlusion of vertices, used to find the seam objective, the package libigl by A. Jacobson [31] was used.

4.2 Determining the Objective Function

This section will determine the weights ($\alpha_{\text{part}}, \alpha_{\text{util}}$ and α_{seam}) of the objective function, as described in Section 3.1, based on a representative subset of the models, namely Models A.1 (logo) and A.2 (propeller). The goal of this section is to adjust the weights of the objective function to achieve partitions of reasonable quality. Here we aim for a partition with at most 2 parts more than the minimum number of parts found in other cases, with the best seam score. The algorithm should have found the best partition, without increasing the number of parts used. In addition, the runtime will be monitored to conclude whether the choice of weights influences the runtime. In Section 4.3 other parameters will be tuned given the objective function weights found in this section.

Setup. In this section, the beam width (b) is set to $b = 3$ and the distance d_π between uniformly picked planes is set to $d_\pi = 5$mm, as recommended by Luo et al. In addition, the two parameters controlling the variety of partitions maintained, based on angle ($\angle_{\text{diff}}$) and distance (d_{diff}), are both set to 0.1.

The weights of the objective function will be tuned based on the Models A.1 (logo) and A.2 (propeller). The score $f(\mathcal{P})$ of a partition $\mathcal{P}$ is determined based on the weighted sum of these 3 objective functions and is expressed below as a reminder in Equation 4.1.

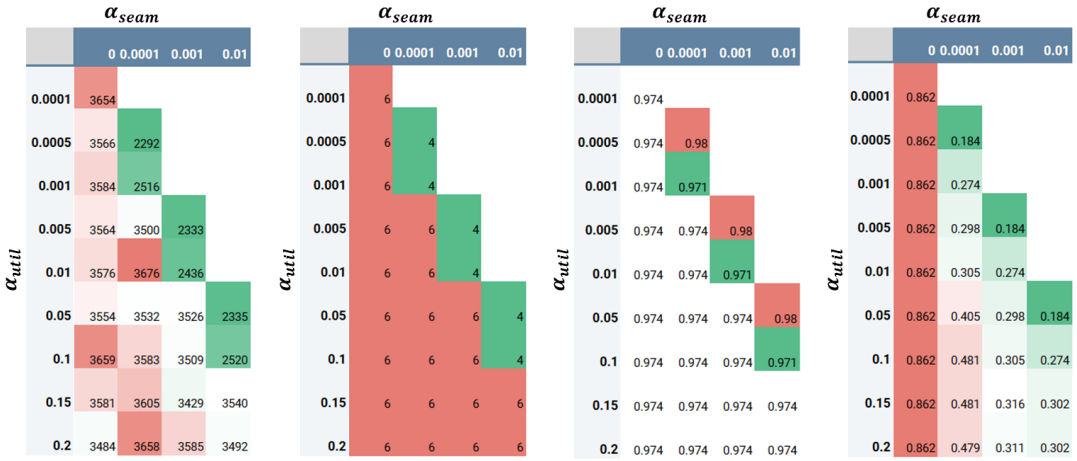
$$f(\mathcal{P}) = \alpha_{\text{part}} \cdot f_{\text{part}}(\mathcal{P}) + \alpha_{\text{util}} \cdot f_{\text{util}}(\mathcal{P}) + \alpha_{\text{seam}} \cdot f_{\text{seam}}(\mathcal{P}) \tag{4.1}$$

To tune the weights, α_{part} will be set to 1, such that the α_{util} and α_{seam} can be tuned with respect to it. For each combination of weights and models under testing, the algorithm will be run and the scores and runtime will be reported. The weights are chosen to be close to the values proposed by Luo et al. In their paper, the values $\alpha_{\text{part}} = 1, \alpha_{\text{util}} = 0.1$, and $\alpha_{\text{seam}} = 0.01$ were chosen. Hence, values were chosen around this value. The value α_{util} is picked from the range $[0, 0.0001, 0.0005, 0.001, 0.005, 0.01, 0.05, 0.1, 0.15, 0, 2]$ while the value α_{seam} is picked from the range $[0, 0.0001, 0.001, 0.01, 0.1]$.

The results will be presented in figures. Moreover, heatmaps showing runtime will use a colour range from green (low runtime) to red (high runtime), while heatmaps showing scores will use a colour range from blue (low/good score) to white (high/bad score). Full results are presented in Appendix B.

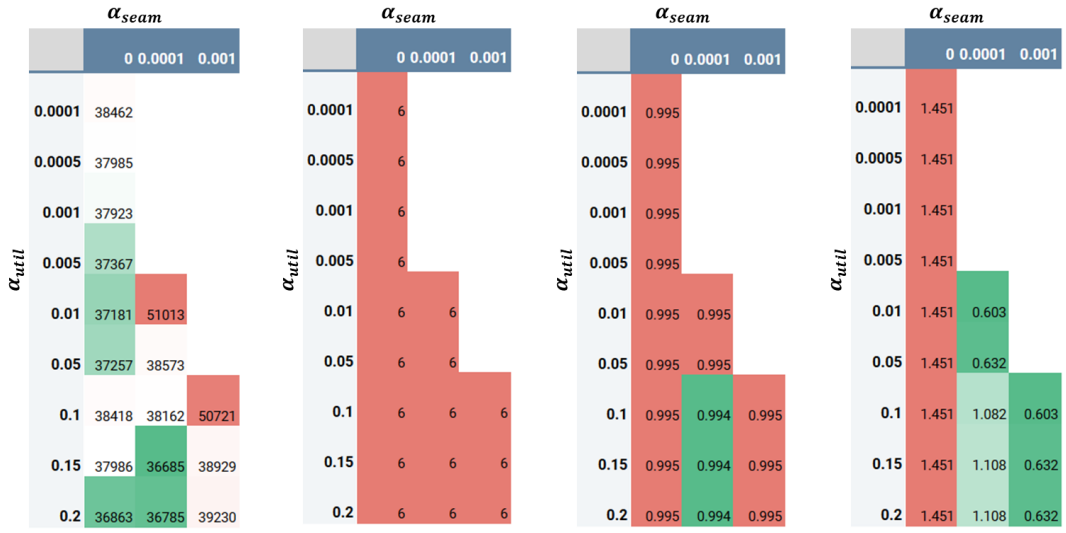
Tuning the Weights. Evaluating the weights and models, as described above, yielded the results shown in Figures 4.1 and 4.2. More detailed results are shown in Tables B.1 and B.2. This part will discuss results pertaining to the runtime and the objective functions' scores, as well as some general remarks. Finally, the weights used in upcoming sections will be determined. Notice that the top left part of each graph was left empty, as these areas were not explored because the number of parts and the runtime in these areas increased dramatically. For Model A.1 the number of parts became higher than 10, while for A.2 the number of parts became higher than 12.

The runtime is not directly dependent on the actual weights, but rather on the number of cuts made. This is visible in the results for $\alpha_{\text{util}} = 0.1$ for Model A.1 (logo) where the seam



(a) Runtime results. (b) Part scores. (c) Util scores. (d) Seam scores.

Figure 4.1: Results of tuning the weights α_{util} and α_{seam} for Model A.1 (logo). For each combination of weights, the scores are rounded to 3 decimals and the runtime is rounded to seconds.



(a) Runtime results. (b) Part scores. (c) Util scores. (d) Seam scores.

Figure 4.2: Results of tuning the weights α_{util} and α_{seam} for Model A.2 (propeller). For each combination of weights, the scores are rounded to 3 decimals and the runtime is rounded to seconds.

score decreases as the seam weight is increased, while the number of parts and the utility score remain the same. This observation is supported by the fact that right of the diagonal, the number of parts increases to more than 30 and the runtime increases to more than 6 hours.

As already discussed by Luo et al., a good balance must be struck between the weights α_{util} and α_{seam}, as the seam objective will favour cutting of small parts. This is clearly visible in the results in Table B.1, as increasing α_{seam} with respect to α_{util} will result in a stable utility and part score until a tipping point is reached, after which the number of parts drastically increases.

In general, it is noticeable that the seam weight function produces slightly varying scores, which is probably caused by the way rays were shot to calculate the ambient occlusion by the library used. This section did not explore the parameter `SEAM_OCCLUSION_RAY_COUNT`, which controls the number of rays used to calculate ambient occlusion in the seam objective due to time constraints. Increasing this parameter is expected to increase the accuracy of the objective. Notice that both the implementation of the original and the proposed variant do not implement the connector objective. This can lead to cuts with a small area and allows for smaller parts to be cut off.

Finally, given the findings discussed above, the proposed weights for running the proposed version of the algorithm are as shown below. The combination of weights should not be in the area where the algorithm produced a large amount of parts, and the weights should result in the best seam scores. Thus, the chosen weights give the best seam score (0.603) for Model A.2 (propeller) while also giving the best seam score (0.305) for Model A.1 within the area where Model A.2 has a reasonable amount of parts. In addition, the chosen weights give a slightly better runtime for both models when compared to the other option of $\alpha_{\text{util}} = 0.01, \alpha_{\text{seam}} = 0.0001$ which produced the same seam scores. In the upcoming sections, these weights will be used to perform all further experiments.

$$\alpha_{\text{part}} = 1$$
$$\alpha_{\text{util}} = 0.1$$
$$\alpha_{\text{seam}} = 0.001$$

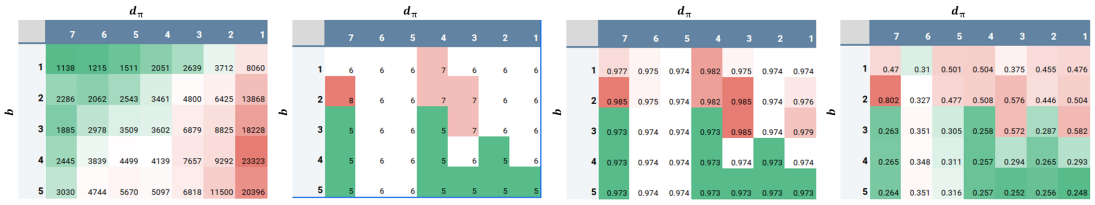
4.3 Testing the New Partitioning Technique

This section will explore the effect of other parameters used in the algorithm. In particular, this section will look at two pairs of parameters. The first pair of parameters, the beam width b and the distance d_π between uniformly picked planes, are expected to influence the quality of the final model as well as the runtime. The second pair of parameters, the parameters used to determine if two partitions are sufficiently different from each other regarding angle ($\angle_{\text{diff}}$) and distance (d_{diff}), are expected to only influence the quality of the final model. Hence, this section will first tune the first pair of parameters, after which it will tune the second pair of parameters. Finally, a combination of parameters will be proposed which is to be used to compare our new algorithm and the original one by Luo et al.

The weights of the objective function are set as determined in the previous section: $\alpha_{\text{part}} = 1, \alpha_{\text{util}} = 0.1$ and $\alpha_{\text{seam}} = 0.001$. As in the previous section, the Models A.1 (logo) and A.2 (propeller) will be used.

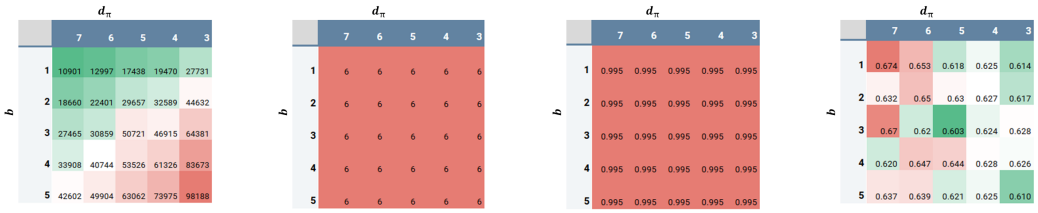
Beam Width, and Plane Distance. This paragraph will tune the parameters beam width (b), and the distance between uniformly spaced planes (d_π). These parameters are expected to influence both the runtime of the algorithm and the quality of the final model. Tables B.3 and B.4 show detailed results for this part, while Figures 4.3 and 4.4 show a heatmap of the runtime and seam score. $\angle_{\text{diff}}$ and d_{diff} will both be set to 0.1.

In the paper by Luo et al., the values for b and d_π were chosen as $b = 3$ and $d_\pi = 5$mm. Hence, values to be tested for b will be chosen from the strictly positive integer range $[1, 2, 3, 4, 5]$. d_π will be chosen from the strictly positive real range $[1, 2, 3, 4, 5, 6, 7]$. Here high values for b and low values for d_π are expected to produce better results, in terms of objective scores.



(a) Runtime results. (b) Part scores. (c) Util scores. (d) Seam scores.

Figure 4.3: Results of tuning the parameters beam width (b), and the distance between uniformly spaced planes (d_π) for Model A.1 (logo). For each combination of weights, the scores are rounded to 3 decimals and the runtime is rounded to seconds.



(a) Runtime results. (b) Part scores. (c) Util scores. (d) Seam scores.

Figure 4.4: Results of tuning the parameters beam width (b), and the distance between uniformly spaced planes (d_π) for Model A.2 (propeller). For each combination of weights, the scores are rounded to 3 decimals and the runtime is rounded to seconds.

First, both heatmaps in Figures 4.3 and 4.4 clearly show that the runtime increases as either the b increases or d_π decreases. So finding a partition requires more time as b increases or d_π decreases, which is less desirable as we want our partition as fast as possible. This can be attributed to the fact that b influences the number of different partitions that are maintained in one round of the algorithm. The number of cuts calculated over each of these b partitions is roughly the same. The same goes for the distance between uniformly picked planes d_π, if you half d_π we expect the number of cuts made to double, as twice as many planes are considered per round. In the same manner, we expect the scores to increase in most cases as more planes

are considered. However, because of the greedy nature of the algorithm, this need not always be the case.

Second, the score seems to be influenced by both b and d_π however, the influence of b appears to be bigger for Model A.1 (logo) while the influence of d_π appears to be greater for Model A.2 (propeller). The expectation was that the effect would be visible for both models, however this is not the case. We expect that the difference in shape between the models mainly influences this. As Model A.1 (logo) has straight parts where each cut costs roughly the same, while Model A.2 (propeller) has more curved surfaces leading to more expensive cuts.

Some final remarks, in Figure 4.4 one might notice that the columns for $d_\pi = 1$ and $d_\pi = 2$ are missing. In Table B.4 these are included however, it is noted that the system ran out of RAM during execution. Due to the size of the model, and the number of possibilities considered too many partitions were maintained.

Finally, the parameters were chosen as $b = 4$ and $d_\pi = 4$, since we did not want to choose values too close to $d_\pi = 2$ where for the larger model Model Figure A.2 (propeller) we had issues with RAM availability. Meanwhile, we did not want the runtime to be too high.

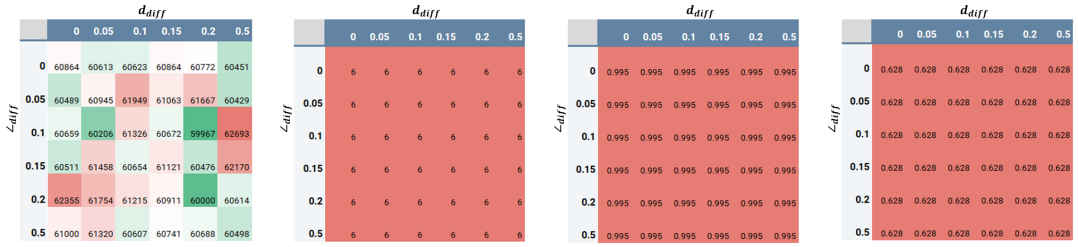
Sufficiently Different. The parameters $\angle_{\text{diff}}$ and d_{diff} control the variance in planes maintained after a round by the function SufficientlyDifferent($\mathcal{P}$, $\mathcal{B}_{\text{result}}$) in EvalCuts($\mathcal{P}, P$). Figures 4.5 and 4.6 show the runtime results, while Tables B.5 and B.6 show the full results. The paper by Luo et al. used 0.1 for both $\angle_{\text{diff}}$ and d_{diff}. The values must be chosen between 0 and 1, as they both express differences over this range in terms of angle and distance. Hence, values over the range $[0, 0.05, 0.01, 0.15, 0.2, 0.5]$ were explored.

(a) Runtime results. (b) Part scores. (c) Util scores. (d) Seam scores.

Figure 4.5: Runtime results of tuning the parameters $\angle_{\text{diff}}$ and d_{diff} for Model A.1 (logo). For each combination of weights, the scores are rounded to 3 decimals and the runtime is rounded to seconds.

In terms of runtime, we expected to see no difference between tests. Looking at the heatmaps in Figures 4.5 and 4.6, there does not appear to be much if any effect from changing the parameters. The runtime of Model A.1 varies between 4157 and 4397 seconds, while for Model A.2 the runtime varies between 59967 and 62693 seconds. If you put this in the context of previous runtime heatmaps this variance is probably caused by randomness. However, more executions of the algorithm would be needed to get more balanced results.

The scores, as shown in Tables B.5 and B.6 were the same for all test cases. For Model A.1 the scores were $f_{\text{part}} = 5$, $f_{\text{util}} = 0.973$, and $f_{\text{seam}} = 0.257$. For Model A.1 the scores were



(a) Runtime results. (b) Part scores. (c) Util scores. (d) Seam scores.

Figure 4.6: Runtime results of tuning the parameters $\angle_{\text{diff}}$ and d_{diff} for Model A.2 (propeller). For each combination of weights, the scores are rounded to 3 decimals and the runtime is rounded to seconds.

$f_{\text{part}} = 6$, $f_{\text{util}} = 0.995$, and $f_{\text{seam}} = 0.628$. As no clear explanation was found for this behaviour after searching $\angle_{\text{diff}} \in [0, 0.2]$ and $d_{\text{diff}} \in [0, 0.2]$, the search space was expanded to also include $\angle_{\text{diff}} = 0.5$ and $d_{\text{diff}} = 0.5$.

As the parameters do not appear to affect the scores nor the runtime, $\angle_{\text{diff}}$ and d_{diff} were both set to 0.1, as was done by Luo et al. Thus the parameters used in the final comparison are:

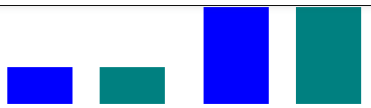
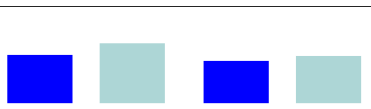
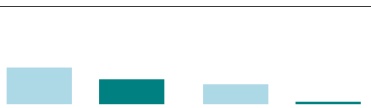
$$b = 4$$
$$d_{\pi} = 4$$
$$\angle_{\text{diff}} = 0.1$$
$$d_{\text{diff}} = 0.1$$

4.4 Comparing the New Method to Luo et al.'s Method

In this final section, both techniques will be compared given the models found in Appendix A and given the weights and parameters as found in Sections 4.2 and 4.3. Both algorithms are run for each model, and the resulting partitions are shown in Appendix C. This section will discuss the difference in runtime, scores and finally general quality of the models.

Table 4.1 summarizes the results more clearly. Each cell contains a barchart showing on the left the score of the original algorihtm (in blue), and on the right the score of the proposed algorithm (in teal). If one of the algorithms scores higher, and thus worse, the bar of that algorithm is colored in a lighter color. To increase visibility of the data, the bars in a given column are scaled to the minimum and maximum values present in that column for both algorithms. Thus, the part column is scaled between 3 and 11; the util score column is scaled between 0.940 and 0.995; the seam score column is scaled between 0.130 and 2.270; and the runtime column is scaled between 720 and 61326.

Table 4.1: Results of partitioning each model are shown, for both the original (left bar, blue) and proposed algorithm (right bar, teal). Each bar is scaled between the column's minimum and maximum value. The bar with the highest (worst) value between the two algorithms is made lighter.

Model	Part Score	Util Score	Seam Score	Runtime
A.1 Logo				
A.2 Propeller				
A.3 Pikachu				
A.4 Bracket				
A.5 Gear Spur 1				
A.6 Gear Spur 2				
A.7 Vase				
A.8 Nut				
A.9 Dog				
A.10 Yorkie				
A.11 Gnome				
A.12 Moai				
A.13 Thinker				

Number of Parts. The proposed algorithm often performed equally well or better than the original algorithm with respect to the number of parts used. The original algorithm produced 10, 6, and 8 parts for the Models A.11, A.12, A.13 respectively, while the proposed algorithm produced one less part in the final partition, Figure 4.7 shows this example for Model A.12 (moai). The partition of Model A.3 produced by the original algorithm contained 9 parts, while the proposed algorithm produced 11 parts for this model. A possible explanation is the minor difference in seam score and the way the model is cut, leading to slightly different seam scores which will then lead to slightly different choices being made by the algorithm. This difference in seam score will be further discussed in the paragraph below about seam obtrusiveness.

(a) Partition using the original algorithm. (b) Partition using the proposed algorithm.

Figure 4.7: Model A.12 (moai) cut using the original and proposed algorithms. Notice that the original algorithm uses one more part than the improved algorithm.

Utility. Like the number of parts, the utility score of the proposed version is often equal to or very close to the utility score of the other algorithm. In most cases, the util score is slightly better for the original algorithm when compared to the proposed algorithm. This is the case for Models A.10, and A.12. Meanwhile, the original algorithm has a better utility score for Models A.3, A.6, A.7, A.8, A.11, and A.13. This is logical, as the resulting parts are all roughly the same size and those resulting parts are relatively small when compared to the build volume. Because of this size difference and the fact that f_{seam} only shows the highest difference between part volume and build volume, the value f_{seam} is quite high for each of the models.

Seam Obtrusiveness. The final score to be discussed is the seam obtrusiveness score f_{seam}. In the models which were investigated, this score was often equally good or worse for the proposed algorithm. In the case of Models A.8 and A.4, the seam score was better for the proposed algorithm when compared to the original.

The difference in score between the two algorithms probably originates from the change in the way the cutting is implemented for both algorithms. The proposed algorithm merges a small part of the vertices if they are in the same place before calculating the seam score, while the original does not do this. The implementation calculates the score as the average score for all vertices on the plane multiplied by the seam length, thus if a subset of the vertices is merged they only add less to the average. However, as can be seen from the final scores, this effect is quite small.

(a) Partition using the original algorithm. (b) Partition using the proposed algorithm.

Figure 4.8: Model A.4 (bracket) cut using the original and proposed algorithms. Notice that both algorithms produce comparable results.

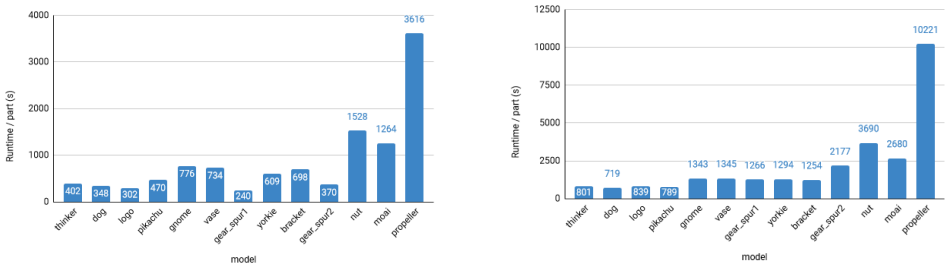
Runtime Let us discuss the difference in runtime between the two algorithms. In Table 4.1 one can clearly see that the runtime is always significantly better for the original algorithm when compared to the proposed algorithm. This is caused by the implementation of the merging of the vertices in the proposed algorithm. Using additional packages like numpy [25] could make this implementation comparable in terms of speed, to the implementation of merging vertices in the original algorithm.

Do note that, even though this implementation detail causes the proposed algorithm to be slower in all cases, the proposed algorithm was expected to be slower because of the amount of possible cuts it is considering. The proposed algorithm would consider each possible combination of connected components on the intersection of a plane and a part, thus leading to many more cuts as explained in Section 3.3. Let us now briefly look at the influence of the number of primitives of the model on the runtime.

Runtime and Primitives. This part will explore the relation between the number of primitives. To this end, the bar plots in Figure 4.9 displays the runtime per part (runtime/f_{part}) for each model, where the models are ordered by the number of triangles. This number was expected to increase. While this effect is somewhat visible in the bar charts as the runtime per part is much higher for models with more vertices, the effect is much less apparent than expected.

This unexpected result can have several reasons. For starters, consider the fact that the first cut is cheaper than every other cut. For the first cut only the original model is considered, while for all subsequent cuts, each of the b partitions has to be cut. Another reason for this behaviour might be that the algorithm has already found a good partition $\mathcal{P}$, but it will only terminate once it has found b partitions that can be printed, i.e. where each part fits in the build volume. Thus the algorithm might perform another cut for the other partitions that do not yet fit, but it will eventually return partition $\mathcal{P}$. This behaviour has been observed to happen several times. A final reason may be that cutting through many triangles, at thick parts of a model, will require more runtime than cutting through thinner parts, with fewer triangles.

Note that a slightly elevated runtime was expected, as the proposed algorithm does consider more possible cuts by considering all possible subsections of the intersection between the model and the plane, as explained in Section 3.2.



(a) Original algorithm.

(b) Proposed algorithm.

Figure 4.9: Models ordered by the number of triangles, showing the runtime divided by the number of parts in the resulting partition.

Qualitative comparison. Finally, some general notes about the visual quality of the models and some general remarks. In terms of quality, the resulting partitions are often the same between both algorithms. However, small differences occur in the choice of cuts, likely because of the minor differences in seam score. This is in line with the expectation, as the proposed version should have more freedom in cutting however it should also be able to perform the same cuts as the original variant.

(a) The partition using the proposed algorithm (see also Figure C.9).

(b) The partition with a plane cutting off its hind legs.

Figure 4.10: Model A.9 (dog) with a plane cutting off its hind legs. You can clearly see that this cut would have not been possible using the original model as it would cut off.

As can be seen in Figure 4.10 the proposed algorithm does perform as expected, being able to cut off only a single part if it would otherwise have to cut off several, however, this behaviour only occurs when a plane cuts off multiple parts at once from a model. Moreover, in the case of the dog, it was possible to subdivide the model such that the original algorithm ended up with the same amount of parts and similar scores. Nevertheless, this case provides evidence that the proposed algorithm allows for greater freedom when choosing cuts.

Chapter 5

Conclusion

This chapter will provide a final conclusion, limitations and possible future improvements to the algorithm as presented in this thesis.

5.1 Conclusion

This thesis explored a novel approach to partitioning models for 3D printing using planes into exactly two parts per cut. To this end, the original algorithm by Luo et al., as well as the proposed algorithm, was implemented to allow for a comparison of the two. Next, the weights of the objective function and the parameters of the proposed algorithm were tuned to achieve optimal results when partitioning models.

As exemplified by the partition in Figure 4.10 this new approach to partitioning works as expected, producing partitions based on planes where a single cut produces exactly two pieces. Now, the additional freedom in choosing cuts does not always provide results of higher quality, with better objective scores. In addition, the proposed algorithm almost always requires more time to run as it considers more possible ways of cutting a model.

5.2 Limitations and Future Work

While the solution presented in this paper allows for more freedom in cuts when compared to the solution presented by Luo et al. [38], this new solution still has several avenues for improvement.

One such limitation is that several objective functions were not included in the implementations of the algorithms to save time in implementing and tuning. Adding these objective functions is expected to improve the quality of the results, and make the results more comparable to the original technique. In addition, both of our implementations lack connectors (pins connecting two parts) as used by Luo et al.

Another limitation is the runtime of the proposed technique. Several parts of the implementation of the proposed technique can benefit from improvements by using third-party packages. In addition, both implementations can be sped up by computing the next partitions in parallel. This parallel computation can even reduce the amount of RAM used as results can be generated in batches, and partitions that are not good enough can be discarded early.

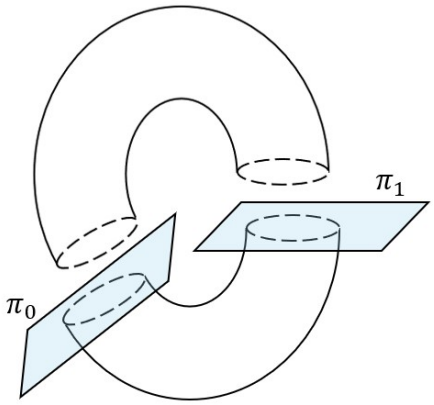


Figure 5.1: Cut a model using two planes and select only a subset of intersections of the model and the plane. In the example, a torus-shaped object is cut using the planes π_0 and π_1. Note that the plane is only shown covering the intersection it is cutting.

Future research could look into new combinations of planar cuts. Consider selecting two or more planes, and cutting the model only on a subset of intersecting components of these two planes, see the example in Figure 5.1. The cutting method included in the proposed technique allows cutting only indicated faces, thus with only minimal improvements such a cut can be made. Note however that the result of such a cut must still be assemblable, especially when connectors are added to the model.

Additionally, the proposed technique does not check whether a cut will result in exactly two parts before cutting. Instead, the proposed technique performs the cut and only then checks whether the cut results in two parts. This may lead to (possibly many) unnecessary cuts being performed which do not result in exactly two parts. Especially for large models, this can increase the runtime dramatically. If a check, to test whether a cut will result in two parts, can be performed efficiently, then such an improvement can positively affect the runtime.

Bibliography

[1] 3dgadgets. Anycubic photon mono 4k monochrome msla resin 3d printer. www.3dgadgets.my/resin-3d-printer/1297-anycubic-photon-mono-4k-monochrome-msla-resin-3d-printer.html. [Online; accessed August 8, 2024].

[2] Bijan Masood Abar, Cambre Kelly, Nicholas B. Allen, and Ken Gall. 1 - historical perspectives on 3d printing. In Peter D. Highlander, editor, *Clinical Applications of 3D Printing in Foot and Ankle Surgery (First Edition)*, pages 1–16. Elsevier, New Delhi, first edition edition, 2024.

[3] Seth Allen and Deba Dutta. Determination and evaluation of support structures in layered manufacturing. *Journal of Design and Manufacturing*, 5:153–162, 1995.

[4] Anycubic. Anycubic photon workshop 3d slicer software (v3.1.4). store.anycubic.com/pages/anycubic-photon-workshop-3d-slicer-software. [Online; accessed August 8, 2024].

[5] Anycubic. Photon mono 4k. store.anycubic.com/products/photon-mono-4k. [Online; accessed June 21, 2024].

[6] Subcommittee F42 ASTM International. Standard terminology for additive manufacturing technologies (f42 on additive manufacturing technologies. subcommittee f42. 91 on terminology). Standard F2792 - 12a, ASTM International, 2012.

[7] Marco Attene. Direct repair of self-intersecting meshes. *Graphical Models*, 76(6):658–668, 2014.

[8] aubenc (Thingiverse). Poor man's openscad screw library. www.thingiverse.com/thing:8796, May 25th, 2011. [Online; accessed June 10, 2024].

[9] Gavin Barill, Neil G. Dickson, Ryan Schmidt, David I. W. Levin, and Alec Jacobson. Fast winding numbers for soups and clouds. *ACM Trans. Graph.*, 37(4), jul 2018.

[10] Kumar Chalsani, Larry Jones, and Larry Roscoe. Support generation for fused deposition modeling. In *1995 International Solid Freeform Fabrication Symposium*, 1995.

[11] P.W.D. Charles. pychop3d. www.github.com/gregstarr/pychop3d, 2024.

[12] Jerome Charton, Stephen Baek, and Youngjun Kim. Mesh repairing using topology graphs. *Journal of Computational Design and Engineering*, 8(1):251–267, 2021.

[13] Charlie CL Wang and Yong Chen. Thickening freeform surfaces for solid fabrication. *Rapid Prototyping Journal*, 19(6):395–406, 2013.

[14] coco__ (Thingiverse). propeller. `www.thingiverse.com/thing:2206869`, March 27th, 2017. [Online; accessed June 10, 2024].

[15] Blender Online Community. *Blender (v3.4.1) - a 3D modelling and rendering package*. Blender Foundation, Stichting Blender Foundation, Amsterdam, 2018.

[16] Andrew Dawood, B Marti Marti, Veronique Sauret-Jackson, and Alastair Darwood. 3d printing in dentistry. *British Dental Journal*, 219(11):521–529, 2015.

[17] M. Dawson-Haggerty. Trimesh 4.4.1 documentation. `www.trimesh.org/`. [Online; accessed June 24, 2024].

[18] H Quynh Dinh, Filipp Gelman, Sylvain Lefebvre, and Frédéric Claux. Modeling and toolpath generation for consumer-level 3d printing. In *ACM SIGGRAPH 2015 Courses*, pages 1–273. Inria, 2015.

[19] André Dolenc and Ismo Mäkelä. Slicing procedures for layered manufacturing techniques. *Computer-Aided Design*, 26(2):119–126, 1994.

[20] flowalistik (Thingiverse). Low-poly pikachu. `www.thingiverse.com/thing:376601`, June 27th, 2014. [Online; accessed June 10, 2024].

[21] fokuspace (Thingiverse). Dog low poly york. `www.thingiverse.com/thing:2579348`, October 11th, 2017. [Online; accessed June 10, 2024].

[22] Henry Fuchs, Zvi M Kedem, and Bruce F Naylor. On visible surface generation by a priori tree structures. In *Proceedings of the 7th Annual Conference on Computer Graphics and Interactive Techniques*, pages 124–133, 1980.

[23] Rey Farly Garcia and Alvin Chua. High compressive strength 3d printed infill based on strut-based lattice structure. *ASEAN Engineering Journal*, 12:89–94, 11 2022.

[24] Zeming Gu, Jianzhong Fu, Hui Lin, and Yong He. Development of 3d bioprinting: From printing methods to biomedical applications. *Asian Journal of Pharmaceutical Sciences*, 15(5):529–557, 2020.

[25] Charles R. Harris, K. Jarrod Millman, Stéfan J. van der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J. Smith, Robert Kern, Matti Picus, Stephan Hoyer, Marten H. van Kerkwijk, Matthew Brett, Allan Haldane, Jaime Fernández del Río, Mark Wiebe, Pearu Peterson, Pierre Gérard-Marchant, Kevin Sheppard, Tyler Reddy, Warren Weckesser, Hameer Abbasi, Christoph Gohlke, and Travis E. Oliphant. Array programming with NumPy. *Nature*, 585(7825):357–362, September 2020.

[26] Martin Held, Gábor Lukács, and László Andor. Pocket machining based on contour-parallel tool paths generated by means of proximity maps. *Computer-Aided Design*, 26(3):189–203, 1994.

[27] Kristian Hildebrand, Bernd Bickel, and Marc Alexa. Orthogonal slicing for additive manufacturing. *Computers & Graphics*, 37(6):669–675, 2013.

[28] Md Aslam Hossain, Altynay Zhumabekova, Suvash Chandra Paul, and Jong Ryeol Kim. A review of 3d printing in construction and its impact on the labor market. *Sustainability*, 12(20):8492, 2020.

[29] Pu Huang, Charlie CL Wang, and Yong Chen. Intersection-free and topologically faithful slicing of implicit solid. *Journal of Computing and Information Science in Engineering*, 13(2):021009, 2013.

[30] Abigail Isaacson, Stephen Swioklo, and Che J Connon. 3d bioprinting of a corneal stroma equivalent. *Experimental Eye Research*, 173:188–193, 2018.

[31] A. Jacobson. libigl python bindings. `www.github.com/libigl/libigl-python-bindings`, 2024.

[32] Ju-Hsien Kao and Fritz B Prinz. Optimal motion planning for deposition in layered manufacturing. In *International Design Engineering Technical Conferences and Computers and Information in Engineering Conference*, volume 80364, page V006T06A018. American Society of Mechanical Engineers, 1998.

[33] E. Karasik, R. Fattal, and M. Werman. Object partitioning for support-free 3d-printing. *Computer Graphics Forum*, 38(2):305–316, 2019.

[34] CF Kirschman and CC Jara-Almonte. A parallel slicing algorithm for solid freeform fabrication processes. In *1992 International Solid Freeform Fabrication Symposium*, pages 26–33, 1992.

[35] Sylvain Lefebvre and LORIA-INRIA Nancy Grand-Est. Icesl: A gpu accelerated csg modeler and slicer. In *18th European Forum on Additive Manufacturing (AEFA'13)*, volume 1, page 3, 2013.

[36] KF Leong, CK Chua, and YM Ng. A study of stereolithography file errors and repair. part 1. generic solution. *The International Journal of Advanced Manufacturing Technology*, 12:407–414, 1996.

[37] Marco Livesu, Stefano Ellero, Jonàs Martínez, Sylvain Lefebvre, and Marco Attene. From 3d models to 3d prints: An overview of the processing pipeline. *Computer Graphics Forum*, 36(2):537–564, 2017.

[38] Linjie Luo, Ilya Baran, Szymon Rusinkiewicz, and Wojciech Matusik. Chopper: Partitioning models into 3d-printable parts. *ACM Transactions on Graphics (TOG)*, 31(6):1–9, 2012.

[39] mce076 (Thingiverse). Corner brackets, system20_bracket_24mm. www.thingiverse.com/thing:2957550, June 13th, 2018. [Online; accessed June 10, 2024].

[40] Sara McMains and Carlo Séquin. A coherent sweep plane slicer for layered manufacturing. In *Proceedings of the Fifth ACM Symposium on Solid Modelling and Applications*, pages 285–295, 1999.

[41] Sara McMains, Jordan Smith, Jianlin Wang, and Carlo Sequin. Layered manufacturing of thin-walled parts. In *International Design Engineering Technical Conferences and Computers and Information in Engineering Conference*, volume 35128, pages 379–387. American Society of Mechanical Engineers, 2000.

[42] Pulak Mohan Pandey, N Venkata Reddy, and Sanjay G Dhande. Slicing procedures in layered manufacturing: a review. *Rapid Prototyping Journal*, 9(5):274–288, 2003.

[43] Spandana Paritala, Kailash Kumar Singaram, Indira Bathina, Mohd Ataullah Khan, and Sri Kalyana Rama Jyosyula. Rheology and pumpability of mix suitable for extrusion-based concrete 3d printing – a review. *Construction and Building Materials*, 402:132962, 2023.

[44] Emiliano Pérez, Santiago Salamanca, Pilar Merchán, and Antonio Adán. A comparison of hole-filling methods in 3d. *International Journal of Applied Mathematics and Computer Science*, 26(4), 2016.

[45] Di Qi, Long Zeng, and Matthew MF Yuen. Robust slicing procedure based on surfel-grid. *Computer-Aided Design and Applications*, 10(6):965–981, 2013.

[46] Jordan R Raney and Jennifer A Lewis. Printing mesoscale architectures. *Mrs Bulletin*, 40(11):943–950, 2015.

[47] riskable (Thingiverse). Low-poly rose twist vase. www.thingiverse.com/thing:2173745, March 13th, 2017. [Online; accessed June 10, 2024].

[48] saleem145 (Thingiverse). Gears, spur4. www.thingiverse.com/thing:214043, December 27th, 2013. [Online; accessed June 10, 2024].

[49] Bolugoddu Sandeep, TTM Kannan, J Chandradass, M Ganesan, and A John Rajan. Scope of 3d printing in manufacturing industries-a review. *Materials Today: Proceedings*, 45:6941–6945, 2021.

[50] John F Sargent and RX Schwartz. *3D printing: Overview, impacts, and the federal role*. Congressional Research Service, 2019.

[51] Jyrki Savolainen and Mikael Collan. How additive manufacturing technology changes business models? – review of literature. *Additive Manufacturing*, 32:101070, 2020.

[52] slavikk (Thingiverse). Low poly moai. www.thingiverse.com/thing:908062, July 2nd, 2015. [Online; accessed June 10, 2024].

[53] Hongtao Song, Nicholas A Rodriguez, Carolyn Conner Seepersad, Richard H Crawford, Morgan Chen, and Eric B Duoss. Development of a variable tensioning system to reduce separation force in large scale stereolithography. *Additive Manufacturing*, 38:101816, 2021.

[54] John C Steuben, Athanasios P Iliopoulos, and John G Michopoulos. Implicit slicing for functionally tailored additive manufacturing. *Computer-Aided Design*, 77:107–119, 2016.

[55] Young Seok Suh and Michael J Wozny. Adaptive slicing of solid freeform fabrication processes. In *1994 International Solid Freeform Fabrication Symposium*, 1994.

[56] Greymi Tan, Nicole Ioannou, Essyrose Mathew, Aristides D Tagalakis, Dimitrios A Lamprou, and Cynthia Yu-Wai-Man. 3d printing in ophthalmology: From medical implants to personalised medicine. *International Journal of Pharmaceutics*, 625:122094, 2022.

[57] AndrewSink (Thingiverse). Low poly dog! `www.thingiverse.com/thing:2797399`, February 17th, 2018. [Online; accessed June 10, 2024].

[58] FunbieStudios (Thingiverse). Low poly gnome. `www.thingiverse.com/thing:149845`, September 12th, 2013. [Online; accessed June 10, 2024].

[59] LxO (Thingiverse). Lowest poly the thinker. `www.thingiverse.com/thing:2164071`, March 9th, 2017. [Online; accessed June 10, 2024].

[60] Justin Tyberg and Jan Helge Bøhn. Local adaptive slicing. *Rapid Prototyping Journal*, 4(3):118–127, 1998.

[61] Weiming Wang, Haiyuan Chao, Jing Tong, Zhouwang Yang, Xin Tong, Hang Li, Xiuping Liu, and Ligang Liu. Saliency-preserving slicing optimization for effective 3d printing. In *Computer Graphics Forum*, volume 34, pages 148–160. Wiley Online Library, 2015.

[62] Weiming Wang, Tuanfeng Y Wang, Zhouwang Yang, Ligang Liu, Xin Tong, Weihua Tong, Jiansong Deng, Falai Chen, and Xiuping Liu. Cost-effective printing of 3d objects with skin-frame structures. *ACM Transactions on Graphics (ToG)*, 32(6):1–10, 2013.

[63] Weiming M Wang, Cédric Zanni, and Leif Kobbelt. Improved surface quality in 3d printing by optimizing the printing direction. In *Computer Graphics Forum*, volume 35, pages 59–70. Wiley Online Library, 2016.

[64] Xiangzhi Wei, Siqi Qiu, Lin Zhu, Ruiliang Feng, Yaobin Tian, Juntong Xi, and Youyi Zheng. Toward support-free 3d printing: A skeletal approach for partitioning models. *IEEE Transactions on Visualization and Computer Graphics*, 24(10):2799–2812, 2018.

[65] Jun Wu, Charlie CL Wang, Xiaoting Zhang, and Rüdiger Westermann. Self-supporting rhombic infill structures for additive manufacturing. *Computer-Aided Design*, 80:32–42, 2016.

[66] Y Yang, Han Tong Loh, JYH Fuh, and YG Wang. Equidistant path generation for improving scanning efficiency in layered manufacturing. *Rapid Prototyping Journal*, 8(1):30–37, 2002.

[67] Patil Yogesh, Patil Richa, NS Chandrashekhar, and KP Karunakaran. Layer separation mechanisms in dlp 3d printing. In *Advances in Additive Manufacturing and Joining: Proceedings of AIMTDR 2018*, pages 179–187. Springer, 2019.

[68] Zhengyan Zhang and Sanjay Joshi. An improved slicing algorithm with efficient contour construction using stl files. *The International Journal of Advanced Manufacturing Technology*, 80:1347–1362, 2015.

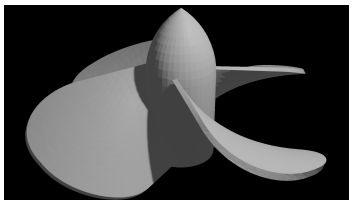
[69] WM Zhu and Kai Ming Yu. Dexel-based direct slicing of multi-material assemblies. *The International Journal of Advanced Manufacturing Technology*, 18:285–302, 2001.

Appendix A

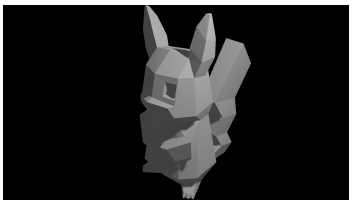
Models used for Testing

This appendix will mention all models used for testing in Chapter 4. The files of each model can be found at `https://github.com/PFvanBeerendonk/final_thesis__partitioning/models`. All models created by the author were created in Blender version 1.3.4 [15]. Table A.1 gives some general statistics for each model, note that all models are triangulated so the number of faces is equal to the number of triangles.

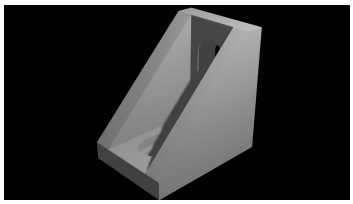
Model A.1: *Logo* (created by the author)



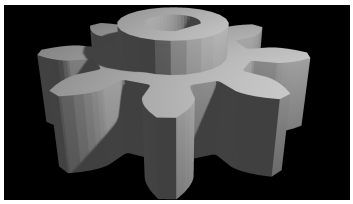
Model A.2: *Propeller* (created by coco__ [14])



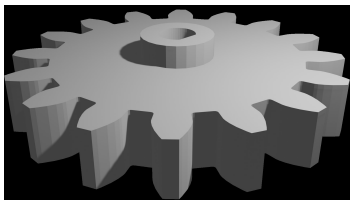
Model A.3: *Pikachu* (created by the flowalistik [20])



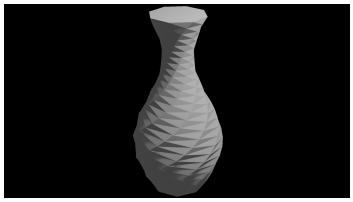
Model A.4: *Bracket* (created by mce076 [39])



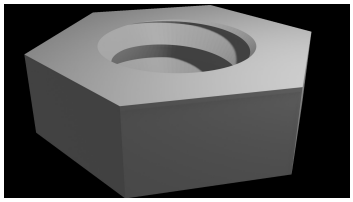
Model A.5: *Gear Spur 1* (created by saleem145 [48])



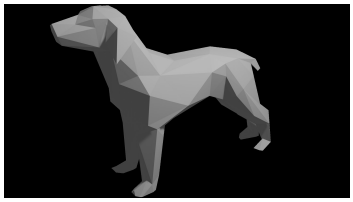
Model A.6: *Gear Spur 2* (created by saleem145 [48])



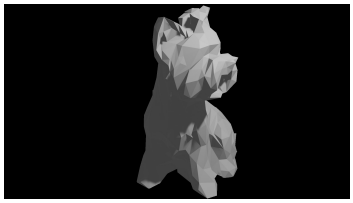
Model A.7: *Vase* (created by riskable [47])



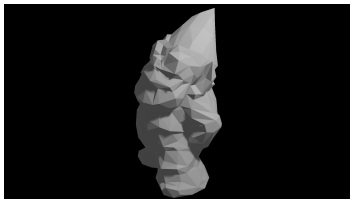
Model A.8: *Nut* (created by the aubenc [8])



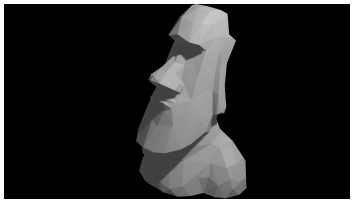
Model A.9: *Dog* (created by AndrewSink [57])



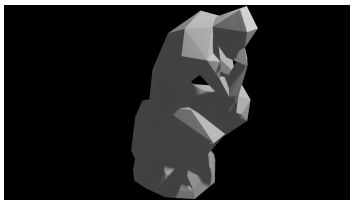
Model A.10: *Yorkie* (created by fokuspace [21])



Model A.11: *Gnome* (created by FunbieStudios [58])



Model A.12: *Moai* (created by slavikk [52])



Model A.13: *Thinker* (created by LxO [59])

Table A.1: For each model, the dimensions of its minimum oriented bounding box (in mm), its volume (in mm^3), and the number of primitives are noted.

Model	**Dimensions Minimum Oriented Bounding Box**			**Volume**	**Primitive counts**		
	Height	**Depth**	**Width**		**Vertices**	**Edges**	**Faces**
A.1 Logo	35.664	94.298	202.677	301615	202	1212	404
A.2 Propeller	89.585	151.377	154.396	142902	3289	19722	6574
A.3 Pikachu (low-poly)	94.683	145.031	197.980	625394	208	1236	412
A.4 Bracket	75.807	106.130	106.130	319316	688	4140	1380
A.5 Gear Spur 1	39.500	93.144	93.144	132193	510	3060	1020
A.6 Gear Spur 2	22.500	99.967	99.967	98286	768	4608	1536
A.7 Vase	96.552	96.552	198.165	700162	496	2964	988
A.8 Nut	36.000	108.000	124.708	254540	1251	7506	2502
A.9 Dog	43.889	131.489	151.925	152181	179	1062	354
A.10 Yorkie	91.737	115.784	176.781	576573	551	3294	1098
A.11 Gnome	102.649	122.257	204.705	939334	452	2700	900
A.12 Moai	114.999	116.931	178.679	707858	1508	9036	3012
A.13 Thinker	100.189	107.304	235.588	709760	122	768	256

Appendix B

Tuning Results

This appendix will show the results of tuning the weights and parameters of the proposed algorithm as discussed in Sections 4.2 and 4.3 of Chapter 4. The actual results of each run of the algorithm during tuning, including generated partitions, can be found at `https://github.com/PFvanBeerendonk/final_thesis__partitioning/results`.

Table B.1: Results of tuning the parameter weights α_{util} and α_{seam} for Model A.1 (tue logo). For each combination of weights, the scores are rounded to 3 decimals and the runtime is rounded to seconds.

α_{util} \ α_{seam}		**0**	**0.0001**	**0.001**	**0.01**	**0.1**
0	f_{part} = f_{util} = f_{seam} = time =	$f_{\text{part}} > 10$ time $>$ 2h	*Not Tested*	*Not Tested*	*Not Tested*	*Not Tested*
0.0001	f_{part} = f_{util} = f_{seam} = time =	6 0.974 0.862 3654	$f_{\text{part}} > 10$ time $>$ 2h	*Not Tested*	*Not Tested*	*Not Tested*
0.0005	f_{part} = f_{util} = f_{seam} = time =	6 0.974 0.862 3566	4 0.980 0.184 2292	*Not Tested*	*Not Tested*	*Not Tested*
0.001	f_{part} = f_{util} = f_{seam} = time =	6 0.974 0.862 3584	4 0.971 0.274 2516	$f_{\text{part}} > 10$ time $>$ 2h	*Not Tested*	*Not Tested*
0.005	f_{part} = f_{util} = f_{seam} = time =	6 0.974 0.862 3564	6 0.974 0.298 3500	4 0.980 0.184 2333	*Not Tested*	*Not Tested*
0.01	f_{part} = f_{util} = f_{seam} = time =	6 0.974 0.862 3576	6 0.974 0.305 3676	4 0.971 0.274 2436	$f_{\text{part}} > 10$ time $>$ 2h	*Not Tested*
0.05	f_{part} = f_{util} = f_{seam} = time =	6 0.974 0.862 3564	6 0.974 0.405 3532	6 0.974 0.298 3526	4 0.980 0.184 2335	*Not Tested*
0.1	f_{part} = f_{util} = f_{seam} = time =	6 0.974 0.862 3659	6 0.974 0.481 3583	6 0.974 0.305 3509	6 0.971 0.274 2520	$f_{\text{part}} > 10$ time $>$ 2h
0.15	f_{part} = f_{util} = f_{seam} = time =	6 0.974 0.862 3581	6 0.974 0.481 3605	6 0.974 0.316 3429	6 0.974 0.302 3540	$f_{\text{part}} > 10$ time $>$ 2h
0.2	f_{part} = f_{util} = f_{seam} = time =	6 0.974 0.862 3484	6 0.974 0.479 3658	6 0.974 0.311 3585	6 0.974 0.302 3492	$f_{\text{part}} > 10$ time $>$ 2h

Table B.2: Results of tuning the parameter weights α_{util} and α_{seam} for Model A.2 (propeller). For each combination of weights, the scores are rounded to 3 decimals and the runtime is rounded to seconds.

α_{util} \ α_{seam}	**0**	**0.0001**	**0.001**	**0.01**	**0.1**
0	f_{part} = f_{util} = f_{seam} = time =	$f_{\text{part}} > 12$ time $>$ 42h	*Not Tested*	*Not Tested*	*Not Tested*	*Not Tested*
0.0001	f_{part} = f_{util} = f_{seam} = time =	6 0.995 1.451 38462	$f_{\text{part}} > 12$ time $>$ 32h	*Not Tested*	*Not Tested*	*Not Tested*
0.0005	f_{part} = f_{util} = f_{seam} = time =	6 0.995 1.451 37985	$f_{\text{part}} > 12$ time $>$ 42h	*Not Tested*	*Not Tested*	*Not Tested*
0.001	f_{part} = f_{util} = f_{seam} = time =	6 0.995 1.451 37923	$f_{\text{part}} > 12$ time $>$ 42h	*Not Tested*	*Not Tested*	*Not Tested*
0.005	f_{part} = f_{util} = f_{seam} = time =	6 0.995 1.451 37367	$f_{\text{part}} > 12$ time $>$ 41h	$f_{\text{part}} > 12$ time $>$ 42h	*Not Tested*	*Not Tested*
0.01	f_{part} = f_{util} = f_{seam} = time =	6 0.995 1.451 37181	6 0.995 0.603 51013	$f_{\text{part}} > 12$ time $>$ 41h	*Not Tested*	*Not Tested*
0.05	f_{part} = f_{util} = f_{seam} = time =	6 0.995 1.451 37257	6 0.995 0.632 38573	$f_{\text{part}} > 12$ time $>$ 41h	$f_{\text{part}} > 12$ time $>$ 42h	*Not Tested*
0.1	f_{part} = f_{util} = f_{seam} = time =	6 0.995 1.451 38418	6 0.994 1.082 38162	6 0.995 0.603 50721	$f_{\text{part}} > 12$ time $>$ 42h	*Not Tested*
0.15	f_{part} = f_{util} = f_{seam} = time =	6 0.995 1.451 37986	6 0.994 1.108 36685	6 0.995 0.632 38929	$f_{\text{part}} > 12$ time $>$ 42h	*Not Tested*
0.2	f_{part} = f_{util} = f_{seam} = time =	6 0.995 1.451 36863	6 0.994 1.108 36785	6 0.995 0.632 39230	$f_{\text{part}} > 12$ time $>$ 42h	*Not Tested*

Table B.3: Results of tuning the parameters beam width (b), and the distance between uniformly spaced planes (d_π) for Model A.1 (logo). For each combination of weights, the scores are rounded to 3 decimals and the runtime is rounded to seconds.

b \ d_π		**7**	**6**	**5**	**4**	**3**	**2**	**1**
1	f_{part} =	6	6	6	7	6	6	6
	f_{util} =	0.977	0.975	0.974	0.982	0.975	0.974	0.974
	f_{seam} =	0.470	0.310	0.501	0.504	0.375	0.455	0.476
	time =	1138	1215	1511	2051	2639	3712	8060
2	f_{part} =	8	6	6	7	7	6	6
	f_{util} =	0.985	0.975	0.974	0.982	0.985	0.974	0.976
	f_{seam} =	0.802	0.327	0.477	0.508	0.576	0.446	0.504
	time =	2286	2062	2543	3461	4800	6425	13868
3	f_{part} =	5	6	6	5	7	6	6
	f_{util} =	0.973	0.974	0.974	0.973	0.985	0.974	0.979
	f_{seam} =	0.263	0.351	0.305	0.258	0.572	0.287	0.582
	time =	1885	2978	3509	3602	6879	8825	18228
4	f_{part} =	5	6	6	5	6	5	6
	f_{util} =	0.973	0.974	0.974	0.973	0.974	0.973	0.974
	f_{seam} =	0.265	0.348	0.311	0.257	0.294	0.265	0.293
	time =	2445	3839	4499	4139	7657	9292	23323
5	f_{part} =	5	6	6	5	5	5	5
	f_{util} =	0.973	0.974	0.974	0.973	0.973	0.973	0.973
	f_{seam} =	0.264	0.351	0.316	0.257	0.252	0.256	0.248
	time =	3030	4744	5670	5097	6818	11500	20396

Table B.4: Results of tuning the parameters beam width (b), and the distance between uniformly spaced planes (d_π) for Model A.2 (propeller). For each combination of weights, the scores are rounded to 3 decimals and the runtime is rounded to seconds.

b \ d_π		**7**	**6**	**5**	**4**	**3**	**2**	**1**
1	f_{part} =	6	6	6	6	6		
	f_{util} =	0.995	0.995	0.995	0.995	0.995	*Insufficient*	*Insufficient*
	f_{seam} =	0.674	0.653	0.618	0.625	0.614	*RAM*	*RAM*
	time =	10901	12997	17438	19470	27731		
2	f_{part} =	6	6	6	6	6		
	f_{util} =	0.995	0.995	0.995	0.995	0.995	*Insufficient*	*Insufficient*
	f_{seam} =	0.632	0.650	0.630	0.627	0.617	*RAM*	*RAM*
	time =	18660	22401	29657	32589	44632		
3	f_{part} =	6	6	6	6	6		
	f_{util} =	0.995	0.995	0.995	0.995	0.995	*Insufficient*	*Insufficient*
	f_{seam} =	0.670	0.620	0.603	0.624	0.628	*RAM*	*RAM*
	time =	27465	30859	50721	46915	64381		
4	f_{part} =	6	6	6	6	6		
	f_{util} =	0.995	0.995	0.995	0.995	0.995	*Insufficient*	*Insufficient*
	f_{seam} =	0.620	0.647	0.644	0.628	0.626	*RAM*	*RAM*
	time =	33908	40744	53526	61326	83673		
5	f_{part} =	6	6	6	6	6		
	f_{util} =	0.995	0.995	0.995	0.995	0.995	*Insufficient*	*Insufficient*
	f_{seam} =	0.637	0.639	0.621	0.625	0.610	*RAM*	*RAM*
	time =	42602	49904	63062	73975	98188		

Table B.5: Results of tuning the parameters $\angle_{\text{diff}}$ and d_{diff} for Model A.1 (logo). For each combination of weights, the scores are rounded to 3 decimals and the runtime is rounded to seconds.

$\angle_{\text{diff}}$ \ d_{diff}		**0**	**0.05**	**0.1**	**0.15**	**0.2**	**0.5**
0	f_{part} =	5	5	5	5	5	5
	f_{util} =	0.973	0.973	0.973	0.973	0.973	0.973
	f_{seam} =	0.257	0.257	0.257	0.257	0.257	0.257
	time =	4163	4171	4152	4171	4163	4138
0.05	f_{part} =	5	5	5	5	5	5
	f_{util} =	0.973	0.973	0.973	0.973	0.973	0.973
	f_{seam} =	0.257	0.257	0.257	0.257	0.257	0.257
	time =	4136	4153	4200	4115	4181	4163
0.1	f_{part} =	5	5	5	5	5	5
	f_{util} =	0.973	0.973	0.973	0.973	0.973	0.973
	f_{seam} =	0.257	0.257	0.257	0.257	0.257	0.257
	time =	4249	4111	4139	4150	4188	4335
0.15	f_{part} =	5	5	5	5	5	5
	f_{util} =	0.973	0.973	0.973	0.973	0.973	0.973
	f_{seam} =	0.257	0.257	0.257	0.257	0.257	0.257
	time =	4159	4140	4200	4176	4131	4397
0.2	f_{part} =	5	5	5	5	5	5
	f_{util} =	0.973	0.973	0.973	0.973	0.973	0.973
	f_{seam} =	0.257	0.257	0.257	0.257	0.257	0.257
	time =	4248	4168	4181	4136	4083	4110
0.5	f_{part} =	5	5	5	5	5	5
	f_{util} =	0.973	0.973	0.973	0.973	0.973	0.973
	f_{seam} =	0.257	0.257	0.257	0.257	0.257	0.257
	time =	4157	4215	4162	4167	4157	4148

Table B.6: Results of tuning the parameters $\angle_{\text{diff}}$ and d_{diff} for Model A.2 (propeller). For each combination of weights, the scores are rounded to 3 decimals and the runtime is rounded to seconds.

$\angle_{\text{diff}}$ \ d_{diff}		**0**	**0.05**	**0.1**	**0.15**	**0.2**	**0.5**
0	f_{part} =	6	6	6	6	6	6
	f_{util} =	0.995	0.995	0.995	0.995	0.995	0.995
	f_{seam} =	0.628	0.628	0.628	0.628	0.628	0.628
	time =	60864	60613	60623	60864	60772	60451
0.05	f_{part} =	6	6	6	6	6	6
	f_{util} =	0.995	0.995	0.995	0.995	0.995	0.995
	f_{seam} =	0.628	0.628	0.628	0.628	0.628	0.628
	time =	60489	60945	61949	61063	61667	60429
0.1	f_{part} =	6	6	6	6	6	6
	f_{util} =	0.995	0.995	0.995	0.995	0.995	0.995
	f_{seam} =	0.628	0.628	0.628	0.628	0.628	0.628
	time =	60659	60206	61326	60672	59967	62693
0.15	f_{part} =	6	6	6	6	6	6
	f_{util} =	0.995	0.995	0.995	0.995	0.995	0.995
	f_{seam} =	0.628	0.628	0.628	0.628	0.628	0.628
	time =	60511	61458	60654	61121	60476	62170
0.2	f_{part} =	6	6	6	6	6	6
	f_{util} =	0.995	0.995	0.995	0.995	0.995	0.995
	f_{seam} =	0.628	0.628	0.628	0.628	0.628	0.628
	time =	62355	61754	61215	60911	60000	60614
0.5	f_{part} =	6	6	6	6	6	6
	f_{util} =	0.995	0.995	0.995	0.995	0.995	0.995
	f_{seam} =	0.628	0.628	0.628	0.628	0.628	0.628
	time =	61000	61320	60607	60741	60688	60498

Appendix C

Partition Results

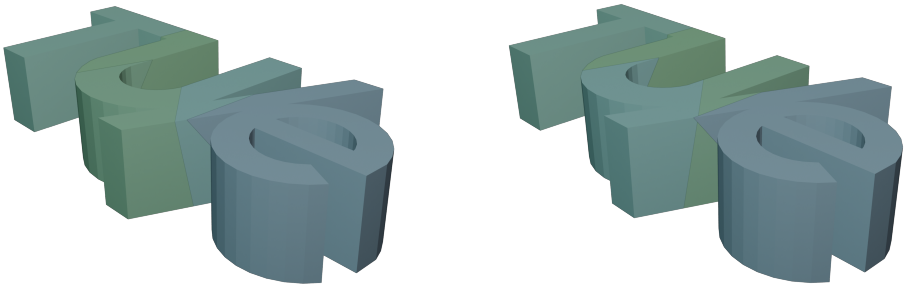
This appendix contains the results as described in Section 4.4 of Chapter 4. In Table C.1 the results for each of these partitions are included. For each model, a figure showing the partition resulting from the original and proposed algorithm is included. The actual results of each run of the algorithm, including generated partitions, can be found at `www.github.com/PFvanBeerendonk/final_thesis__partitioning/results/final`

Table C.1: Results of partitioning each model are shown, for both the original and proposed algorithm. The scores are rounded to 3 decimals and the runtime is rounded to seconds.

Model		**Original Algorithm**	**Proposed Algorithm**
A.1 Logo	$f_{\text{part}} =$	5	5
	$f_{\text{util}} =$	0.973	0.973
	$f_{\text{seam}} =$	0.169	0.257
	time =	1509	4194
A.2 Propeller	$f_{\text{part}} =$	6	6
	$f_{\text{util}} =$	0.995	0.995
	$f_{\text{seam}} =$	0.601	0.628
	time =	21698	61326
A.3 Pikachu	$f_{\text{part}} =$	9	11
	$f_{\text{util}} =$	0.974	0.985
	$f_{\text{seam}} =$	0.463	0.872
	time =	4226	8675
A.4 Bracket	$f_{\text{part}} =$	3	3
	$f_{\text{util}} =$	0.940	0.940
	$f_{\text{seam}} =$	2.270	1.082
	time =	2094	3763
A.5 Gear Spur 1	$f_{\text{part}} =$	3	3
	$f_{\text{util}} =$	0.975	0.975
	$f_{\text{seam}} =$	0.304	0.432
	time =	720	3798

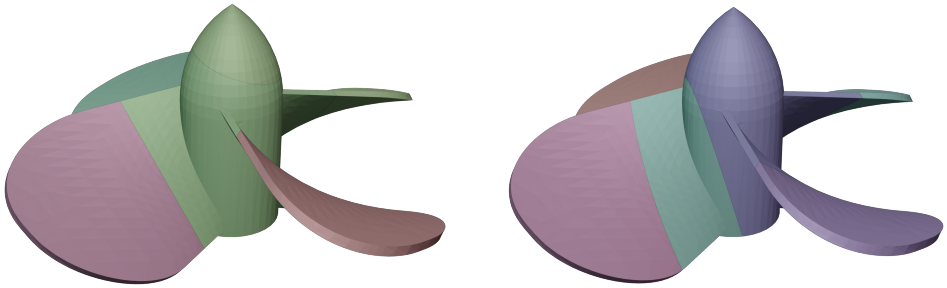
Table C.2: (Table C.1 continued).

Model		Original Algorithm	Proposed Algorithm
A.6 Gear Spur 2	$f_{\text{part}} =$	3	3
	$f_{\text{util}} =$	0.982	0.983
	$f_{\text{seam}} =$	0.304	0.563
	time =	1111	6530
A.7 Vase	$f_{\text{part}} =$	7	8
	$f_{\text{util}} =$	0.964	0.967
	$f_{\text{seam}} =$	0.130	0.599
	time =	5137	10758
A.8 Nut	$f_{\text{part}} =$	6	6
	$f_{\text{util}} =$	0.978	0.982
	$f_{\text{seam}} =$	1.882	0.832
	time =	5137	22140
A.9 Dog	$f_{\text{part}} =$	4	4
	$f_{\text{util}} =$	0.983	0.983
	$f_{\text{seam}} =$	0.241	0.285
	time =	1393	2876
A.10 Yorkie	$f_{\text{part}} =$	5	5
	$f_{\text{util}} =$	0.944	0.943
	$f_{\text{seam}} =$	0.433	0.495
	time =	3047	6472
A.11 Gnome	$f_{\text{part}} =$	10	9
	$f_{\text{util}} =$	0.958	0.965
	$f_{\text{seam}} =$	0.680	0.993
	time =	7758	12086
A.12 Moai	$f_{\text{part}} =$	6	5
	$f_{\text{util}} =$	0.951	0.941
	$f_{\text{seam}} =$	0.334	0.441
	time =	7584	13398
A.13 Thinker	$f_{\text{part}} =$	8	7
	$f_{\text{util}} =$	0.952	0.995
	$f_{\text{seam}} =$	0.620	0.782
	time =	3214	5604



(a) Original Algorithm. (b) Proposed Algorithm.

Figure C.1: Model A.1 (logo).



(a) Original Algorithm. (b) Proposed Algorithm.

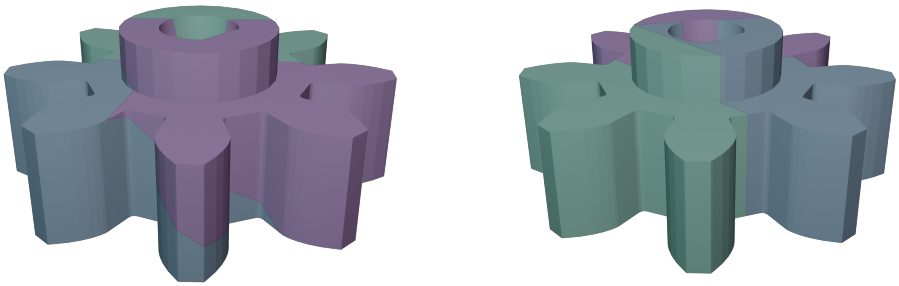
Figure C.2: Model A.2 (propeller).

(a) Original Algorithm. (b) Proposed Algorithm.

Figure C.3: Model A.3 (pikachu).

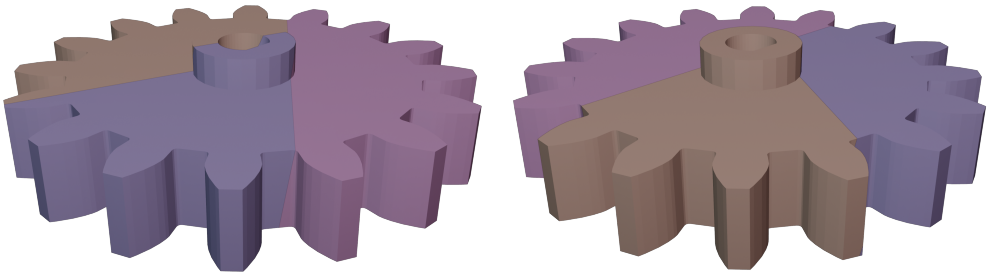
(a) Original Algorithm. (b) Proposed Algorithm.

Figure C.4: Model A.4 (bracket).



(a) Original Algorithm. (b) Proposed Algorithm.

Figure C.5: Model A.5 (gear spur 1).



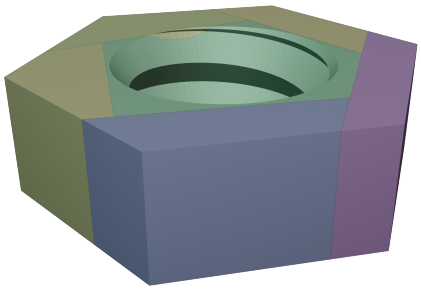
(a) Original Algorithm. (b) Proposed Algorithm.

Figure C.6: Model A.6 (gear spur 2).

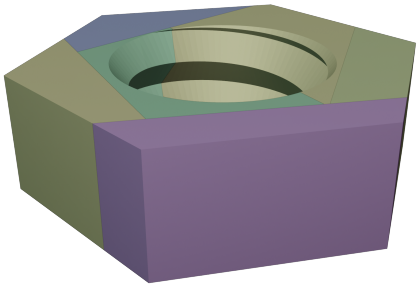
(a) Original Algorithm.

(b) Proposed Algorithm.

Figure C.7: Model A.7 (vase).

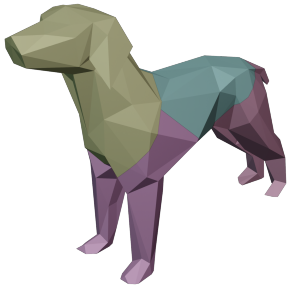


(a) Original Algorithm.

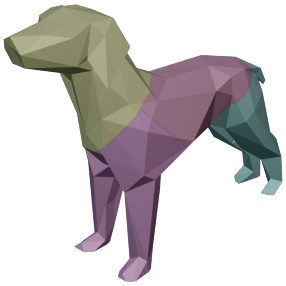


(b) Proposed Algorithm.

Figure C.8: Model A.8 (nut).



(a) Original Algorithm.



(b) Proposed Algorithm.

Figure C.9: Model A.9 (dog).

(a) Original Algorithm.

(b) Proposed Algorithm.

Figure C.10: Model A.10 (yorkie).

(a) Original Algorithm.

(b) Proposed Algorithm.

Figure C.11: Model A.11 (gnome).

(a) Original Algorithm.

(b) Proposed Algorithm.

Figure C.12: Model A.12 (moai).

(a) Original Algorithm.

(b) Proposed Algorithm.

Figure C.13: Model A.13 (thinker).