

Software Simulation of 5-Axis CNC Milling using Multidirectional Heightmaps

by

Marshall Hahn

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 2010

© Marshall Hahn 2010

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

Machinists often simulate a part program to verify its correctness, since mistakes can cause damage to the part, machine, oneself, or others. A popular approach for part program simulation involves representing the stock (the material the part is being carved from) as a heightmap. Although this approach is computationally fast and memory efficient, only objects that are representable as functional surfaces (e.g., $z = f(x, y)$) can be machined. This thesis presents a new heightmap-based data structure, called a multidirectional heightmap, that does not have this limitation. A multidirectional heightmap, in response to an overhang, recursively subdivides itself until each piece can be represented by an axis-aligned heightmap. More precisely, a multidirectional heightmap is a k D-tree with the property that all cells are “functional”: each cell contains a heightmap that represents a functional portion of the stock. To improve accuracy, each regular heightmap can be replaced by a 3-Way Heightmap, a new type of heightmap that samples the tool along all three stock axis directions (three ways) rather than just one. The experimental results herein suggest that the multidirectional heightmap data structure achieves a good level of performance with respect to memory usage, CPU usage, and approximation error.

Acknowledgements

I would like to thank my supervisor Stephen Mann for his support, guidance, and patience, and also for suggesting the intriguing idea of stock representation via multiple heightmaps. I would also like to thank my readers, Craig Kaplan and Sanjeev Bedi, for their helpful comments and suggestions. I would also like to thank the members of the Computer Graphics Lab for helping me hone my public speaking skills, and for all the enchanting coffee hours. I would also like to thank my family for their love and support.

Dedication

Dedicated to my parents.

Contents

List of Tables	viii
List of Figures	xi
1 Introduction	1
1.1 The ToolSim Project	2
1.2 Objectives	4
1.3 Outline	4
2 Background	5
2.1 Milling	5
2.2 ToolSim	6
2.3 Related Work	8
3 Cell Subdivision	10
3.1 Good-Bad Maps	11
3.2 Possible Split Planes for Cell Subdivision	14
3.2.1 Sign Conflict Split Planes	16
3.2.2 Indirect Split Planes	20
3.2.3 Direct Split Planes	23
3.3 Minimization Selection Heuristics	31

4	Stock Rendering with Continuous Seams	36
4.1	Discontinuous Seams	36
4.2	3-Way Heightmaps	38
4.3	3-Map Mesh Construction	40
4.3.1	Exterior Mesh Construction	40
4.3.2	Interior Mesh Construction	42
4.4	3-Maps versus Regular Heightmaps	45
5	Implementation Issues	47
5.1	Data Structure Implementation	47
5.2	Solid Segment Construction	48
5.2.1	Heightmap Solid Segments	49
5.2.2	Tool Solid Segments	51
5.3	Good-bad Map Region Bounding	52
6	Evaluation By Simulation	54
6.1	Simulation Examples	54
6.2	Heuristic Evaluation	59
6.3	CPU and Memory Usage	64
6.4	Maximum Error of Surface Approximation	69
7	Conclusion	72
7.1	Summary and Contributions	72
7.2	Limitations	73
7.3	Other Simulation Techniques	73
7.4	Optimizations	75
	References	76

List of Tables

3.1	The four types of connecting sets.	25
6.1	Simulation parameters	54
6.2	Ruleset total subdivision count.	60
6.3	Ruleset percentage increase/decrease of total subdivision count. . .	60
6.4	Overall ruleset performance.	61
6.5	Run time results.	67
6.6	Memory usage.	68
6.7	Closest error data.	71
6.8	Simulation parameters	71
6.9	Heightmaps versus 3-maps with half the density.	71
6.10	Closest error data.	71

List of Figures

1.1	The process of milling.	1
1.2	Some nonfunctional parts.	2
1.3	A 2D example of simulation with multidirectional heightmaps.	3
2.1	Machine axes.	5
2.2	An example toolpath.	6
2.3	Stamp intersection.	7
2.4	Swept surface intersection	7
2.5	Grazing curve swept surface construction.	8
2.6	Discontinuous swept surface.	8
3.1	The CUTCELL algorithm.	11
3.2	Good bad-map pixel types.	12
3.3	Usage of good-bad maps	13
3.4	Splitting with a Quadtree versus a k D-tree.	14
3.5	The unresolved/resolved curve classification.	15
3.6	Sign conflict example	17
3.7	Complete isolation versus partial isolation	18
3.8	An example of multiple conflicting good regions.	18
3.9	Calculation of partial/complete split planes.	19
3.10	A 2D example of indirect split planes.	20
3.11	A 3D indirect split example.	21
3.12	A more complex 3D indirect split plane example.	22

3.13	Two DSP series examples.	23
3.14	Segments sets and overlap intervals.	24
3.15	The COMPUTEOVERLAPINTERVALS algorithm.	26
3.16	The COMPUTESETS procedure.	26
3.17	The COMPUTEOVERLAPINTERVAL procedure.	27
3.18	calculation of DSP series	28
3.19	The EXPANDSETS procedure.	29
3.20	The RELAXSETS procedure.	29
3.21	A 3D direct splits example	30
3.22	A 2D illustration of the Closest-Toolcut-Direction (CTD) rule.	32
3.23	3D illustration of Closest-Toolcut-Direction rule.	33
3.24	The Furthest-From-Side (FFS) rule.	34
4.1	Discontinuous seam example.	37
4.2	Embedded heightmap illustrations.	39
4.3	The partial side square cases.	41
4.4	The partial non-side square cases.	42
4.5	The partial cube cases.	43
4.6	Heightmap and 3-map wireframe representations of a vertical wall.	45
4.7	A comparison of 3-maps and regular heightmaps.	46
5.1	Multidirectional heightmap memory layout.	48
5.2	Solid segments and usage.	50
5.3	Calculation of bounding rectangles for pixel regions.	53
6.1	Cylinder Spiral (CS).	55
6.2	Donut With Overhang 1 (D1).	56
6.3	Donut With Overhang 2 (D2).	56
6.4	Donut Spiral (DS1).	56
6.5	Donut Spiral (DS2).	56

6.6	Top face drilling and angled edge cut (MA1). The top face cuts connect with the other cut.	57
6.7	Top face drilling and angled edge cut (MA2).	57
6.8	NonAA 45 Degree Drill (NA).	58
6.9	Angled drilling aligned with plane (A1).	58
6.10	Drill To Center of Cube Along Each Axis (A2).	58
6.11	Two planar toolcuts (A3).	58
6.12	A case where the CTD rule fails.	62
6.13	Closest Error Calculation	69

Chapter 1

Introduction

A computer numerically controlled (CNC) milling machine carves a piece of material, such as wood or metal, into a particular object, or part, using a fast spinning computer-controlled tool. For instance, Figure 1.1 illustrates the milling of a simple sign. The user specifies the path that the tool is to follow via a *toolpath*. The machine will execute the toolpath exactly as specified, even if the given movements will cause damage to the part, machine, or people. Although some support for automatic toolpath generation exists, it is still not possible to avoid manual part programming completely; people make mistakes [2]. Machine simulation is the safest and most cost effective way to test multi-axis toolpaths [20].



Figure 1.1: The process of milling.

1.1 The ToolSim Project

Israeli developed a CNC machining simulator called ToolSim [10]. ToolSim uses a heightmap to represent the stock. Although a stock represented in this manner can be updated and rendered easily and efficiently, only objects that are representable as functional surfaces (e.g., $z = f(x, y)$) can be machined. However, not all machinable objects meet this requirement; examples are given in Figure 1.2. The main goal of my thesis is to develop a new heightmap-based data structure, called a *multidirectional heightmap (mdh-map)*, that does not have this limitation. This data structure, in response to an overhang, recursively subdivides itself until each piece can be represented by an axis-aligned heightmap.

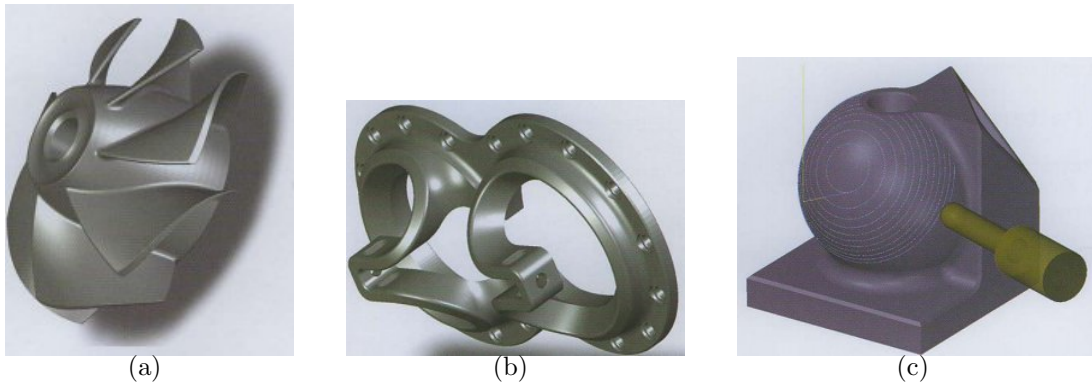


Figure 1.2: Some parts that cannot be represented using a heightmap [2].

More precisely, a multidirectional heightmap is a k D-tree with the property that all cells are “functional”: each cell contains a heightmap that represents a functional portion of the stock. Furthermore, a tool is said to “cut” a cell if it occupies some portion of the cell’s solid space. To maintain the functional cell property, the following simple rule is applied recursively to each cut cell: if the surface within a cell is nonfunctional, subdivide the surface (and cell) into two pieces with an axis-aligned split plane. To improve accuracy, each cell can be represented by a *3-Way Heightmap (3-map)*, a new type of heightmap that samples the tool along all three stock axis directions (three ways) rather than just one. Since a 3-map samples along two directions within each boundary face of its cell, discontinuous seams will not occur if 3-maps are used in place of regular heightmaps.

As an illustration, a 2D simulation example is given in Figure 1.3 (although machining is a 3D problem, 2D examples are easier to understand). This example uses white arrows to indicate the direction of each cell’s heightmap, and it has the following four steps:

- (a) Initially, the multidirectional heightmap consists of one uncut cell C .
- (b) Cell C is cut by the tool, and is represented with a heightmap in the direction of $-\vec{x}$.
- (c) Cell C becomes nonfunctional after a second cut. Thus, the resulting curve is cut into three functional pieces coloured red, green and blue. This partitioning is accomplished in the following recursive fashion. Cell C is first subdivided by the split plane (black line) labelled A . The upper child of this split contains the green curve, and consequently, requires no further subdivision; however, the lower child is nonfunctional because it contains the nonfunctional red and blue curve. As a result, the lower child is subdivided by split plane B , which isolates the red curve portion from the blue curve portion. The recursive process now terminates since all cells are functional: the green, red, and blue curves are represented using heightmaps with directions $-\vec{x}$, \vec{z} and $-\vec{x}$, respectively.
- (d) A cut is made into the top cell; a cyan curve that is functional with respect to direction \vec{z} is the result. However, the top cell is already representing the green curve with a heightmap in the direction of $-\vec{x}$. As a result, the green and cyan curves are separated from each other by split plane C . No further subdivision is required since the children of this split are functional.

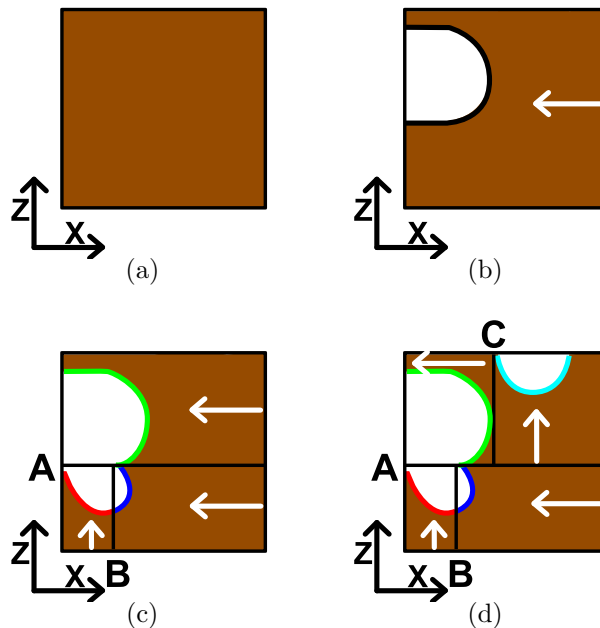


Figure 1.3: A multidirectional heightmap demonstration.

1.2 Objectives

The goal of this thesis is to describe the multidirectional heightmap data structure in detail, and show that it has good performance with respect to memory usage, CPU usage, and approximation error. I describe in detail when cell subdivision is necessary, and where split planes should be placed to reduce the number of subdivision operations necessary. I note why discontinuous seams can occur, and show how this problem can be prevented using another new data structure called a 3-Way Heightmap (3-map). Finally, beneficial aspects of 3-maps as compared to regular heightmaps are noted.

1.3 Outline

Chapter 2 describes some basic CNC machining concepts, relevant aspects of Tool-Sim's internals, and related work. Chapter 3 explains when and where a cell should be subdivided. Chapter 4 describes how to render a multidirectional heightmap with continuous seams. Chapter 5 covers additional implementation details not covered in chapters 3 and 4. Chapter 6 experimentally evaluates the performance of multidirectional heightmaps. Chapter 7 summarizes the important results in this thesis, notes the limitations of those results, and outlines future work that could be done.

Chapter 2

Background

This chapter covers some basic CNC machining concepts and terminology (Section 2.1), how ToolSim simulates the milling process (Section 2.2), and related research (Section 2.3).

2.1 Milling

A *CNC milling machine* carves a solid block of material, called the *stock*, into an object. The tool can be moved along a number of *axes*. A three axis machine can translate the tool along the stock's \vec{x} , \vec{y} , and \vec{z} axes (see Figure 2.1(a)). A five axis machine is similar but has two additional rotational axes. There is no standard physical implementation for the rotational axes; one possibility is given in Figure 2.1(b). A five axis machine can mill nonfunctional objects, but a three axis machine cannot.

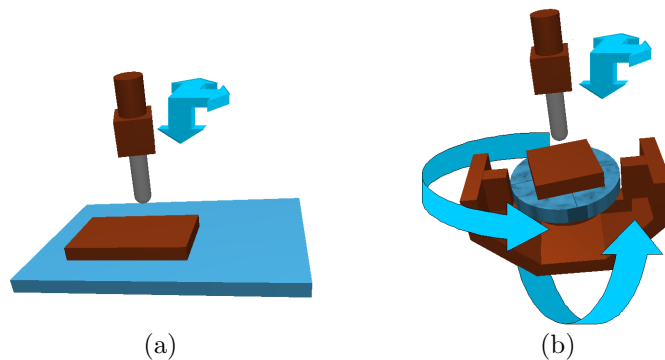


Figure 2.1: A three axis machine (part (a)) and a five axis machine (part (b)) [10].

A *toolpath* is a list of commands called g-codes that are executed by the machine in the order given. The most important g-code specifies a *machine coordinate*, a tuple of numbers that encode a particular tool location, and possibly orientation, relative to the stock (the orientation part is omitted if no rotational axes are available). The machine linearly interpolates between each successive machine coordinate pair (see Figure 2.2) to produce the intended part, which is referred to as the *design surface*. There are many other g-codes [20], but most of them specify actions that ToolSim does not simulate. For instance, ToolSim does not simulate the command used to control the tool’s speed of rotation.

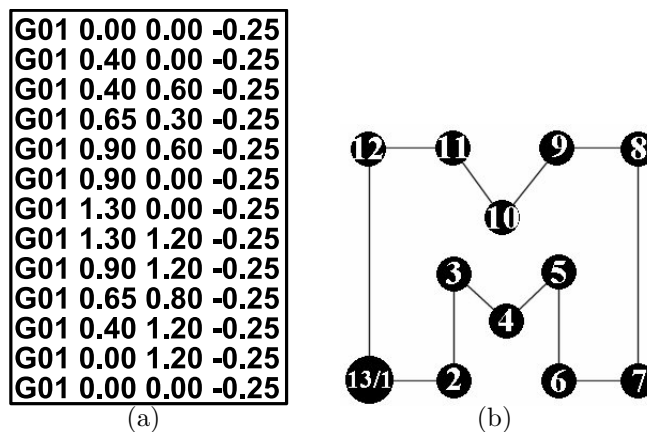


Figure 2.2: A toolpath (part (a)) and the path the tool will trace when it is executed (part (b)).

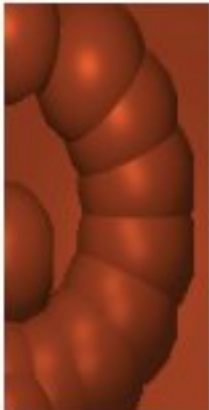
2.2 ToolSim

In this section, I cover the aspects of ToolSim’s internals that are essential to understand my thesis. ToolSim simulates a toolpath without taking into account the forces involved; in concept, the volume swept by the tool is calculated, and this volume is subtracted from the stock representation. Since the tool spins significantly faster than its speed in relation to the stock, it can be represented using simple geometric primitives. Specifically, the tool’s base is represented using a cylinder and the tip is represented using a sphere, cylinder, cone or torus (in this work, I mostly rely on a spherical tip but sometimes use a cylindrical tip). Furthermore, ToolSim assumes that the input toolpath specifies a functional design surface. Therefore, the stock can be represented as heightmap.

The machine’s continuous motion is approximated with discrete time steps.

That is, linear interpolation over time is used to generate the tool’s position in between each machine coordinate. The state of the simulation at each of these steps is referred to as an *in-between frame* or *in-between step*. The tool’s intersection with the stock (a heightmap) is computed for each in-between step, a process called *stamping*. More precisely, stamping involves intersecting a number of rays with the geometric primitives representing the tool; an axis-aligned bounding box is used to prune the number of ray intersection computations necessary.

The smaller the step size (the number of in-between steps), the more accurate the representation of the design surface. In other words, a more accurate simulation is obtained when the degree of overlap between the current stamp and previous stamp is high (see Figure 2.3). Clearly the stamping approach can be computational wasteful. Therefore, ToolSim approximates the tool’s cutting path with a swept surface, an approach first developed for CNC machining by Blackmore et al. [3]. As can be see via a comparison of figures 2.3 and 2.4, greater accuracy can be achieved with a larger step size when swept surfaces are used instead of stamping.



(a) Large step size.



(b) Small step size.



(a) Large step size.



(b) Small step size.

Figure 2.3: Stamp intersection [10].

Figure 2.4: Swept surface intersection [10].

ToolSim approximates the tool’s swept surface as a piecewise polygonal surface, an approach developed by Roth et al. [4] and Mann and Bedi [13]. To see how it is constructed, note that the tool is always in contact with the swept surface along a curve. This curve, referred to as a *grazing curve*, is approximated using a fixed number of points interconnected by line segments. The grazing curve for the current in-between frame is connected by line segments to the grazing curve of the previous in-between frame to obtain a triangular strip (see Figure 2.5), and this triangular strip is used to update the stock for the current in-between frame. Stamping is still sometimes used to deal with discontinuous tool direction changes (see Figure 2.6).

It should be noted that Figures 2.5 and 2.6 illustrate swept surfaces generated from the tool tip (a sphere) only. A ToolSim user can specify that the base of the tool should be intersected with the stock as well, and if this feature is enabled, it will be necessary to generate a swept surface for the tool base (a cylinder) as well.

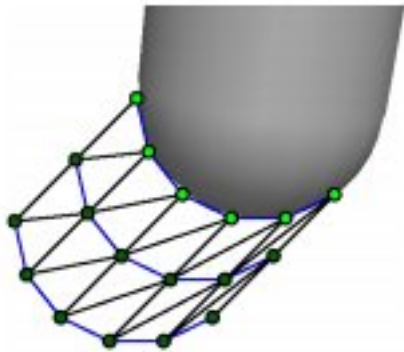


Figure 2.5: A swept surface created from grazing curve points [10].

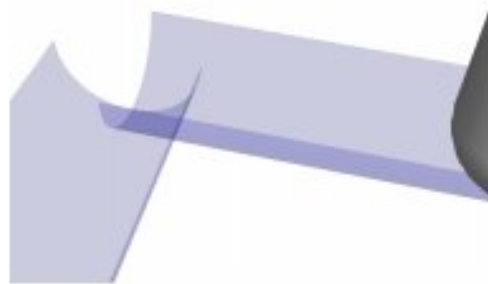


Figure 2.6: Swept surfaces created from a discontinuous toolpath [10].

ToolSim can render each in-between frame to create an animation of the milling process; alternatively, the expense of rendering can be forgone if only the final result is desired. The heightmap data is interpreted as a triangular mesh, and a normal vector is stored with each height sample so that smooth shading via OpenGL is possible [5]. To avoid the CPU-GPU data exchange bottleneck, the heightmap data is uniformly partitioned into cells, and the content of each cell is represented by a display list.

ToolSim has three parameters that allow the user to trade off simulation speed versus accuracy. The *stock density* specifies the number of height samples per unit area by which to represent the design surface. The *step size* indicates how many in-between steps should be used in between consecutive machine coordinates. Finally, the *grazing curve density* is the number points used to construct each grazing curve.

2.3 Related Work

In this section, I briefly mention some previous research involving the representation of solid objects with multiple heightmaps [17, 18]; simulation techniques utilizing other stock representations will be covered in Section 7.2. Santos et al. [18] represent solid objects using “solid heightmap sets”, layers of heightmaps having parallel directions. Odd layers add geometric information while even layers subtract. In

contrast, the approach of Ochotta et al. [17] places no restrictions on the heightmap directions. Whereas I use heightmaps for their usefulness as a dynamic data structure, the other approaches use them to reduce the memory requirements of a static object.

Chapter 3

Cell Subdivision

In Figure 1.3, a simple 2D example was used to illustrate the dynamic nature of a multidirectional heightmap. That is, the following simple rule recursively processes any cut cell: if the surface within a cell is nonfunctional, subdivide the surface (and cell) into two pieces with an axis-aligned split plane. During Step (b) for instance, subdivision was deemed necessary after the tool cut the multidirectional heightmap a second time. The cell was then subdivided using an appropriately-placed split plane. Next, the same logic was applied to each child cell, and as a result, the lower child was also subdivided.

The purpose of this chapter is to explain when and where a cell should be subdivided, but first, some concepts and terminology must be defined. A cell is a bounded region of space that occupies some portion of a solid object and is classified as either completely solid (*solid*), partially solid (*partial*), or completely empty (*empty*). In this work, a cell is either a box (*box cell*), rectangle (*rectangle cell*), or a line segment (*segment cell*). A partial cell is divided into one solid portion and one empty portion by a separating entity, a triangular mesh surface for a box (*separating surface*), a piecewise linear curve for a rectangle (*separating curve*), and finally, a point for a line segment (*separating point*).

A separating curve/surface may be disjoint. For instance, a disjoint separating curve occurs in Figure 3.2. In this figure, the cell outline is the black box, and the separating curve (black) divides the empty portion (white) from the solid portion (brown). The same conventions are used to illustrate all other cells shown in this work.

A heightmap is a collection of heights, calculated at grid locations relative to an axis-aligned plane, that represent a functional surface. The separating surface

of a functional cell is represented using a heightmap. The direction of such a heightmap is referred to as the *direction of representation* of the heightmap/cell, and in each example, this direction is indicated with a white arrow. Whenever the term “heightmap” is used, it refers to a heightmap that composes a portion of the stock. Collectively, the heightmaps a multidirectional heightmap consists of serve as a surface representation for the solid being machined.

The term *stock* refers to a multidirectional heightmap. The notation $F(p, \vec{v}_1, \vec{v}_2, \vec{v}_3)$ denotes a frame named F specified by point p and vectors \vec{v}_1 , \vec{v}_2 and \vec{v}_3 . The stock has a frame $S(P, \vec{x}, \vec{y}, \vec{z})$, where P is a corner of the root node’s cell. It often appears near the lower left corner of the figures; see Figure 3.2 for instance.

The pseudocode of Figure 3.1 overviews how a surface representation is found for a cell that has become nonfunctional. The algorithm uses a set of three good-bad maps to determine if the cell must be subdivided, as described in Section 3.1. If cell subdivision is necessary, possibly several (sign conflict, indirect and direct) split planes are calculated using the algorithms described in Section 3.2. Finally, one of these split planes is selected for application using the heuristics described in Section 3.3.

```

CUTCELL(const sweptTool& tool, const Cell& cell)
{
  Generate three good-bad maps  $G_X$ ,  $G_Y$ , and  $G_Z$ , one for each stock axis.

  if  $G_X$ ,  $G_Y$ , or  $G_Z$  corresponds to a valid direction of representation  $\vec{d}$ 
  then
  {
    Resample  $cell$ 's separating surface  $S$  in the direction of  $\vec{d}$ .
    Subtract  $tool$  from  $S$ .
  }

  else
  {
    Produce a list  $L_P$  of direct, indirect and sign conflict splits.
    Select from  $L_P$  a single split plane  $P$  using a combination of heuristics.
    Split the  $cell$  with plane  $P$  into  $leftChildCell$  and  $rightChildCell$ .
    CutCell( $tool$ ,  $leftChildCell$ )
    CutCell( $tool$ ,  $rightChildCell$ )
  }
}

```

Figure 3.1: The CUTCELL algorithm.

3.1 Good-Bad Maps

A \vec{v} -good-bad map is a 2D image that indicates which portions of a cell’s separating surface can be represented by a heightmap with direction $\pm\vec{v}$ (good), and which portions cannot (bad). Each pixel indicates whether or not the separating surface

is functional at a particular grid location. A bad pixel indicates it is not, and a good pixel indicates that it is. Each type of pixel is illustrated in Figure 3.2, which shows a 1D \vec{x} -good-bad map generated from a disjoint separating curve. Rays that are parallel to \vec{x} , the *direction of computation*, are cast from each grid location. The number of times a ray intersects the separating curve (or surface) determines the value of the corresponding pixel. A pixel is bad when there is more than one intersection, and good otherwise.

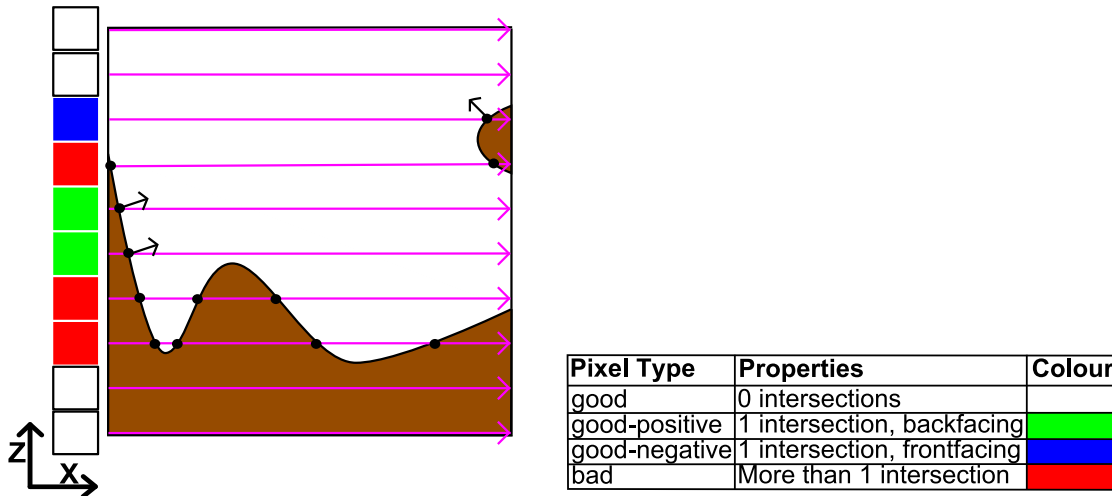


Figure 3.2: A 1D good-bad map. To the right is a table summarizing the pixel types. To the left is a 1D \vec{x} -good-bad map generated from a disjoint separating curve.

Although a single intersection point indicates that the surface is functional, a heightmap surface is backfacing relative to its direction of representation. Therefore, it is necessary to distinguish between intersection points that are backfacing relative to the direction of computation, and those that are frontfacing. The former case indicates a good-positive pixel and the latter indicates a good-negative pixel. Good (unsigned) pixels can also occur if there is no intersection point.

When a good-bad map contains both good-positive and good-negative pixels, the pixels are said to have “conflicting” signs, and a situation called a *sign conflict* exists. Hence, a good-bad map indicates a “valid” direction of representation exists for a cell’s separating surface if and only if **(1)** there are no bad pixels, and **(2)** there are no sign conflicts. Furthermore, a cell is functional if and only if a valid direction of representation exists.

If a cell becomes nonfunctional after a toolcut, a recursive subdivision process then restores the functional cell property (the CUTCELL algorithm, Figure 3.1). As a cell is functional only if a valid direction of representation exists, three good-bad

maps, one in the direction of each stock frame axis, are used to decide if the recursive case should be applied. As an illustration, Figure 3.3(b) displays a particular set of good-bad maps generated from the cell being cut in Figure 3.3(a). None of these good-bad maps correspond to a valid direction of representation, and therefore, cell subdivision is necessary.

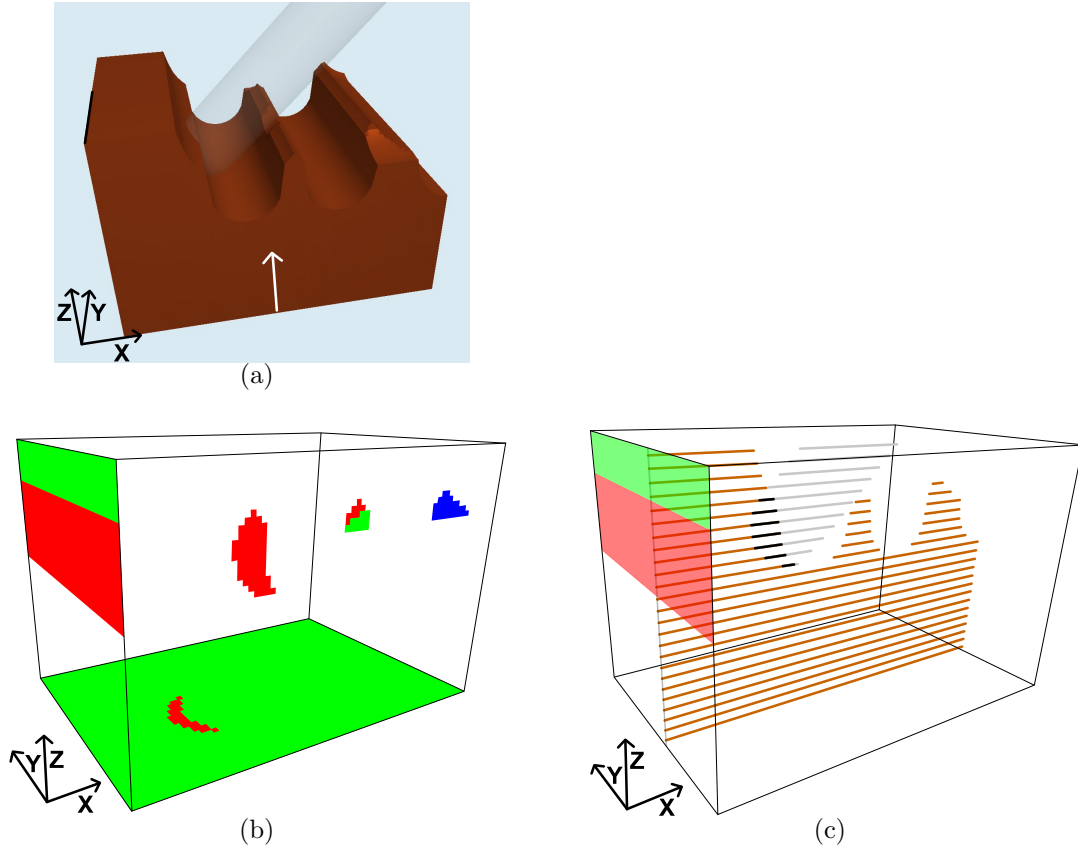


Figure 3.3: 2D good-bad map generation. **(a)** Angled drilling into a heightmap surface. **(b)** A set of three good-bad maps, one for each stock axis direction, generated from the boolean subtraction of the tool's swept volume from the cell of part (a). Since each good-bad map contains bad (red) pixels, a heightmap cannot represent the surface that is being machined; recursive cell subdivision is necessary. **(c)** Computation of one column of pixels for the \vec{x} -good-bad map from part (b).

The separating surface from which good-bad maps are generated is created using constructive solid geometry (CSG) [14]: the volume swept by the tool is subtracted from the cell's solid space. For example, Figure 3.3(c) illustrates how CSG is used to compute one column of pixels for the \vec{x} -good-bad map of Figure 3.3(b). Line segments that represent solid objects are produced by intersecting rays with both the cell's solid space and tool's swept volume. The tool line segments (gray) are

subtracted from the cell line segments (brown). Along each ray, the intersection points that remain are counted to determine the value of the associated pixel.

3.2 Possible Split Planes for Cell Subdivision

A cell must be subdivided if it becomes nonfunctional. While it would be simplest to always divide a cell into eight equally-sized children, as is done with an Octree, minimization of the total number of subdivisions is desirable for performance reasons. Both memory consumption and traversal time increase as the tree grows. Moreover, cell subdivision operations can be expensive since every height must be examined. It is more appropriate to organize the cells as a k D-tree, since more “intelligent” split planes can be computed at a reasonable cost. Figure 3.4 illustrates that such split planes exist. It compares the k D-tree and Quadtree approaches when applied to the example of Figure 1.3. Clearly the k D-tree approach requires fewer subdivisions. If the stock were even larger relative to the size of the cuts, the quad-tree approach would require even more splits, while the k D-tree approach would not.

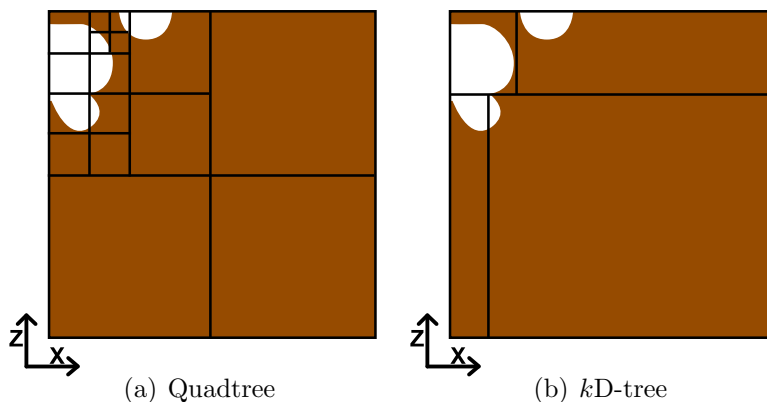


Figure 3.4: A comparison of the k D-tree approach and the Quadtree approach when applied to the example of Figure 1.3. The k D-tree approach requires fewer splits.

Algorithms for intelligent split plane computation are described in this section, and they have a few common features. For one, they all exploit the following concept: whenever a cell becomes nonfunctional, usually only part of the cell’s new separating surface cannot be represented by the current heightmap of the cell. The unrepresentable curve/surface regions together compose an *unresolved curve/surface* and all other curve/surface regions compose a *resolved curve/surface*. Since the

resolved surface already has a representation, the unresolved surface should be the focus when calculating split planes.

Figure 3.5(a), which shows a cell that has been cut by the tool (gray), illustrates the unresolved/resolved curve classification. With respect to the \vec{z} axis, the current direction of representation, the red curve is nonfunctional and thus unresolved, while the green curve is resolved. A second example is given in Figure 3.5(b). Recall that a heightmap curve/surface must be backfacing relative to its direction of representation, which is \vec{x} in this case. After the toolcut, this condition still holds for the green curve, which is therefore resolved; but it does not hold for blue curve, which is consequently unresolved.

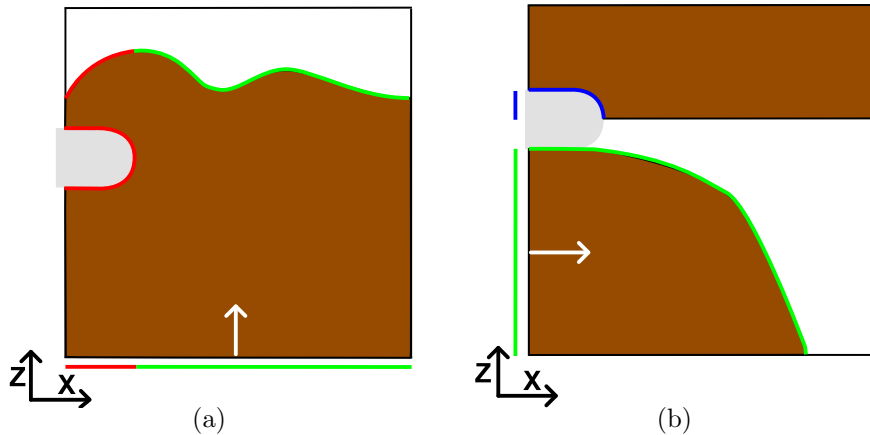


Figure 3.5: The unresolved/resolved curve classification. The red curve of part (a) is unresolved because it is nonfunctional with respect to the heightmap’s direction \vec{z} ; the blue curve of part (b) is unresolved because it is frontfacing relative to the heightmap’s direction \vec{x} .

The other feature split plane computation algorithms have in common is that they all utilize information provided by good-bad maps. We saw in Section 3.1 how good-bad maps are used to determine the functionality of a cell. When first partitioned into regions of “related” pixels, they are also useful for split plane computation purposes. There are two ways the pixels are partitioned. First, the pixels are partitioned based upon the unresolved/resolved classification of the separating curve/surface: rays that intersect the unresolved surface generate *unresolved regions*, and all other rays generate *resolved regions*. Next, the pixels are partitioned based upon pixel type (bad, good-positive, etc.).

I have derived three split plane calculation algorithms based on the above concepts, and each one computes exactly one of the following split plane types:

- Sign conflict split planes, described in Section 3.2.1, help eliminate a sign conflict.
- Direct split planes, described in Section 3.2.3, help *directly eliminate* an unresolved bad region. That is, they help cut the unresolved surface into functional pieces relative to an axis \vec{a} , and thereby, help directly eliminate an unresolved bad region with direction \vec{a} .
- Indirect split planes, described in Section 3.2.2, help *indirectly “eliminate”* an unresolved bad region with direction \vec{a} . Specifically, an *indirect split (plane)* isolates an unresolved good region with direction \vec{b} from surrounding bad regions, thereby allowing the unresolved curve/surface to be represented using the alternative direction of representation \vec{b} .

Direct splits are calculated *directly* from 3D intersection information, but only a *subset of rays* are considered (i.e., rays that generate unresolved pixels). In contrast, indirect and sign conflict splits are calculated from 2D good-bad maps, but 3D intersection information from *all rays* is considered *indirectly*; furthermore, all resolved pixels can be computed using a series of computationally cheap projection operations rather than with ray intersection calculations (see Section 5.2.1). This optimization is one reason why it would be costly to calculate all split planes directly from 3D intersection information; however, at least some direct consideration of 3D intersection information is necessary since, in some cases, all unresolved pixels will be bad. As all split planes are axis-aligned (e.g., $x = 4$), they are all ultimately calculated from the projections of 3D intersection information onto stock frame axes. All three split plane types will be discussed in turn.

3.2.1 Sign Conflict Split Planes

Recall that if a \vec{v} -good-bad map contains a sign conflict (contains both good-positive and good-negative pixels), the cell’s separating surface cannot be represented using a heightmap with direction \vec{v} . The goal of a sign conflict split is to eliminate, or resolve, a sign conflict. Specifically, a *sign conflict split (plane)* helps isolate an unresolved good region from other regions that conflict (more on the unresolved part later). As an illustration, Figure 3.6(a) is a cube with two removed corners. After the second corner (the lower of the two) was removed by the tool (see Figure 3.6(b)), a good-bad map was generated for each direction. Since there was a sign conflict in every direction, cell subdivision was necessary. Any one of sign conflict split

planes A , B , or C (shown as black outlines) could have restored the functional cell property.

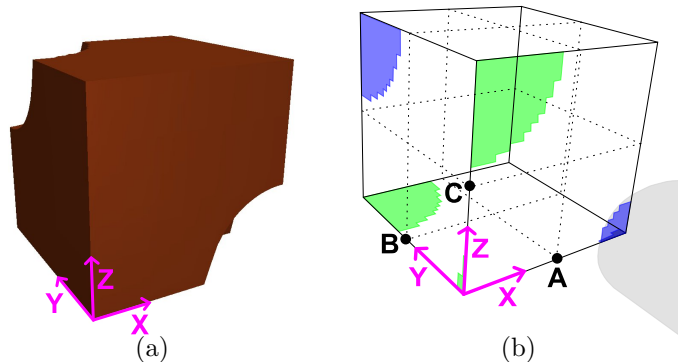


Figure 3.6: A sign conflict split plane example. **(a)** A cube with two removed corners. **(b)** The three good-bad maps generated after the lower corner was removed by the tool. Any one of splits A , B , or C could have restored the functional cell property.

As Figure 3.7 illustrates, a split plane can be classified as either complete or partial. A “complete” split plane completely isolates an unresolved good region from conflicting regions to its left or right. For instance, split planes A , B and C of Figure 3.7(a) are complete split planes, since they each completely isolate the unresolved good region (blue) from a conflicting good region (green). But note that more splitting is necessary after either split B or C whereas no further splitting is required after split A .

On the other hand, a “partial” split plane partially isolates an unresolved good region from conflicting regions to its left or right. For example, splits A and B in Figure 3.7(b) partially isolate the unresolved good region (blue) from a conflicting region (green); the region is fully isolated after complete split C . In general, further splitting is usually necessary after application of a partial split.

Only unresolved regions must be isolated to achieve resolution; isolation of conflicting resolved regions from each other is not necessary. For example, Figure 3.8(a) shows a cell with direction of representation \vec{z} that became nonfunctional after the tool (gray) drilled into its side. The \vec{y} -good-bad map shown in Figure 3.8(b) was generated from this cell. This good-bad map contains an unresolved good-negative region U , two resolved good-positive regions that conflict with U , and three resolved good-negative regions. Five split planes A , B_1 , C_1 , B_2 and C_2 could have been calculated from this good-bad map. Each of splits B_2 and C_2 isolates a pair of conflicting *resolved* regions from each other, while split A isolates an *unresolved* region U from the two conflicting regions above (ignoring the other two splits for

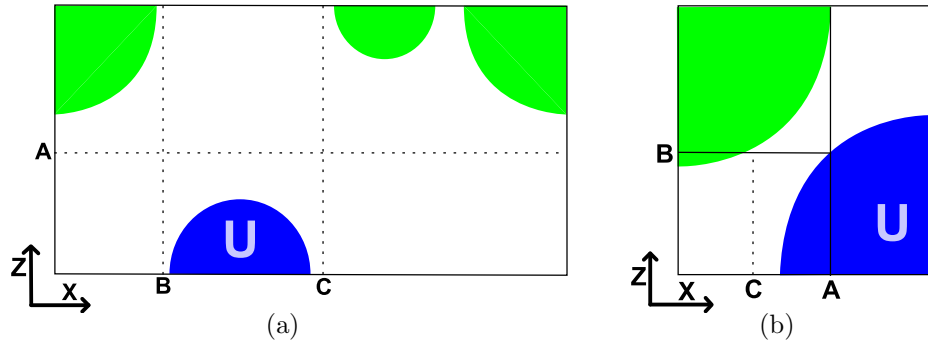


Figure 3.7: Complete versus partial split planes. (a) Three complete splits A , B , and C . (b) Two partial splits A and B . Split C , the final split, is a complete split plane.

now). After application of split A , resolution would be achieved. The top child would use direction of representation \vec{z} (the parent's direction of representation), and the bottom child would use direction $-\vec{y}$. If any combination of splits B_2 and C_2 were used instead, split A would still be required; these splits are useless.

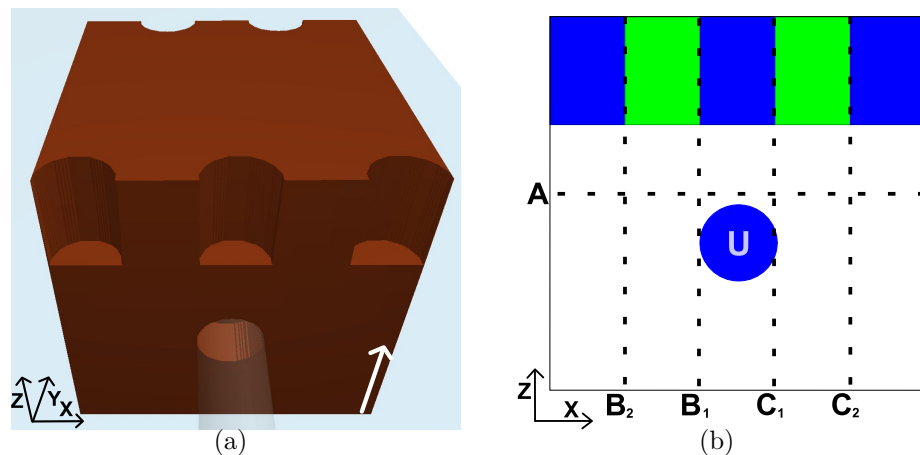


Figure 3.8: Unresolved good region isolation. (a) A cell with direction of representation \vec{z} becomes nonfunctional after it is cut by the tool (transparent gray). (b) The \vec{y} -good-bad map. Resolution can only be achieved if the unresolved good-negative region U is isolated from the resolved conflicting regions above. Only split A , or the pair of splits B_1 and B_2 , could have accomplished this goal.

Together, split planes B_1 and C_1 could also achieve resolution. Split B_1 isolates the unresolved region (blue) from the conflicting resolved region (green) to the left, and split C_1 isolates the unresolved region (blue) from the conflicting resolved region (green) to the right. However, since minimization of the total number of split planes is desirable, resolution would be best achieved via split A (ignoring

other high level criteria discussed in Section 3.3).

The following algorithm is used to compute from a good-bad map the sign conflict split planes that intersect an axis \vec{a} (Figure 3.9 provides a visualization):

1. Compute the orthogonal projection of each region onto axis \vec{a} , with segment U (blue) being the orthogonal projection of an unresolved good region, segment C_L being the closest “conflicting” segment to the left of segment U , and segment C_R being the closest conflicting segment to the right of segment U .
2. Compare the resulting line segments as follows:
 - A. If segment U and segment C_L (or C_R) overlap, each overlapped endpoint (black) corresponds to a partial split plane. For instance, two partial split planes (black dashed lines) occur along the \vec{z} axis in Figure 3.9.
 - B. If segment U and segment C_L (or C_R) do not overlap, a *separating interval* exists between them. Any point within the range of a separating interval corresponds to a complete split plane. For instance, along the \vec{x} axis in Figure 3.9, a separating interval (black) occurs on each side of segment U , and the range of each one is shown as a dark box.

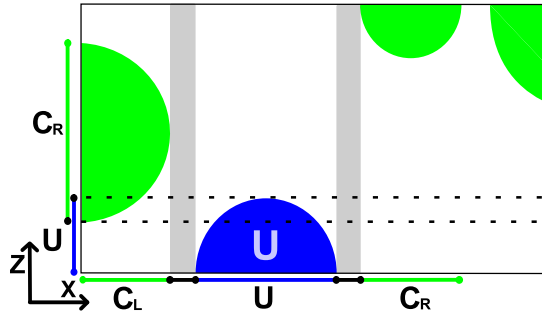


Figure 3.9: Split calculation via orthogonal projection of regions onto stock frame axes.

Note that a good-bad map may contain a sign conflict as well as various bad regions. If this is the case, either indirect splits (Section 3.2.2) or direct splits (Section 3.2.3) are applied first to eliminate the bad regions. If any of the resulting child cells contain sign conflicts, only then are sign conflict splits computed. While possibly non-optimal, this algorithm is easy to implement.

3.2.2 Indirect Split Planes

The indirect split plane concept is demonstrated using a simple 2D example (Figure 3.10(a)). The tool (gray) has cut into the side of a cell with direction of representation \vec{z} . The red portion of the resulting separating curve cannot be represented using the current heightmap, and is thus unresolved. A \vec{v} -good-bad map G was then generated to check if the unresolved curve is representable using a heightmap with an alternative direction of representation \vec{v} , \vec{x} in this case (shown to the left of the cell as red and blue line segments). Consider the unresolved region belonging to good-bad map G . If it is at least partially composed of good pixels, at least part of the unresolved surface can be represented using direction \vec{v} . An *indirect split (plane)* isolates an unresolved good region from surrounding bad regions, thereby allowing the unresolved curve/surface to be represented using an alternative direction of representation. In the case of Figure 3.10(a), the blue unresolved region is good, and the red unresolved region is bad. Any split within the range of the dark box divides in between these regions, and is therefore is an indirect split. Figure 3.11 shows a 3D example similar to the 2D example of Figure 3.10(a).

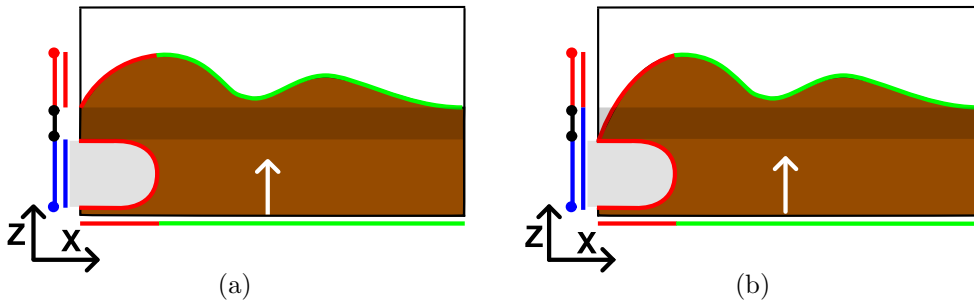


Figure 3.10: A 2D example of indirect split planes. **(a)** The tool (gray) has cut a cell with direction of representation \vec{z} . The lower component of the unresolved curve U (red) is representable in the direction of $-\vec{x}$ because the corresponding unresolved region is good-negative (blue); the upper component is not since the corresponding unresolved region is bad (red). A separating interval (black) exists between the unresolved good region and the unresolved bad region. **(b)** This example is nearly identical to the part (a) example, but differs because the unresolved surface is not disjoint.

Indirect split planes are computed from a good-bad map using a technique similar to the method used for sign conflict split planes. The key difference is the following: each split plane computed should isolate an unresolved good region from surrounding bad regions. However, there are two reasons why it is not necessary to isolate the entire unresolved good region. First of all, the part of the unresolved

good region generated from only the heightmap (no ray passed through the tool) can be ignored. The reason why is that any child cell that does not contain the tool can be represented with the same direction used by the parent cell. Consider Figure 3.10(b) for instance. After a split within the range of the dark box, the top child contains part of the unresolved surface, but does not contain the tool. Thus, the top child can be represented with direction \vec{z} , the parent cell's direction of representation. Notice that the separating interval (black) overlaps the part of the unresolved good region (blue) generated from non-tool-intersecting rays.

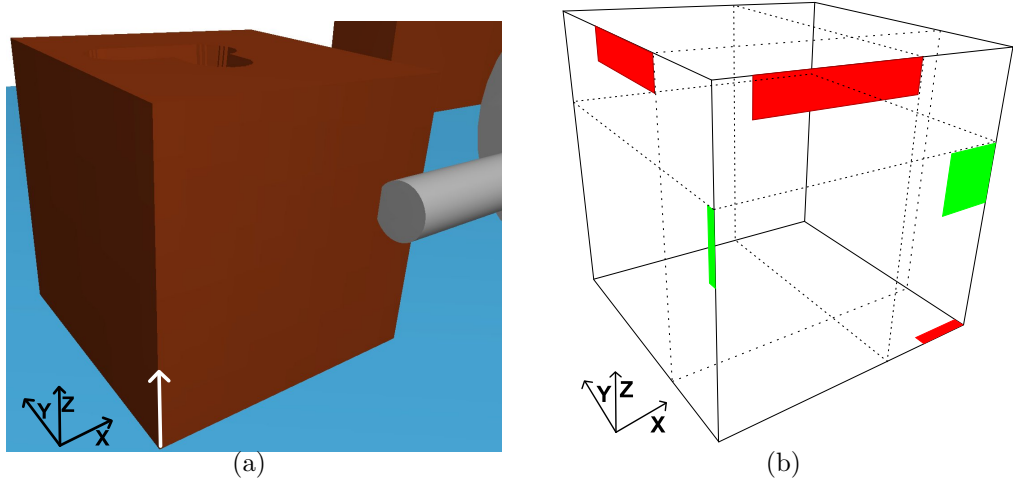


Figure 3.11: A 3D indirect split example. **(a)** A cell with direction of representation \vec{z} is cut a second time (previously the top face was cut). **(b)** The three good-bad maps generated in response to this event. The \vec{z} -good-bad map indicates that an unresolved surface is present. Fortunately, both the \vec{x} -good-bad map and \vec{y} -good-bad map possess an unresolved region composed of only good-positive pixels (green). Therefore, the unresolved surface can be represented with either direction \vec{x} or \vec{y} . However, both good-bad maps also contain a bad region (red). Three indirect split planes (dashed outlines) could eliminate this problem, and each one isolates an unresolved good region from a bad region.

Figure 3.12 illustrates the second reason isolation of the entire unresolved good region is unnecessary. Recall that the unresolved surface cannot be represented with the direction of the current heightmap. Of the other two directions, one direction may be able to represent the part of the unresolved surface that the other direction cannot. As a result, the unresolved bad region from one map can be used to restrict the good region that must be isolated in the other.

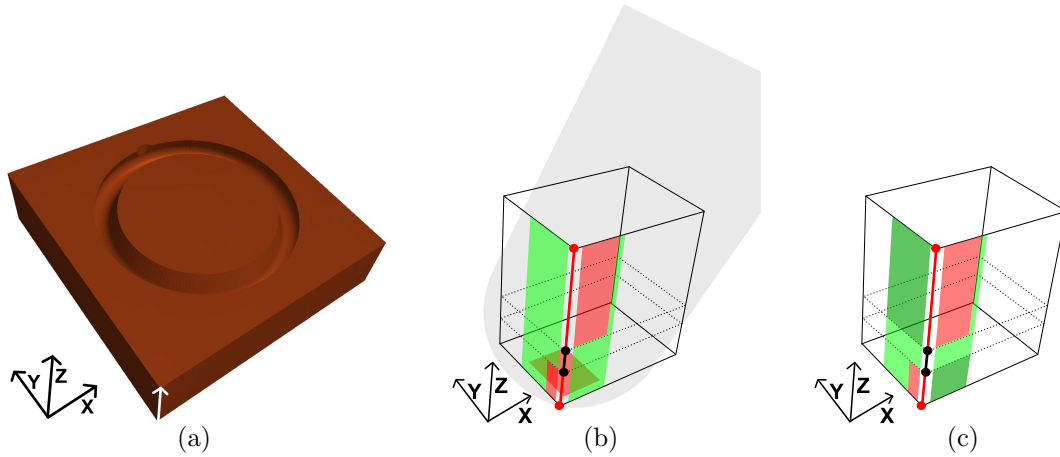


Figure 3.12: A more complex 3D indirect split plane example. **(a)** A donut with an overhang along the outer rim was machined by rotating an angled tool. **(b)** A cell that occurred during the machining process. The tool (gray) cut the cell in such a way that there was an unresolved surface relative to the cell's direction of representation \vec{z} , as indicated by the unresolved bad region (red) the corresponding good-bad map contains. The two other good-bad maps each contain an unresolved region that is part good (green) and part bad. One of these regions indicates that part of the upper portion of unresolved surface cannot be represented with direction \vec{y} ; another indicates part of the lower portion cannot be represented with direction \vec{x} . A split plane in between these bad regions (see the black dashed lines) will resolve the situation. **(c)** How the separating interval (black) is calculated. The unresolved bad region in the direction of \vec{x} is used to restrict the unresolved good region in the direction \vec{y} (dark green) that must be isolated.

3.2.3 Direct Split Planes

The split planes discussed so far are calculated from 2D/1D information (good-bad maps). Such split planes are insufficient to deal with all possibilities. For example, Figure 3.13(a) shows a cell with direction of representation \vec{z} that was cut by the tool (gray). Notice that the unresolved regions (red) generated from the unresolved curve (red) consist entirely of bad pixels. To calculate sign conflict splits and/or indirect splits, an unresolved good region must be present. Consequently, another variety of split plane is needed: the *direct split (plane)*.

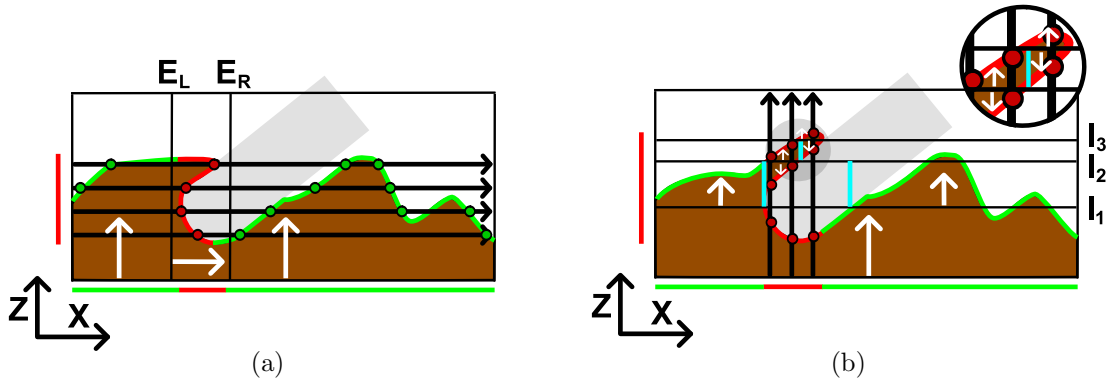


Figure 3.13: Two DSP series examples, each for cells with direction of representation \vec{z} . (a) A DSP series with direction \vec{x} and two exterior splits E_L and E_R . (b) A DSP series with direction \vec{z} and three interior splits I_1 , I_2 and I_3 . Three additional (sign conflict) splits (cyan) were required after its application.

The goal of a *direct split plane (DSP) series* is the direct elimination of an unresolved bad region with direction \vec{a} . With respect to an axis \vec{a} , the *interior (direct) splits* cut the unresolved surface into functional pieces, and each *exterior (direct) split* isolates the unresolved surface from a portion of resolved surface to the left or right. After a DSP series is applied, the leftmost and rightmost child cells will not contain a portion of the unresolved surface and can therefore each use the same direction of representation as the parent cell. In contrast, all other child cells will contain a portion of the unresolved surface functional with respect to axis \vec{a} ; but before the unresolved surface can be represented with a direction parallel to \vec{a} , these cells may require additional (indirect and/or sign conflict) splits.

Continuing with the Figure 3.13(a) example, a DSP series with direction \vec{x} was generated after the toolcut. That is, a ray (black) was cast from each unresolved pixel (red), and two exterior splits E_L and E_R were calculated from the intersection information along these rays. After application of this DSP series, the middle

child cell contained a portion of resolved/unresolved curve (green/red) functional and backfacing with respect to \vec{x} ; thus, this cell used direction of representation \vec{x} . In contrast, the leftmost and rightmost child cells used the same direction of representation as the parent (\vec{z}), because the unresolved curve was absent from their interiors.

A second DSP series example is given in Figure 3.13(b). Again, the tool (gray) has cut a cell with direction of representation \vec{z} . In this case, a DSP series with direction \vec{z} , and three interior splits I_1 , I_2 , and I_3 , was calculated. Notice that after this DSP series was applied, the two middle cells were subdivided by sign conflict splits (cyan). A larger view of the circled sign conflict split plane is shown near the upper right corner.

DSP Series Computation

The DSP series for an axis \vec{a} , the *axis of computation*, is computed from particular sets of line segments (*segment sets*). Rays are cast from each unresolved pixel of the \vec{a} -good-bad map; see the rays (black) generating the unresolved bad region (red) in Figure 3.14(a) for instance. Next, along each ray, successive intersection pairs are connected by line segments. Only three sets of line segments are of interest: a set of *tool* (intersecting) segments (orange), a set of directly to the *left-of-tool* segments (purple) and finally, a set of directly to the *right-of-tool* segments (cyan).

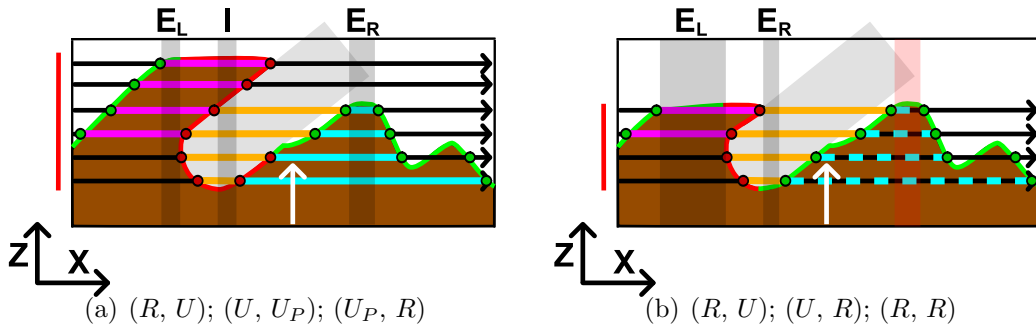


Figure 3.14: Two cells (a) and (b) with direction of representation \vec{z} . After each cell was cut by the tool, a DSP series for direction \vec{x} was computed. Below each cell, the notation from Table 3.1 denotes each connecting set in the order *left-of-tool*, *tool* (intersecting), and *right-of-tool*. Every overlap interval computed is shown as a dark box.

In general, every segment set generated (e.g., those of Figure 3.14) will be a *connecting set*, a set of line segments that “connect” two distinct functional portions of curve/surface S_L and S_R ; that is, all left segment endpoints coincide with S_L

and all right segment endpoints coincide with S_R . Moreover, every segment set shown in Figure 3.14(a) has the following property: all member segments intersect when orthogonally projected onto the axis of computation (\vec{x} in this case); their intersection is referred to as an *overlap (interval)*. Any plane within the range of a “connecting” set’s overlap interval (see the dark boxes labelled E_L , I , and E_R) divides in between a curve/surface to the left (S_L) and a curve/surface to the right (S_R), and is therefore a *direct split*.

In terms of the resolution status of the connected curves/surfaces, four types of connecting sets can occur, and they are summarized in Table 3.1. As an example, the segment sets generated in Figure 3.14(a) are of types 1, 3, and 2 (left to right order). Another example is given in Figure 3.14(b), and it illustrates types 1, 2, and 4. Note that no overlap interval (the red box) was computed from the type 4 connecting set (it lies in between two resolved curve portions).

Type	Definition	Associated Split Type	Composing Segments
1	$(R, U \text{ or } U_P)$	left exterior	left-of-tool xor tool
2	$(U \text{ or } U_P, R)$	right exterior	right-of-tool xor tool
3	$(U \text{ or } U_P, U \text{ or } U_P)$	interior	tool
4	(R, R)	none	left-of-tool xor right-of-tool

Table 3.1: The four types of connecting sets. The notation (RS_L, RS_R) is used to denote a connecting set, with RS_L as the resolution status of the left curve/surface, and RS_R as the resolution status of the right curve/surface. Resolution status will be one of the following: resolved (R), unresolved (U), or partially unresolved and partially resolved (U_P).

In general, the members of each respective segment set generated (i.e., the *left-of-tool*, *tool*, and *right-of-tool* sets) may not overlap; thus, as is necessary, Algorithm COMPUTEOVERLAPINTERVALS (Figure 3.15) partitions each segment set (input as three lists L_L , L_T , and L_R) into smaller sets whose respective members do overlap (*overlap sets*), and outputs the corresponding overlap interval of each one (as a list L_O). From the output list L_O a DSP series is calculated. That is, one direct split plane within the range of each overlap interval is computed (see Section 3.3). Algorithm COMPUTEOVERLAPINTERVALS has four main steps, and these steps are implemented using procedures COMPUTESETS, COMPUTEOVERLAPINTERVAL, EXPANDSETS, and RELAXSETS. All steps will be discussed in turn.

Steps 1 and 3 of algorithm COMPUTEOVERLAPINTERVALS are implemented with procedure COMPUTESETS (Figure 3.16). This procedure assigns each segment of a connecting set (a list L_I initialized as either *tool*, *left-of-tool* or *right-of-tool*) to an overlap set, and outputs each corresponding overlap interval (to a list L_O). The

```

COMPUTEOVERLAPINTERVALS(segment& U,
segmentList& LL, segmentList& LT, segmentList& LR, segmentList& LO)
{
  Step (1) { COMPUTESETS(LT, LO)
  Step (2) { EXPANDSETS(LL, LO)
             EXPANDSETS(LR, LO)
  Step (3) { COMPUTESETS(LL, LO)
             COMPUTESETS(LR, LO)
  Step (4) { RELAXSETS(U, LO)

```

Figure 3.15: The COMPUTEOVERLAPINTERVALS algorithm.

segments of the input list L_I are sorted in increasing order of associated grid index (left to right, bottom to top). Processing of list L_I consists of repeated application of procedure COMPUTEOVERLAPINTERVAL (Figure 3.17), with argument S_O set equal to a sentinel value. The sentinel values causes S_O to be set equal to the first segment of L_I . Thus, the net effect of procedure COMPUTEOVERLAPINTERVAL is the following: **(1)** the segments from L_I that overlap the first segment of L_I are deleted, and **(2)** the first segment of L_I is shrunk until it equals the corresponding overlap interval. The resulting overlap interval is then appended to the output list L_O , but only if the corresponding overlap set does not connect two resolved surface portions. The algorithm terminates when list L_I becomes empty.

```

COMPUTESETS(segmentList& LI, segmentList& LO)
{
  while not LI.isEmpty()
  do { /* SO is initialized to sentinel value */
      segment SO ← segment(DBL_MIN, DBL_MAX)
      bool notType4Set ← COMPUTEOVERLAPSEGMENT(LI, SO)
      if notType4Set then LO.add(SO)

```

Figure 3.16: The COMPUTESETS procedure.

As an example of procedure COMPUTESETS, consider Figure 3.18(a). A cell with direction of representation \vec{z} was cut by the tool (gray). After the connecting sets in the direction of \vec{x} were generated, procedure COMPUTESETS processed the *tool* segments (orange) in the following way. The first pass of procedure COMPUTEOVERLAPINTERVAL iterated over the segments in bottom to top order. The overlap of the first two segments was calculated. Next, the overlap of this result and the next segment was calculated. After the third segment from the top was reached, the overlap of segments examined so far (S_O) became a single point (which corresponds split plane A). As shown in Figure 3.18(b), procedure COM-

```

COMPUTEOVERLAPINTERVAL(segmentList& L, segment& SO)
{
  bool allResolved ← true
  L.initIteration()
  while L.hasNext()
  {
    segment SC ← L.getNext()
    if SO.overlaps(SC)
    {
      do {
        if SO.length() > 0 then SO ← overlap(SO, SC)
        then {
          if SC.hasUnresolvedEndpoint() then allResolved ← false
          L.deleteNext()
        }
      }
    }
  }
  return ( not allResolved )
}

```

Figure 3.17: The COMPUTEOVERLAPINTERVAL procedure.

PUTEOVERLAPINTERVAL also removed all segments of the output overlap set from the *tool* set; the overlap of the remaining segments (see overlap interval B) was then calculated via second pass of procedure COMPUTEOVERLAPINTERVAL. After this second pass, the *tool* set became empty (see Figure 3.18(c)).

Algorithm COMPUTEOVERLAPINTERVALS as described so far processes each segment set independently. Consequently, the overlap intervals produced from *tool* segments (during step 1) may overlap the members of another connecting set. Thus, during step 2 the *tool* overlap sets are expanded to include additional members from the *left-of-tool* and *right-of-tool* sets; this processing helps reduce the total number of split planes calculated (i.e., fewer segments will be intersected by two planes). Step 2 is implemented using procedure EXPANDSETS (Figure 3.19), which computes the overlap of each *tool* overlap interval (input as list L_O) with segments from another set (input as list L_I). Similar to procedure COMPUTESETS, procedure EXPANDSETS shrinks segments (the *tool* overlap intervals) using procedure COMPUTEOVERLAPINTERVAL. Note that the overlapped *left-of-tool* and *right-of-tool* segments are removed from the *left-of-tool* and *right-of-tool* sets, respectively. Therefore, step 3 processes any remaining *left-of-tool* and *right-of-tool* segments.

As an example of step 2, let us consider further the Figure 3.18 example. The overlap intervals A and B from step 1 were examined in sequence. Since overlap interval A was already a single point, it could shrink no further; however, some *left-of-tool* segments were overlapped by interval A , and these segments were removed from that set as a result (see Figure 3.18(c)). In contrast, overlap interval B overlapped several *right-of-tool* segments, and was shrunk until it represented the overlap of those segments as well; the shrunken overlap interval B is shown in

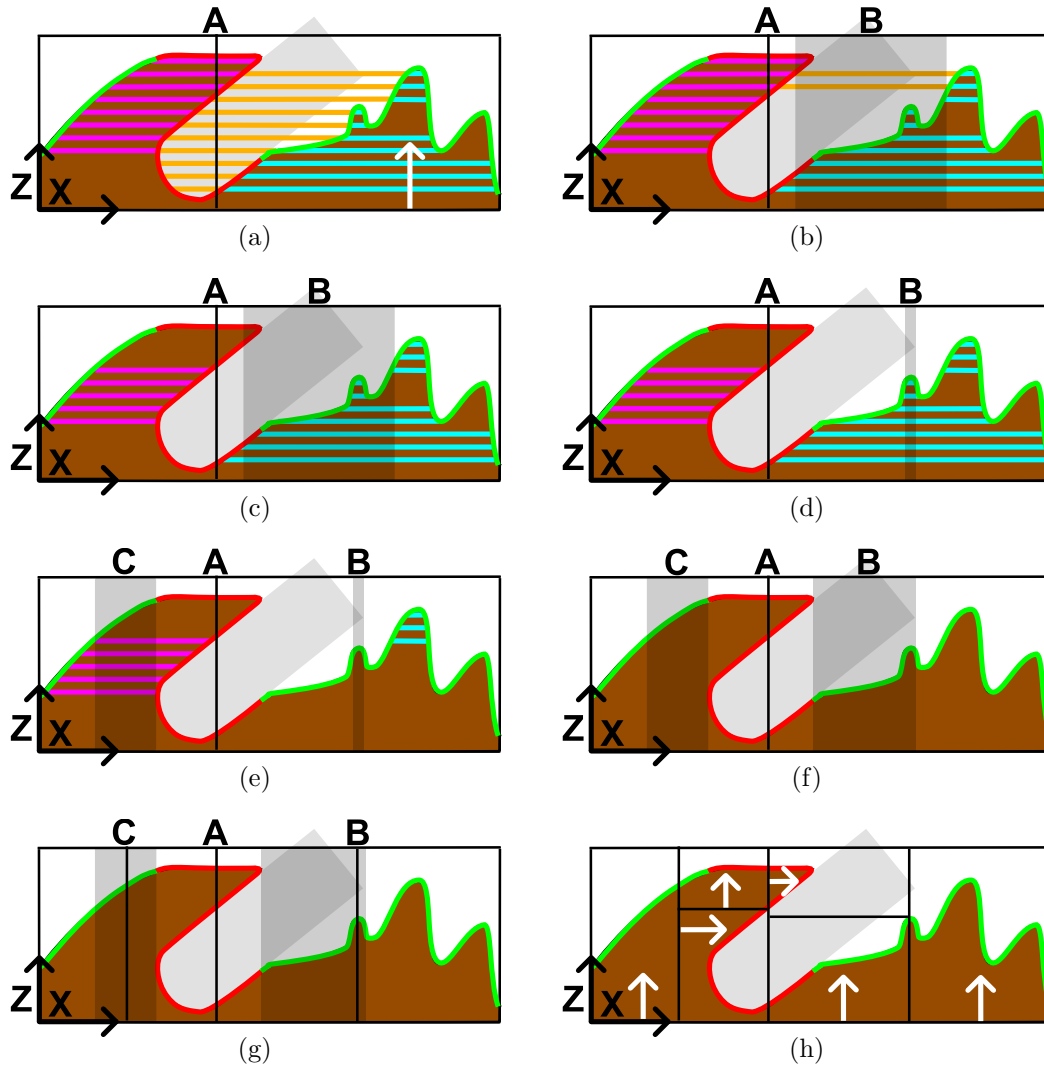


Figure 3.18: An example showing calculation (A-F) and usage (G-H) of a DSP series. (a) *Step 1* processing begins. The orange segments are examined in a bottom to top sequence. After examination of the third segment from the top, the overlap of segments examined so far becomes a single point; the black line labelled A corresponds to its range. (b) The orange segments that remain are then examined; the resulting overlap interval is indicated with the dark box labelled B. (c) *Step 2* processing begins since no orange segments remain. All purple segments overlapped by point A are removed from the purple set. (d) Overlap interval B is shrunk to represent its overlap with several cyan segments. (e) *Step 3* processing begins. The overlap of all purple segments is found, and is indicated by the dark box labelled C. The three remaining cyan segments are discarded since they connect two resolved curves. (f) *Step 4* processing begins. Overlap interval B is expanded until it touches the unresolved surface. DSP series computation is now complete. (g) The overlap intervals are selected for use in alphabetical order, and a single split plane is calculated from each one. (h) Two sign conflict splits are necessary after these direct splits are applied.

```

EXPANDSETS(segmentList&  $L_I$ , segmentList&  $L_O$ )
{
   $L_O$ .initIteration()
  while  $L_O$ .hasNext()
  {
    do {
      segment&  $S_O \leftarrow L_O$ .getNext()
      COMPUTEOVERLAPSEGMENT( $L_I$ ,  $S_O$ )
      if  $L_I$ .isEmpty() then break
    }
  }
}

```

Figure 3.19: The EXPANDSETS procedure.

Figure 3.18(d). Next, the overlapped *right-of-tool* segments were removed from the *right-of-tool* set, and step 2 processing was concluded (see Figure 3.18(e)). During step 3 processing, the remaining *left-of-tool* and *right-of-tool* segments were processed using procedure COMPUTESETS (as were the *tool* segments beforehand). An overlap interval C for the *left-of-tool* segments was found, but the *right-of-tool* segments were simply deleted (since they connect resolved curves).

The purpose of step 4 (see procedure RELAXSETS, Figure 3.20) is to increase the range of overlap intervals corresponding to exterior split planes; doing so can improve the performance of the FFS heuristic discussed in Section 3.3. Let U be a bounding interval for the unresolved surface along the axis of computation \vec{a} . It is possible that several of the uppermost segments of a connecting set will connect two resolved curve portions. An exterior split need not divide such segments in two. Thus, any overlap interval outside the range of bounding interval U can be expanded to touch the range of U . For example, overlap interval B of Figure 3.18(e) is shown after expansion in Figure 3.18(f).

```

RELAXSETS(segment&  $U$ , segmentList&  $L_O$ )
{
   $L_O$ .initIteration()
  while  $L_O$ .hasNext()
  {
    do {
      segment&  $S_O \leftarrow L_O$ .getNext()
      /* a segment is defined by its left (L) and right (R) endpoints */
      if  $S_O.L > U.R$  then  $S_O.L \leftarrow U.R$ 
      if  $S_O.R < U.L$  then  $S_O.R \leftarrow U.L$ 
    }
  }
}

```

Figure 3.20: The RELAXSETS procedure.

So far only 2D direct split examples have been shown. It is difficult to give a complete 3D example. Instead the example of Figure 3.21 is given, which shows two cross sections (green and blue in Figure 3.21(a)) of a cell with direction of representation \vec{z} . The solid was drilled into at an angle, and when the tool reached a certain depth, an overhang occurred (see the red unresolved curves). The green

cross section illustrates the DSP series for direction \vec{x} (see splits A , B and C of Figure 3.21(b)); the blue cross section illustrates the DSP series for direction \vec{y} (see splits A and B of Figure 3.21(c)) and \vec{z} (see split A of Figure 3.21(d)). The split planes and overlap intervals shown apply to the nonvisible slices as well.

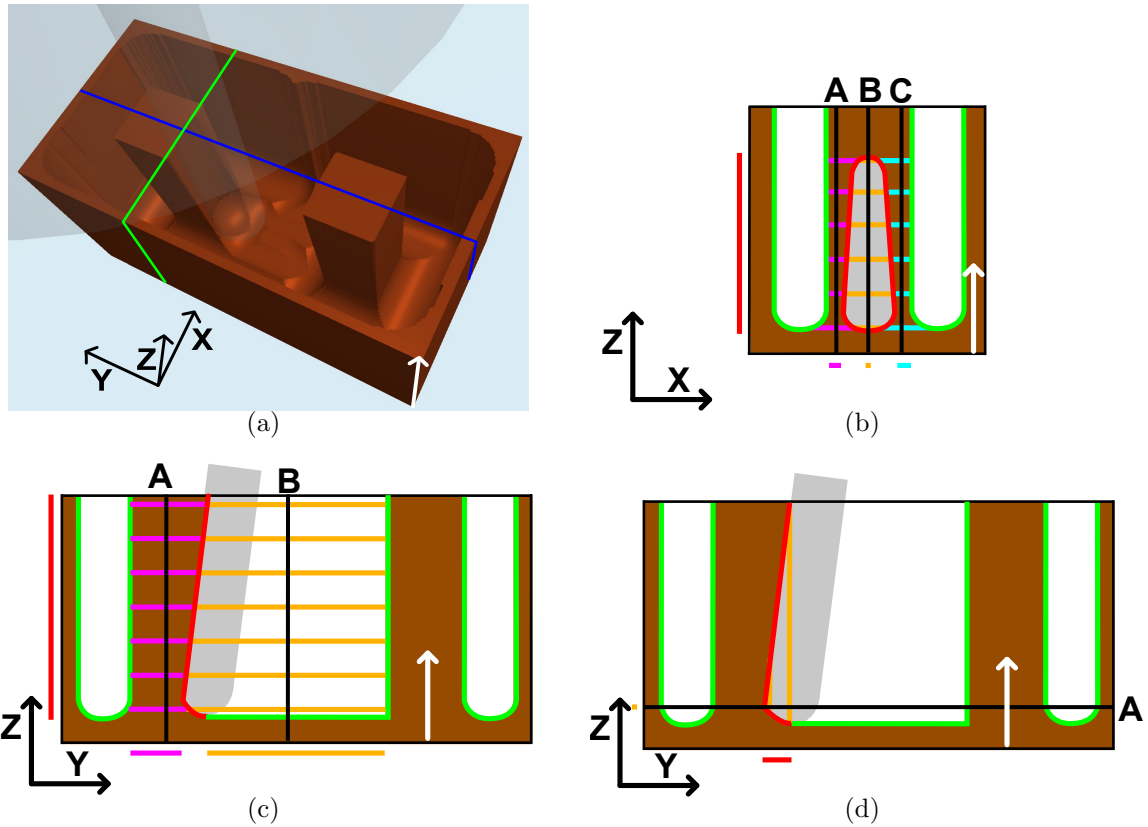


Figure 3.21: A 3D direct split example. In part (a), a second cut made into a previously cut cell. A DSP series is computed along the \vec{x} axis (part (b)), the \vec{y} axis (part (c)), and \vec{z} axis (part (d)). Each DSP series is illustrated using a single 2D slice. The part (b) slice corresponds to the green outline, and the parts (c) and (d) slices correspond to the blue outline. Along the exterior of the cells, the corresponding unresolved bad regions (red) and overlap intervals (purple, orange and cyan) are shown.

3.3 Minimization Selection Heuristics

Typically, when a cell requires subdivision, one of possibly several split planes could be chosen. This section will explain the system of heuristic rules used to make this choice. We already know that complete split planes occur within the range of a separating interval (see the black intervals along axis \vec{x} in Figure 3.9), and direct split planes occur within the range of an overlap interval (see Figure 3.14(a)). Also, a point corresponding to a partial split plane can also be viewed as having a range of only one split plane (see the black points along axis \vec{z} in Figure 3.9). So in all cases, a split plane is a member of a family of split planes all within some range specified by an interval (along an axis). Such an interval is referred to as a *split interval*. Therefore, the opening statement of this section can be restated as the following: when a cell requires subdivision, one of possibly several split intervals must be chosen; next, a single split plane within this interval is used to subdivide the cell.

Ideally, the number of subdivision operations should be minimized (splitting is both expensive computationally and in terms of memory). However, to find the minimum number of splits, **(1)** a global analysis of the entire toolpath would be required and **(2)** knowledge of the final answer may be required; such an algorithm would be computationally expensive and difficult to develop. Instead of finding the minimum number of splits, a multidirectional heightmap utilizes several heuristics intended to minimize its total subdivision count: Closest-Toolcut-Direction (CTD), Split-Plane-Type (SPT), Furthest-From-Side (FFS), and Random-Selection (RS). Each rule is assigned a priority value. Initially, the highest priority rule is used to select a split interval. If there are ties, the rule with the next highest priority is used to break that tie, and so on. It is not clear which combination of rules is most effective; I investigate this question experimentally in Section 6.2. Here all heuristics will be described in turn (and in isolation).

The **Closest-Toolcut-Direction** rule assigns priority to a split interval based upon the size of the angle between the members of the split interval (which are parallel to each other) and the tool's direction \vec{t} ; the smaller this angle, the higher the priority. Without loss of generality, this rule will be further explored using split planes (split intervals having only one member).

Figure 3.22 illustrates a scenario where the rule proves beneficial. The figure shows five in-between steps (a) through (e) of angled drilling into the side of a block. Initially the cut is representable during step (a), but an overhang occurs during step (b). The lower half of the figure shows what happens if the opposite of the CTD

rule is applied: the red split plane, the plane whose angle with \vec{t} is the largest, is chosen. First, the right child then requires an additional split plane (cyan). More importantly, notice that this choice ultimately causes a similar overhang to occur for the left child during step (c). If same bad choice is then made again, the outcome is *repetitive bad behaviour*, a process the CTD rule could have prevented. Specifically, during step (b) this rule would have selected the more tool-aligned split plane (black). As a result, no additional subdivision would have been required (see top half of steps (b) to (e)). Another example of repetitive bad behaviour is given in Figure 3.23.

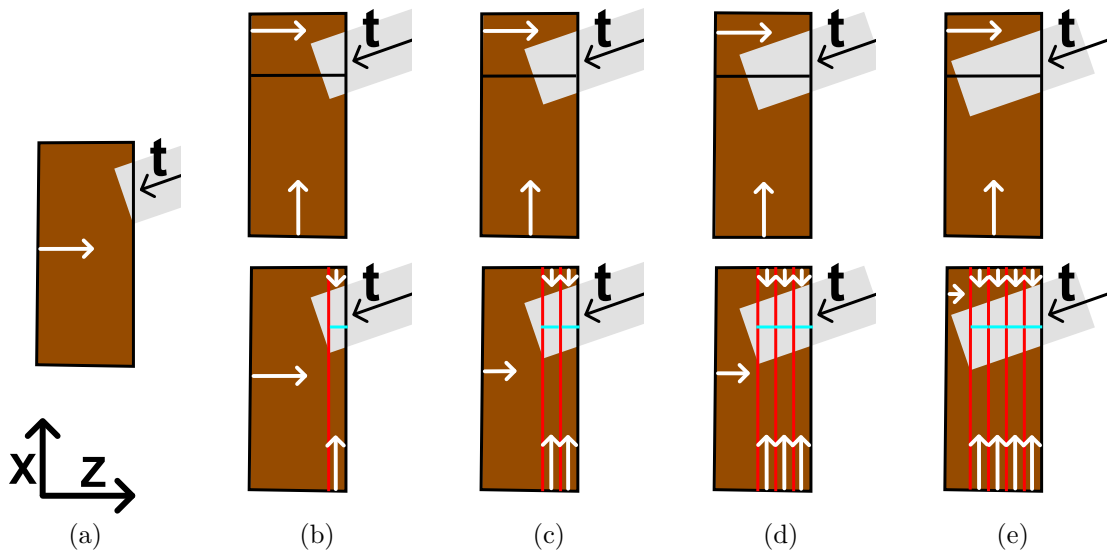


Figure 3.22: A 2D illustration of the Closest-Toolcut-Direction (CTD) rule. The toolcut becomes unrepresentable during step (b). **The Upper Half:** if the tool-aligned split plane (black) is chosen, no further subdivision is required during steps (c) to (e). **The Lower Half:** if the nontool-aligned split plane is chosen for each step, repetitive bad behaviour is the outcome.

The **Split-Plane-Type** rule assigns priority to each split interval according to the type of split plane associated with it. Given two split planes with different types, choosing one over the other can lead to fewer cell subdivisions for the current in-between step. We know that after at most two complete splits no further subdivision is required (see Figure 3.7(a)). Also, often less subdivision is required if a direct split that is the sole member of its DSP series is selected instead of other direct splits (compare Figures 3.21(c) and 3.21(d) for instance). However, a complete split is often superior to a sole-member direct split, since additional (indirect and/or sign conflict) splits may be required after the latter split plane type. Consequently, the

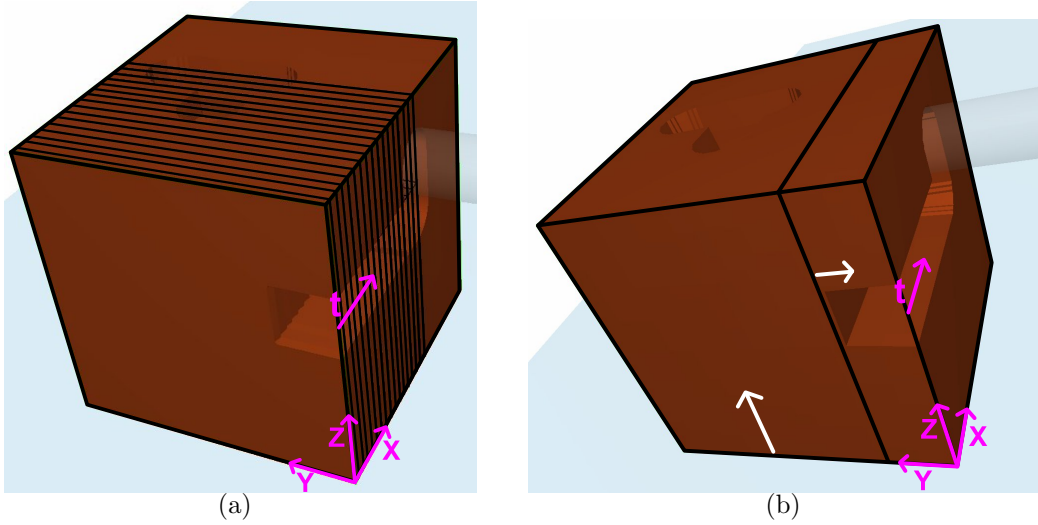


Figure 3.23: A 3D illustration of how the Closest-Toolcut-Direction rule avoids repetitive bad behaviour. **(a)** In response to an overhang, a split plane angled 90 degrees relative to the tool's direction \vec{t} is chosen. A similar overhang then occurs for the right child. This same bad choice is then made again, and the process repeats. **(b)** The split plane angled 0 degrees with respect to \vec{t} is chosen; no further subdivision is required.

SPT rule ranks split intervals in the following order: complete splits; sole-member direct splits; and with the same rank, partial splits and non-sole-member direct splits.

The goal of the **Furthest-From-Side** rule is to ensure that the dimensions of each child cell are as equal as possible. That is, the ideal child cell is a cube according to this rule. Thus, the ideal split plane would divide a cell at the middle along the longest dimension (see the dashed line labelled S_A in Figure 3.24). Of course, such a split plane may not exist; so the closer a split plane is to this ideal, the higher the priority that split plane will be assigned.

The FFS rule helps reduce the total number of subdivisions required. For example, note that the FFS rule would also select the black split plane in Figure 3.22(b), and thus would also avoid repetitive bad behaviour. However, the continuation of this example shown in Figure 3.24(a) better illustrates the strength of the FFS rule. The multidirectional heightmap has been cut a second time by the tool (gray). To the left, we see what happens if the four close together red split planes were used to resolve the first cut; then four cells must be subdivided twice in response to the second cut. On the other hand, the right side of Figure 3.24(a) shows how if the black split were chosen to resolve the first cut instead, only one cell must be subdivided.

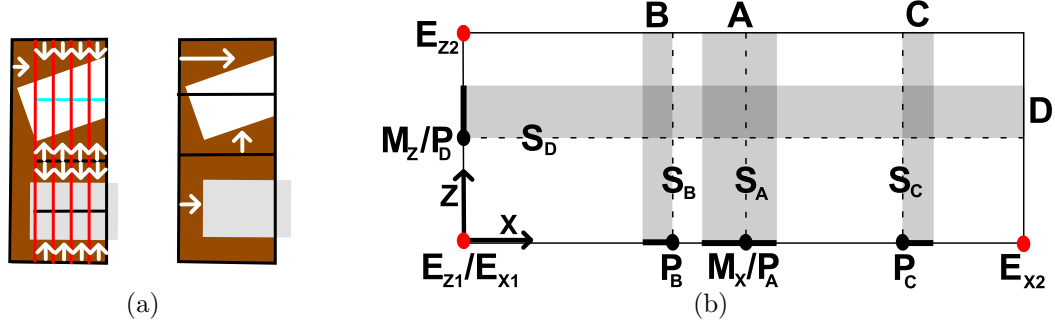


Figure 3.24: The Furthest-From-Side rule. **(a)** A continuation of the Figure 3.22 example. **Left:** Five children must be subdivided. **Right:** One child must be subdivided. **(b)** An illustration of the FFS priority equation.

To define the FFS rule more precisely, let R be a split interval along an axis \vec{a} , $M_{\vec{a}}$ be the midpoint along the cell edge $(E_{\vec{a}1}, E_{\vec{a}2})$ parallel to \vec{a} , and S_R be the plane within the range of R closest to point $M_{\vec{a}}$. Then the FFS rule assigns priority to split interval R based upon the equation $|M_{\vec{a}} - E_{\vec{a}1}| - |M_{\vec{a}} - P_R|$, where P_R is the intersection point between plane S_R and split interval R ; the larger this result, the higher the priority assigned. The $|M_{\vec{a}} - P_R|$ term of the equation measures how close plane S_R is to midpoint $M_{\vec{a}}$. When this term is subtracted from $|M_{\vec{a}} - E_{\vec{a}1}|$, we get plane S_R 's distance towards the closest surrounding cell boundary. Thus, the FFS priority equation will assign highest priority to the following split range: the one containing the split plane furthest from its surrounding cell boundaries.

The example in Figure 3.24(b) illustrates how the FFS rule assigns priority. Observe that if splitting is restricted to one axis only, the closer a split interval is the middle of its corresponding dimension, the higher the priority it is assigned. Consider the \vec{x} axis for instance. Split interval A has the highest priority because it contains midpoint M_x . Likewise, split interval B has higher priority than split interval C since point P_B is closer to midpoint M_x than point P_C . Split interval D , which is along the \vec{z} axis, is also available if the arbitrary one axis restriction is removed. Similar to split interval A , split interval D also contains the midpoint (M_z) of its corresponding cell edge. However, point P_D is closer to a surrounding cell boundary (see the two red points along the \vec{z} axis) than any other point within a split interval (compare to points P_A , P_B and P_C). Split interval D will thus be assigned the lowest priority.

The **Random-Selection** rule simply chooses a split interval at random, thereby avoiding repetitive behaviour. This rule is also useful for evaluating the effectiveness of various combinations of the other rules, as is done in Section 6.2.

After a split interval is selected using a combination of the above rules (CTD, SPT, FFS, RS), a single split plane must be chosen from within its range. Currently, my implementation selects the split plane that is closest to the middle. For instance, if split interval A in Figure 3.24(b) were selected, split plane S_A would be used for subdivision.

Chapter 4

Stock Rendering with Continuous Seams

This chapter explains how a multidirectional heightmap is rendered as a continuous surface. Specifically, the goal is to construct a triangular mesh from the sample data, and then, using the normals stored with each height, render it with smooth shading via OpenGL. Ideally, the portion of mesh corresponding to each individual heightmap could be constructed in isolation (a trivial task). However, as is covered in Section 4.1, these surface fragments do not always fit together in a continuous way. The solution to this problem, discussed in Section 4.2, is to modify the heightmap data structure to include additional surface data. This new type of heightmap is called a *3-Way Heightmap* (3-map), and how to construct a triangular mesh from such a data structure is the topic of Section 4.3. Some other advantages of 3-maps over regular heightmaps are discussed in Section 4.4.

4.1 Discontinuous Seams

To make rendering easier, the multidirectional heightmap surface representation was simplified as follows. The space a stock occupies is uniformly subdivided into cubical cells by a 3D grid called the *sampling grid*. The purpose of the sampling grid is to ensure that all surface data is aligned in a convenient way. In particular, a point is said to be *grid-aligned* if it coincides with a grid line; every height corresponds to a particular grid-aligned point in space, referred to as a *surface sample*, specified relative to the stock frame. Likewise, a plane is said to be *grid-aligned* if it coincides with at least one cube face (it never occupies the interior of any cube); all split

planes are grid-aligned.

While the above simplifications are helpful, discontinuous seams between adjacent cells are still possible. To be more precise, a *common boundary* is defined as the plane of intersection between two adjacent cells. The curve of intersection between a heightmap surface and a common boundary is a piecewise linear curve referred to as a *boundary curve (b-curve)*. A b-curve is an approximation of the curve of intersection between a common boundary and the precise surface being machined, a curve referred to as the *true curve (t-curve)*. Thus, a discontinuous seam occurs when two heightmaps sharing a common boundary have different b-curve representations for the corresponding t-curve.

An illustration of a discontinuous seam is shown in Figure 4.1(a). In the figure, we see the common boundary between a heightmap in the direction of \vec{x} (red) and a heightmap in the direction of \vec{z} (blue). The t-curve is black and coloured dashed lines indicate its b-curve representations. Also visible are the grid-aligned surface samples (coloured dots) each b-curve is composed of.

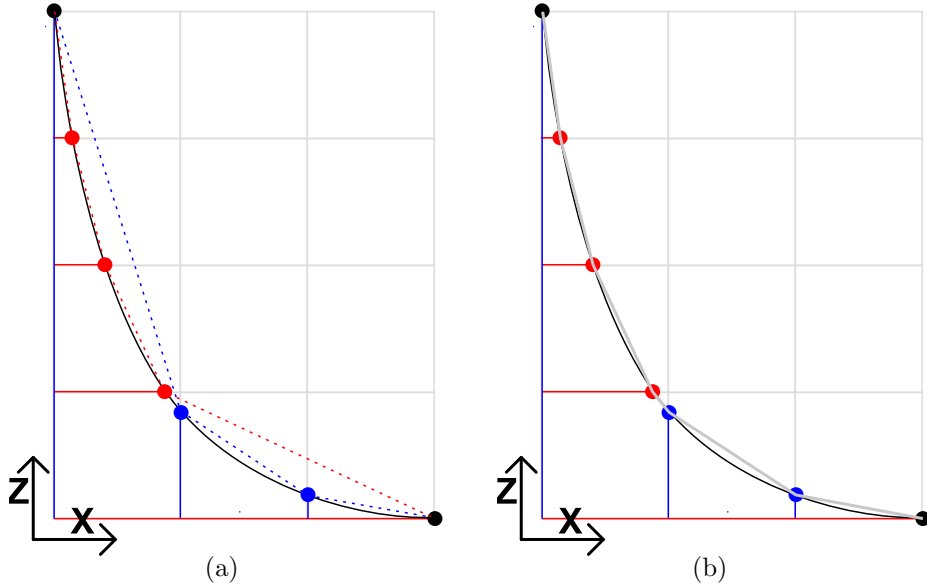


Figure 4.1: A comparison of the b-curve representations between two heightmaps (part **(a)**) and two 3-maps (part **(b)**). The red heightmap/3-map has sampling direction \vec{x} , and the blue heightmap/3-map has sampling direction \vec{z} . The true boundary curve is black and coloured dashed lines indicate its b-curve representations. Only in the case of part **(b)** are both b-curve representations the same (both sampling directions are used).

Notice that the red b-curve provides a different representation of the t-curve than that of the blue b-curve. In general, this problem can only occur between

two heightmaps with different sampling directions. In this case, the red heightmap samples the t-curve along grid lines parallel to the \vec{x} axis, and the blue heightmap samples the t-curve along grid lines parallel to the \vec{z} axis. Surface samples common to both b-curves only occur where these grid lines intersect (see the black dots in Figure 4.1(a)). Provided that each b-curve is extended to include the sample points that belong to the other, the seam will be continuous. That is, both b-curves will provide the representation shown as a gray piecewise linear curve in Figure 4.1(b). In the next section, we will see how the heightmap data structure can be extended so that, given two b-curves sharing a common boundary, both representations will be equivalent.

4.2 3-Way Heightmaps

A *3-Way Heightmap (3-map)* samples the tool along all three stock axis directions (three ways) rather than just one. Specifically, a 3-map will potentially embed a 1D heightmap between all pairs of adjacent heights (including non-cell-boundary heights, to simplify rendering and reduce approximation error). To be more precise, let H_L be a height that is adjacent to a smaller height H_S (see Figure 4.2(a)). There is a grid line coinciding with each of heights H_L and H_S , and the space between them is divided into squares by other grid lines called *dividing lines*. Along each dividing line above H_S but below or equal to H_L , an *embedded heightmap* stores a height measured relative to height H_L ; heights H_L and H_S are referred to as *parent heights* (green) and the other heights are referred to as *embedded heights* (red). Note that an embedded heightmap does not exist for every pair of adjacent heights; see Figure 4.2(b) for example.

In Figure 4.1(a), we saw that a discontinuous seam can occur between two heightmaps with different sampling directions. Since a 3-map samples along two directions within each boundary face of its cell, discontinuous seams will not occur if 3-maps are used in place of regular heightmaps. As an illustration, consider how 3-maps could be applied to Figure 4.1(a). In the blue heightmap, a 1D heightmap consisting of three red heights would be embedded in between the two leftmost blue heights. Likewise, a 1D heightmap consisting two blue heights would be embedded in between the two bottommost red heights of the red heightmap. Since both heightmaps now sample the t-curve along all grid lines (two directions), the seam will be continuous as in Figure 4.1(b).

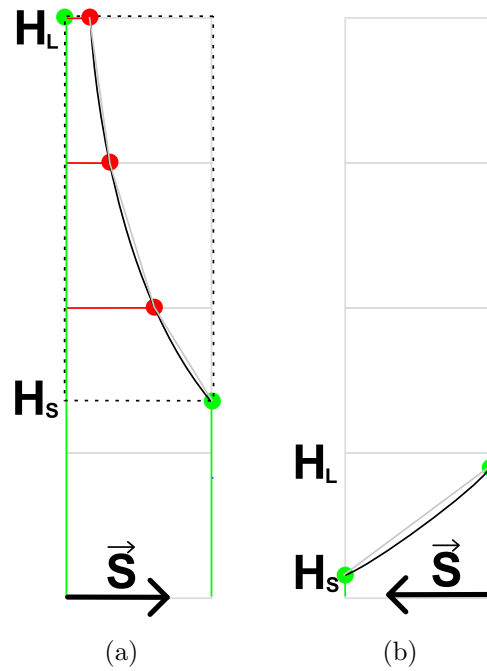


Figure 4.2: Embedded heightmap illustrations. Embedded heights are red; parent heights are green and labelled H_L (large) and H_S (small); surface samples are appropriately-coloured dots; t-curves are black; and b-curves are gray. **(a)** An embedded 1D heightmap with three entries. **(b)** A pair of heights that do not require an embedded heightmap.

4.3 3-Map Mesh Construction

A 3-map mesh consists of two components: the *interior mesh*, which consists of all triangles not coincident with a face of the 3-map’s cell; and the *exterior mesh*, which consists of all triangles on a face of the 3-map’s cell. The interior mesh is generated using a specialized version of the marching cubes algorithm [12], and the exterior mesh is generated using a specialized version of the marching squares algorithm, the unpublished 2D version of marching cubes [16]. But while marching cubes has an ambiguity problem, when applied to CNC machining there is enough additional information to properly interpret the ambiguous faces; bilinear interpolation [16] is not necessary.

4.3.1 Exterior Mesh Construction

Besides partitioning the interior of a 3-map’s cell into cubical cells, the sampling grid also partitions a 3-map’s cell boundary into cells called *squares*. An exterior mesh is constructed through examination of these squares. Only the nonempty squares coincident with the boundary of the multidirectional heightmap must be considered; nonempty squares that occur along a common boundary are not visible to the viewer. For each nonempty square examined, a line loop that bounds its solid portion is calculated and subsequently triangulated. For a solid square, the line loop is simply the square’s corner points; for a partial square, it can be shown that all surface samples as well as solid corner points form the line loop required, a fact I will demonstrate with a case-based proof.

The possibilities for a partial square depend on its relationship with the 3-map’s direction of representation \vec{v} . A *nonside square* has a normal parallel to \vec{v} . Consequently, each edge of a nonside square may coincide with an embedded surface sample, and is referred to as *embedded edge*. In contrast, a *side square* has a normal perpendicular to \vec{v} . Thus, for a side square, only the two edges perpendicular to \vec{v} are embedded edges; each other edge may contain a parent surface sample, and is referred to as *parent edge*.

The only possible partial side squares (i.e., the only squares functional with respect to the direction of representation) are shown in Figure 4.3. Again, parent heights are shown in green, and embedded heights in red. In addition, the points of each line loop are shown as dots, which are black in the case of solid corners, and in the case of surface samples, green or red as appropriate. Observe that each square has two surface samples interconnected by a separating curve (a gray line

segment). Thus, in all cases the surface samples and solid corners compose the line loop required.

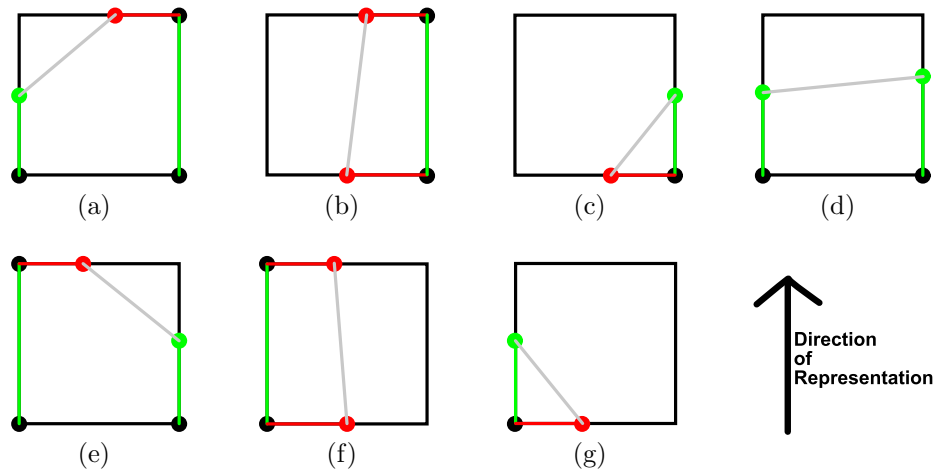


Figure 4.3: The partial side square cases. Solid corners are black; embedded heights and surface samples are red; parent heights and surface samples are green; separating curves are gray line segments.

The reason for only seven partial side squares should be explained more precisely. The relationship between two parent heights solely determines what embedded heights, if any, will be present between them (see Figure 4.2). Also, there is only one way a parent edge can be partial: the bottom half must be solid, and the upper half must be empty. Thus, since a side square has two parent edges, each of which can be partial, empty or full, there are nine possible cases; but only seven of these cases are partial.

The only possible partial nonside squares (up to symmetry) are shown in Figure 4.4. Since each corner is either coincident with a parent height (green dot) or not (no dot), there are sixteen possible nonside square configurations; but only fourteen of these are partial. Only four partial cases (five if the second interpretation of configuration (d) is considered as a separate case) are given in Figure 4.4 because all other cases are rotationally symmetric to one of these cases. From each of cases (a) to (c), three more cases can be generated from repeated rotation in the plane by 90 degrees. Similarly, another case is obtained from each interpretation of case (d) via a planar rotation of 90 degrees.

Configuration (d), the “ambiguous face”, has two interpretations [16]. The left interpretation has one solid portion and two empty portions (*closed interpretation*), whereas the right interpretation has one empty portion and two solid portions (*open*

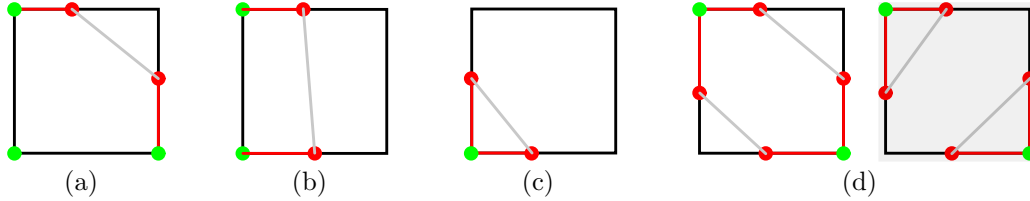


Figure 4.4: The partial nonside square cases. Embedded heights and surface samples are red; parent heights are green dots; separating curves are gray line segments.

interpretation). Although the open interpretation can occur (e.g., imagine a ball nose tool that skims across the face from corner to corner), and is easy to detect, it was not implemented to simplify interior mesh generation. I primarily developed the 3-map data structure to prevent the occurrence of discontinuous seams; a single consistent interpretation of the ambiguous face is sufficient to achieve this goal.

It may seem that 3-maps will not always guarantee continuous seams since there is no configuration (d) for side squares (the necessary sample data is not available for this type of side square to be possible). This is not a problem since if a second corner of a side square is removed, the heightmap will be resampled in a new direction: the side square will become a nonside square.

The algorithm given above is similar to the marching squares algorithm. As noted earlier, one difference arises because extra information is available to interpret the ambiguous face. The only other difference is that the marching squares algorithm does not make any distinction between squares based upon “direction of representation”. Specifically, it only applies those cases shown in Figure 4.4.

4.3.2 Interior Mesh Construction

An interior mesh is constructed via an examination of all cubes inside a 3-map. For each cube, a line loop that bounds its separating surface is constructed and subsequently triangulated. By careful consideration of all cases, I have developed an algorithm to generate this line loop. These cases (up to symmetry) are given in Figure 4.5) as “proof” of its correctness.

As previously mentioned, a parent edge can be in one of three possible states (empty, partial, or full). Thus, there are $3^4 - 2 = 79$ partial cube configurations. However, up to rotational and reflective symmetry, only the 19 configurations (five with more than one interpretation) shown in Figure 4.5 are unique. Additional cases can be generated from each cube via a 90 degree rotation about the direction

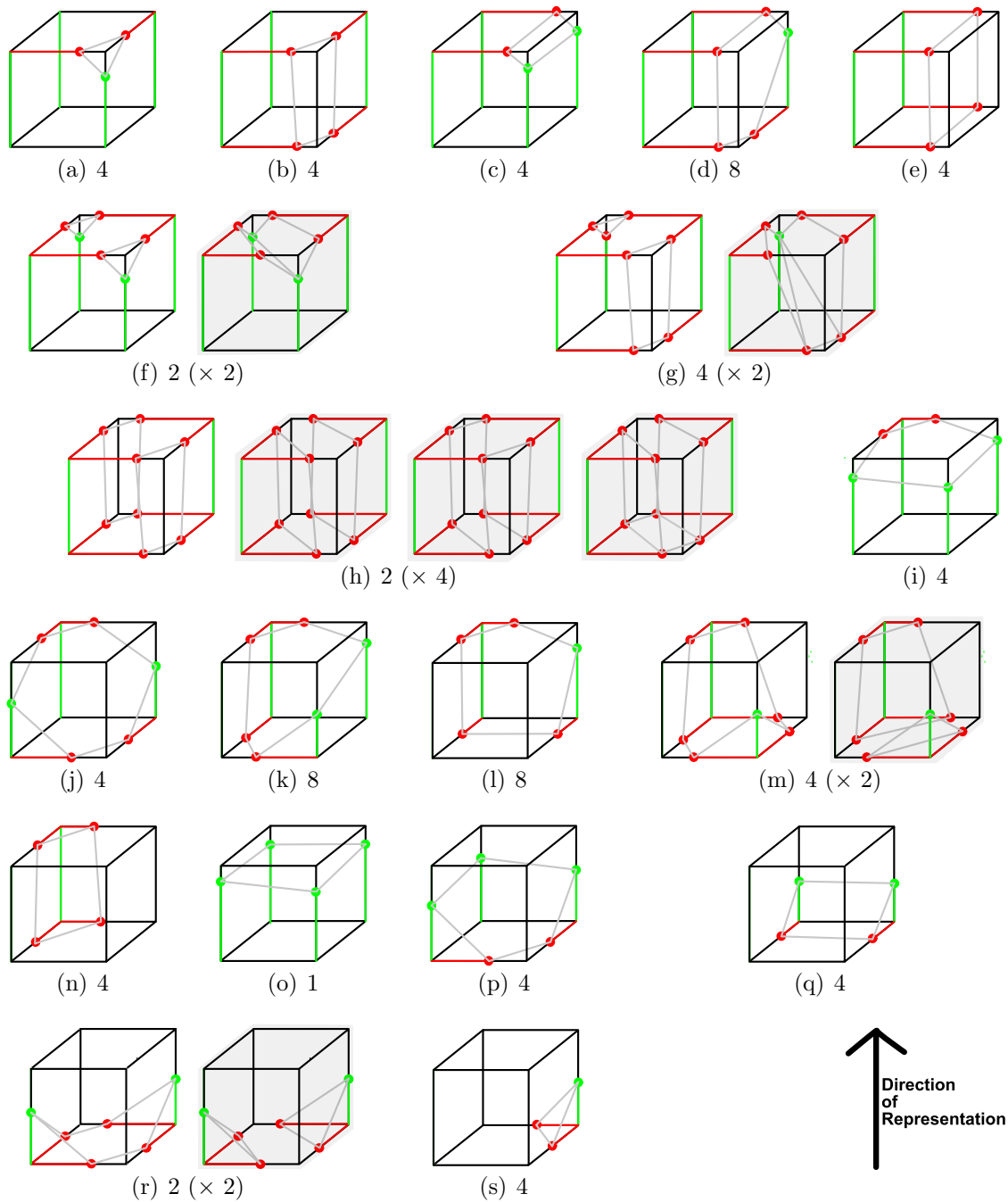


Figure 4.5: The partial cube cases. Embedded heights and surface samples are red; parent heights and surface samples are green; solid line loops are gray. Below each cube, the total number of symmetric configurations is given. Some configurations have more than one interpretation due to the presence of ambiguous face(s). The closed-interpretation cases (dark-coloured) are not implemented.

of representation (except for case (o)). Additional cases can also be generated from cubes (d) , (g) , (k) , (l) and (m) by moving a single parent surface sample to a different parent edge. The total number of cases that can be generated from each cube is noted just below it.

Configurations (f) , (g) , (m) , and (r) have an ambiguous face and thus, two interpretations; case (h) has two ambiguous faces (four interpretations). Thus, seven more unique cases (dark-coloured) would be needed to properly handle the ambiguous face (the total number of cases would be $79 + 18 = 97$). Necessary line-loop diagonals are shown for some of the alternative interpretations; if the ambiguous face is to be interpreted as closed, diagonals that coincide with the ambiguous face must be avoided. That said, my implementation only supports the leftmost interpretation of each of these configurations for reasons I discussed earlier.

The following algorithm is used to derive the line loop for all supported (lightly-coloured) cases. Iterate over all side squares (see Figure 4.3) in counterclockwise order. For cases (a) , (b) , and (c) add the surface samples in bottom to top order; for cases (e) , (f) , and (g) add the surface samples in top to bottom order; and finally, add the surface samples of case (d) in left to right order. However, do not add a point a second time if it was already added for the previous square.

Observe that cases (f) , (g) and (h) of Figure 4.5 have more than one line loop. The above algorithm still works but is applied to such cubes twice, each time to a pair of side squares that share a nonfull parent edge.

It should be noted how the interior mesh construction algorithm differs from marching cubes. The first version of marching cubes considered 256 cases (a cube corner can be either absent or present) [12]. However, more cases are required to deal with the ambiguous face [16]. In contrast, a 3-map cube has the restriction that, relative to the direction of representation, a lower corner cannot be absent unless the corner above it is also absent. Thus, there are 81 cases in total (97 cases if ambiguous faces are interpreted properly), as was previously mentioned. Moreover, this restriction makes it easy to generate the cases on the fly; the marching cubes algorithm uses a case lookup table instead.

4.4 3-Maps versus Regular Heightmaps

Although the 3-map data structure was developed to deal with the discontinuous seams problem, it has other benefits as well. Consider a heightmap and a 3-map representing the same design surface with the same density. The 3-map will use more memory because of the embedded 1D heightmaps; but the 3-map will also have lower approximation error, and it can be rendered with better quality of shading due to the additional normal samples it stores. These concepts are illustrated by Figures 4.6 and 4.7. Figure 4.6 compares heightmap (part (b)) and 3-map (part (c)) wireframe representations of a “vertical wall” (part (a)). Clearly, the 3-map representation is substantially more accurate.

Figure 4.7 compares heightmap and 3-map representations of a design surface with vertical walls. Furthermore, the comparison is shown at various densities (32, 64, 128), and each 3-map representation is shown once with flat shading and once with Gourand shading. Observe that, even at the highest density (128), the heightmap representation of the curved walls is jagged. Also, Gourand shading cannot help here since all normals happen to have the same direction, and the tool-cut would thus be invisible if it were used. In contrast, the 3-map representation suffers from neither of these problems; if Gourand shading is enabled, the curved walls are smoothly represented even at the lowest density (32).

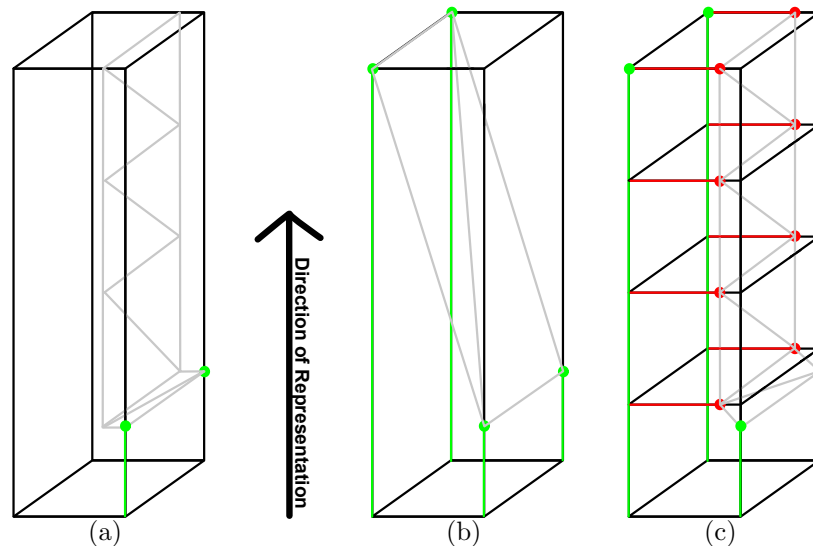


Figure 4.6: Heightmap (b) and 3-map (c) wireframe representations of a “vertical wall” (a). Parent heights and surface samples are green; embedded heights and surface samples are red; triangular mesh edges are gray.










Density	Heightmap	3-map, flat shading	3-map, Gourand shading
32			
64			
128			

Figure 4.7: For a particular design surface, a comparison of a single-cell heightmap representation and a single-cell 3-map representation.

Chapter 5

Implementation Issues

This chapter covers some additional implementation details not covered in chapters 3 and 4. Section 5.1 describes the memory layout of a multidirectional heightmap. Section 5.2 describes how to compute the solid-representing line segments necessary for good-bad map generation and direct split plane computation. Finally, Section 5.3 covers good-bad map region bounding.

5.1 Data Structure Implementation

In this section I explain the memory layout of a multidirectional heightmap for the current implementation. It is illustrated in Figure 5.1. I utilize arrays for purposes of fast spatial look up of data; lazy allocation is used to help minimize memory usage.

A multidirectional heightmap is represented by a **Stock** object, which links to the root node of a k D-tree composed of **Cell** nodes. The grid data of each **Cell** is stored in a separate **ThreeMap** object. This allows allocation of the grid, possibly a large quantity of data, to be forgone if the **Cell** is **PARTIAL** or **EMPTY**, as indicated by the **solidness** field. Each entry of the *grid* is a pointer to one of three types of **GridEntry** objects. A **PartialGridEntry** object stores a parent height/normal and two arrays of embedded height/normal pairs. A **SolidGridEntry** object is assigned to a grid entry if no embedded heightmap is needed there and the parent height located there is solid; an **EmptyGridEntry** object has a similar purpose. Thus, many grid entries will only use the memory required to store a pointer. The other two grids (*DLrebuild* and *DLid*) are for display list management.

A multidirectional heightmap's memory usage depends on the number of cells, the size of each cell, and the way in which each cell has been cut. In other words, the amount of memory used depends a great deal on the toolpath itself; not just on the stock's density. Therefore, it is difficult to estimate the memory usage of a simulation beforehand.

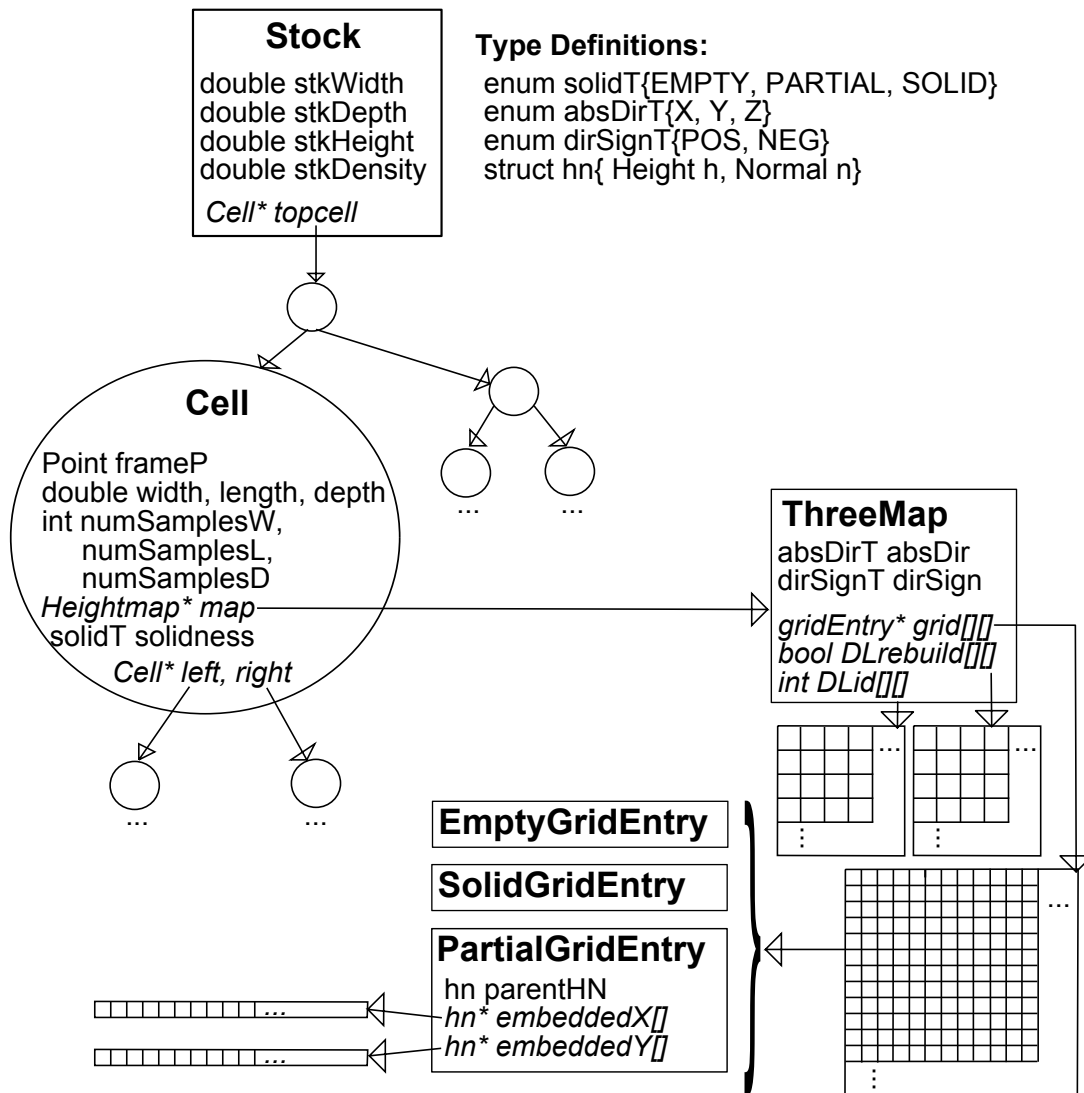


Figure 5.1: Multidirectional heightmap memory layout.

5.2 Solid Segment Construction

Recall that constructive solid geometry (CSG) is applied when generating a good-bad map: the tool line segments are subtracted from the cell line segments (see

Figure 3.3(c)). The purpose of this section is to explain how the solid-representing line segments (*solid segments*) are computed from heightmaps (Section 5.2.1) and the tool’s swept volume (Section 5.2.2).

5.2.1 Heightmap Solid Segments

This section discusses how solid segments parallel to a direction \vec{v} are constructed from a heightmap with direction \vec{u} . These segments are needed to compute good-bad maps (Section 3.1) and direct splits (Section 3.2.3). Here I will focus on the case where \vec{v} is perpendicular \vec{u} ; the opposite case is trivial since each height directly maps to a solid segment.

To see how the solid segments are computed, it is helpful to think of the heightmap data as being partitioned into slices of heights interconnected by line segments. Each slice is considered independently. The general idea is to use a segment’s orthogonal projection to prune the number of rays that must be intersected with it. Whenever enough intersection data is available along a ray, a solid segment is output.

The algorithm applied to each slice is illustrated in Figure 5.2(a). The segments of the slice are examined in sequence, starting with a segment that touches cell boundary face F_S , and ending with a segment that touches cell boundary face F_E . Let segment S_C (red) be the segment currently under consideration. The projection S_P (purple) of segment S_C onto face F_S is computed, and for each grid point within this projection, a ray intersection point (green) with segment S_C is calculated.

Since other rays may be processed in between successive intersection calculations involving a particular ray, it is necessary to allocate and maintain a buffer of previous intersections. Whenever a new intersection I_N is found along a ray, solid segments are output in the following cases:

- (1) There is no previous intersection, and the new intersection I_N is backfacing. In this case, the output segment is bounded by face F_S and intersection I_N ; see the green segments in Figure 5.2(b) for example.
- (2) The previous intersection I_P is frontfacing, and the new intersection I_N is backfacing. In this case, the output segment is bounded by intersection I_P and intersection I_N ; see the red segment in Figure 5.2(b) for example.

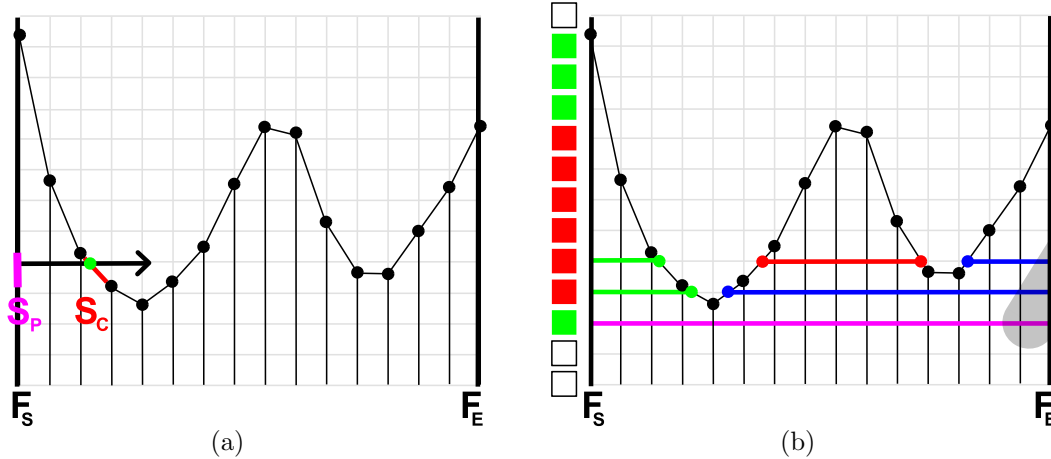


Figure 5.2: Solid segment computation and usage. **(a)** Generation of solid segments for the horizontal direction from a heightmap with heights in the vertical direction. **(b)** After the part (a) heightmap is cut by the tool (gray), usage of solid segments for computation of the horizontally-directed good-bad map.

- (3) The previous intersection I_P is frontfacing, and there is no new intersection. In this case, the output segment is bounded by intersection I_P and face F_E ; see the blue segments in Figure 5.2(b) for example.

There may be no intersections along some rays; this is simple to deal with. For each grid point below the minimum height of the slice, a segment bounded by face F_S and face F_E is output; see the purple segment in Figure 5.2(b) for example.

The good-bad map pixels generated from a particular slice of solid segments are shown in Figure 5.2(b) to the left of face F_S . This figure also demonstrates an optimization that is applied to the above algorithm when it is used to generate a good-bad map. If a ray does not intersect the tool, then it is not necessary to compute any solid segments along that ray. That is, after the orthogonal projection S_P of the current segment S_C is determined, it is not necessary to cast rays from the grid points within projection S_P . Instead, it is sufficient to count the number of times each grid point is contained by a segment's projection; this is enough information to distinguish between good and bad pixels. Furthermore, to distinguish between good-positive and good-negative pixels, an orientation value (backfacing/frontfacing) corresponding to the first intersection found must be stored with each grid point; if more than one intersection occurs, the value is simply ignored.

In Section 3.2.3 I mentioned that direct split planes are also computed from solid segments that occur along tool-intersecting rays (the *left-of-tool*, *tool* and *right-of-*

tool sets of line segments). I made this decision based upon two facts: **(1)** the above optimization would have no benefit if all solid segments were considered when calculating direct split planes, and **(2)** consideration of the *left-of-tool*, *tool* and *right-of-tool* segments is sufficient to handle situations where all unresolved regions consist entirely of bad pixels.

5.2.2 Tool Solid Segments

ToolSim approximates the tool’s swept surface as a piecewise polygonal surface (see Figure 2.5). It is not necessary for this surface to fully bound the swept volume if the stock is represented as a regular heightmap, because only the closest intersection with the tool is needed; solid intervals are necessary when a multidirectional heightmap is used instead. Consequently, extra triangles are added to each portion of swept surface between two in-between frames to obtain a triangle strip loop, and a “stamp” is used to bound each of the two holes that remain. As a result, my implementation uses one stamp per in-between step. In contrast, stamps are only necessary to deal with discontinuous direction changes when the stock is represented as a regular heightmap.

Spatial look-up techniques similar to those described in Section 5.2.1 are applied to efficiently compute solid intervals from the stamps and triangles. Furthermore, the solid intervals are stored in an intermediate buffer, one for each stock axis direction, since often the same information is needed for more than one cell. The time needed to compute most simulations is drastically reduced by doing so.

Each buffer is a 2-dimensional array to allow access to entries based upon spatial location. This approach is not without its disadvantages, however. If the tool’s orthogonal projection onto a buffer has an angle close to 45 degrees, there will be many unused entries. Whenever all intersection data must be accessed, I simply scan every entry and ignore the empty ones, a rather inefficient approach (and the slowdown can be quite noticeable). One possible way to remove this limitation is to store each table as an array of rows, each possibly having a distinct number of entries, and use the first entry of each row to store its spatial displacement relative to a full grid.

5.3 Good-bad Map Region Bounding

This section describes an algorithm that computes a tight bounding rectangle for each region composed of a particular pixel type P . Bounding rectangles are necessary to compute split planes from good-bad maps. As discussed in Section 3.2.1, split planes are calculated from good-bad maps via a comparison of the orthogonal projections of regions (see Figure 3.9). The two projections resulting from a region can be obtained directly from a tight bounding rectangle for that region.

Given a good-bad map and a pixel type P , all rectangles that tightly bound pixels of type P are computed using a row by row examination of all pixels. The following steps are carried out for each row (ignoring the boundary conditions):

- (1) Each pixel is examined in left to right order to compute a bounding rectangle for each continuous sequence of type P pixels. This is done by keeping track of when a P pixel sequence is entered and when it is left. All bounding rectangles found are added to the **current list**.
- (2) The **current list** is compared to another list of rectangles, the **previous list**, which includes only rectangles that bound pixels in previous rows. Any **current list** member C that is adjacent to a **previous list** member B is expanded enough to fully overlap B . B is then removed from the **previous list**.
- (3) Any **previous list** members that remain after step (2) are moved to the **done list**, which contains all bounding rectangles that completely bound a region of P pixels.
- (4) All **current list** members are added to the **previous list**. Go back to **Step 1**, but apply the steps to the next row.

Figure 5.3 illustrates the algorithm as applied to a simple example.

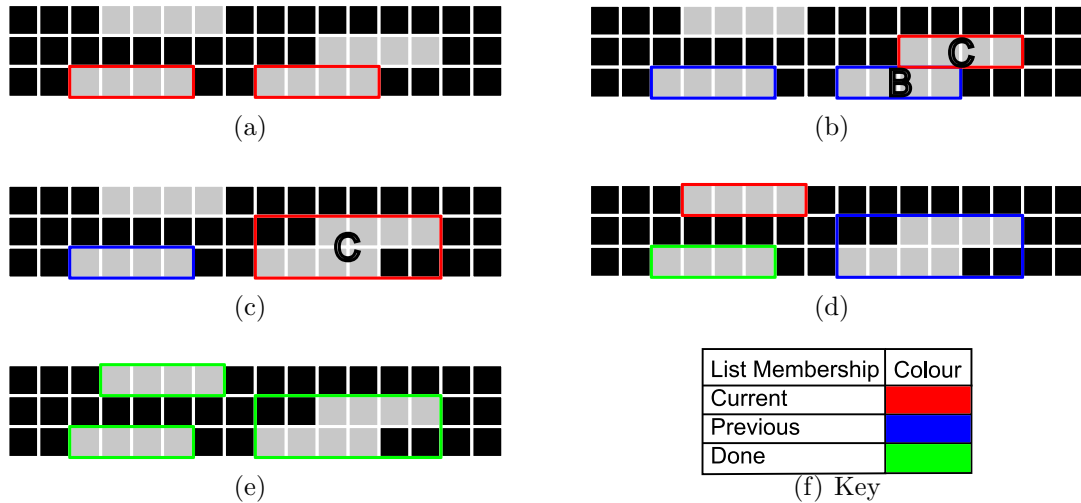


Figure 5.3: An example illustrating how bounding rectangles are calculated for all gray pixel regions. (a) Bounding rectangles are computed for the first row and added to the current list. (b) The current list becomes the previous list. A bounding box from the second row is added to the new current list. The current list is compared to the previous list. A current list member C is adjacent to a previous list member B . (c) Rect C is expanded to fully cover B . B is deleted from the previous list. (d) The previous list member that was not adjacent to a current list member is added to done list. The current list becomes the previous list. A bounding box from the third row is added to the new current list. (e) The current list is compared to the previous list. The current list member is not adjacent to the previous list member. So the previous list member is moved to the done list. The current list member is also added to done list since the third row is the last.

Chapter 6

Evaluation By Simulation

In this chapter I experimentally evaluate various aspects of a multidirectional heightmap's performance. Section 6.1 presents numerous examples of solid objects that can be represented. Section 6.2 evaluates how successful the heuristics presented in Section 3.3 are. Section 6.3 evaluates the CPU and memory usage of a multidirectional heightmap. Finally, Section 6.4 addresses the relationship between maximum approximation error and stock density.

6.1 Simulation Examples

This section presents some examples that were machined using a multidirectional heightmap (Figures 6.1, 6.2, 6.3, 6.4, 6.5, 6.6, 6.7, 6.8, 6.9, 6.10, 6.11). These examples were used as test cases for the various experiments presented in this chapter. The simulation parameters used to compute these examples are given in Table 6.1.

Simulation Parameter	Value
stock density	100
grazing curve density	256
number of in-between steps	10

Table 6.1: Simulation parameters used to compute the examples.

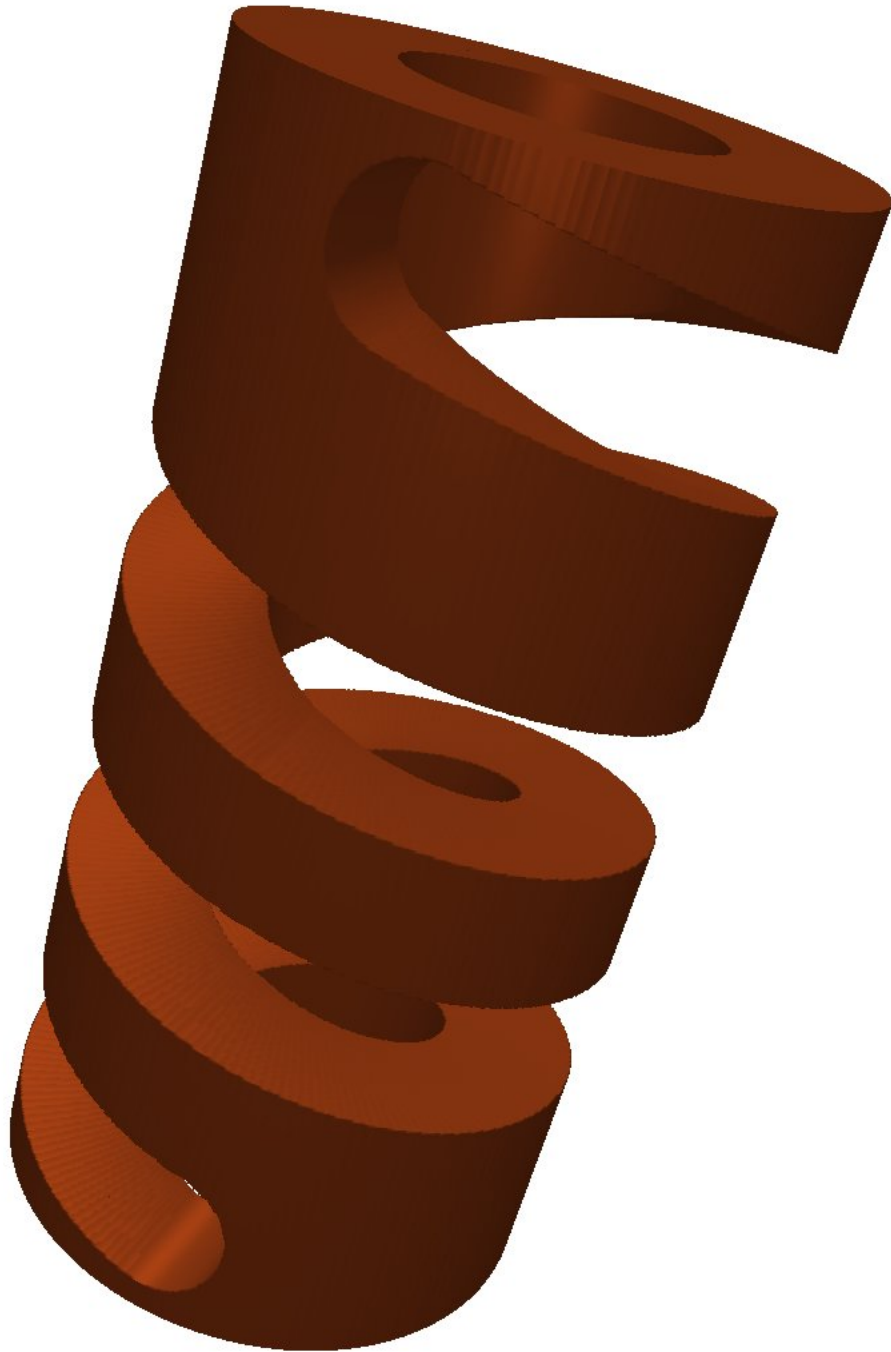


Figure 6.1: Cylinder Spiral (CS).

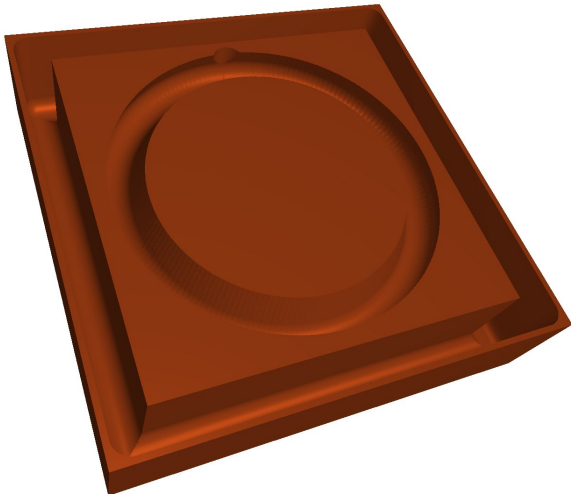


Figure 6.2: Donut With Overhang 1 (D1).

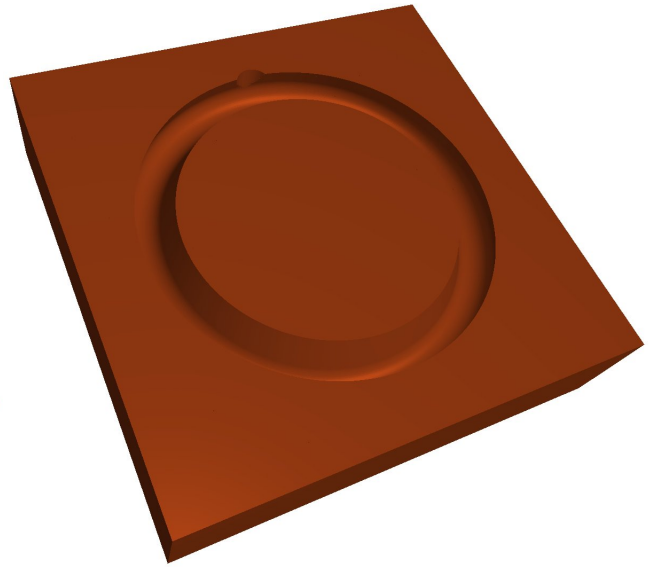


Figure 6.3: Donut With Overhang 2 (D2).



Figure 6.4: Donut Spiral (DS1).



Figure 6.5: Donut Spiral (DS2).

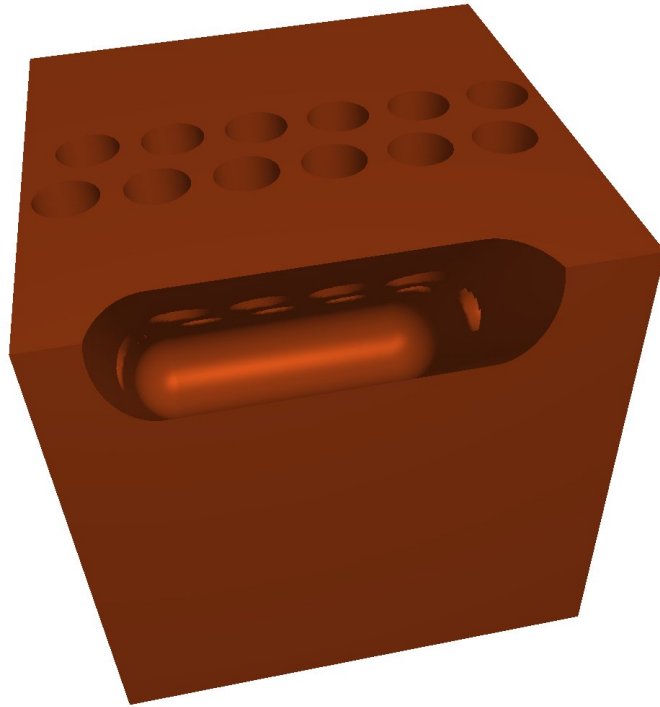


Figure 6.6: Top face drilling and angled edge cut (MA1). The top face cuts connect with the other cut.

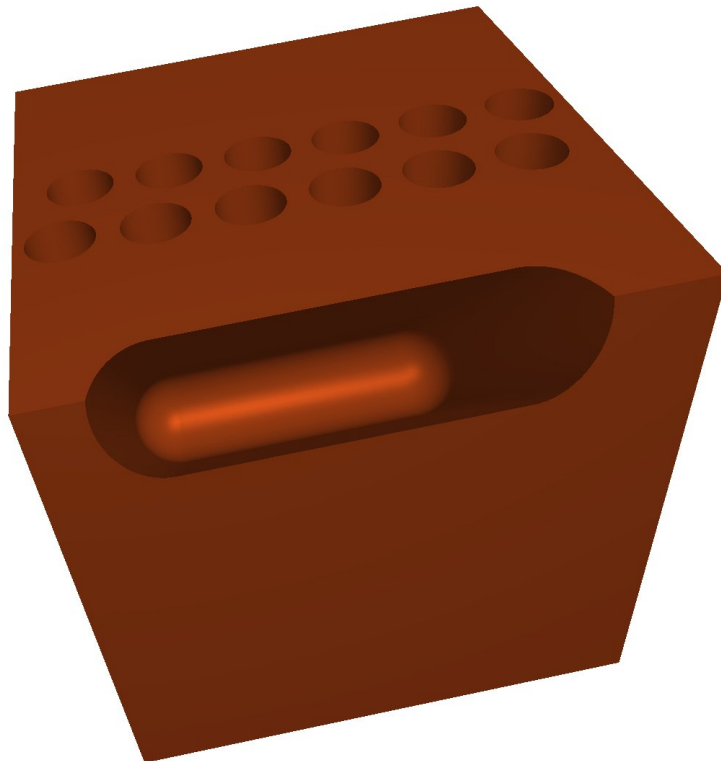


Figure 6.7: Top face drilling and angled edge cut (MA2).

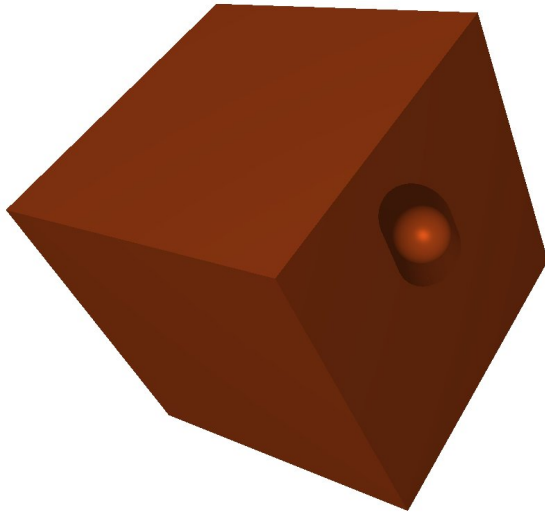


Figure 6.8: NonAA 45 Degree Drill (NA).

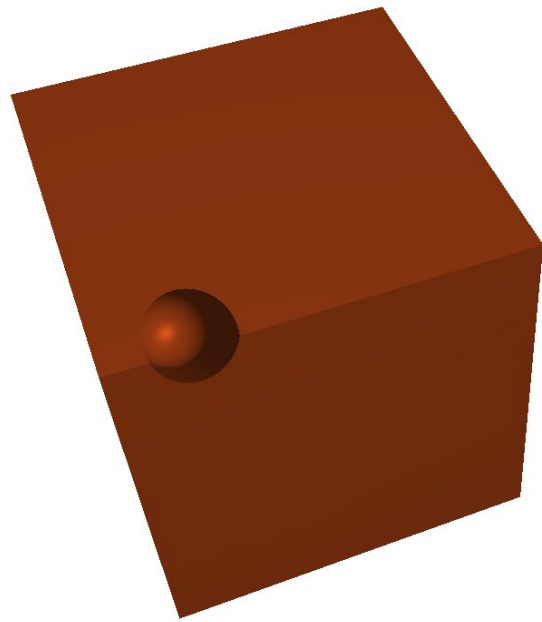


Figure 6.9: Angled drilling aligned with plane (A1).

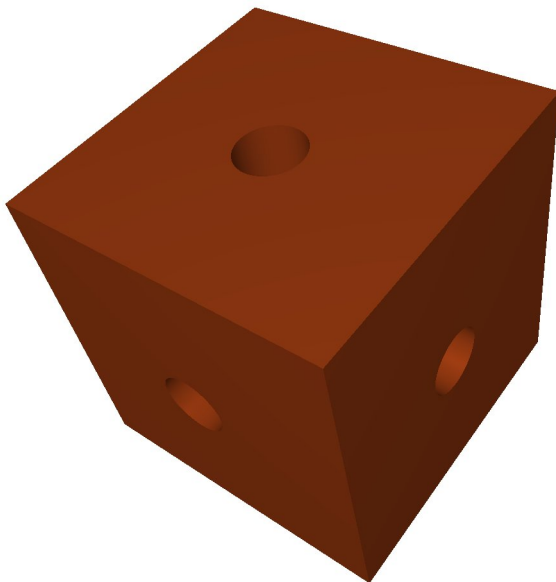


Figure 6.10: Drill To Center of Cube Along Each Axis (A2).

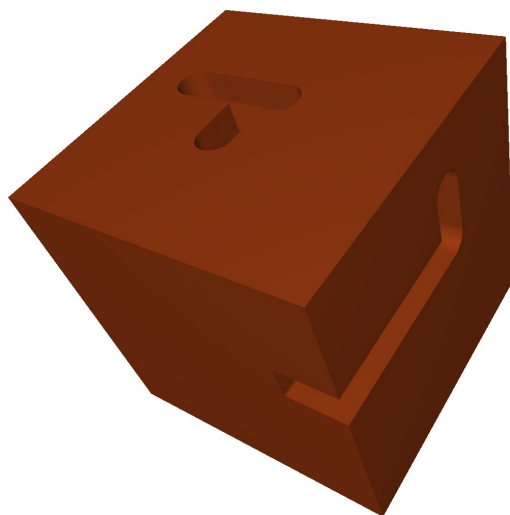


Figure 6.11: Two planar toolcuts (A3).

6.2 Heuristic Evaluation

The goal of this section is to determine which combination of heuristics from Section 3.3 is most effective at minimizing the total cell subdivision count. I evaluate each rule combination (*ruleset*) experimentally using the examples from Section 6.1 as test cases (and the simulation parameters of Table 6.1). Each ruleset is denoted as a tuple of rules in order of descending priority. To help avoid repetitive bad behaviour, the Random-Selection (RS) rule is always used and is always the lowest priority rule.

Recall the other heuristic rules are Furthest-From-Side (FFS), Closest-Toolcut-Direction (CTD) and Split-Plane-Type (SPT). Table 6.2 tabulates the total subdivision count for ruleset versus test case. I computed additional data (see Tables 6.3 and 6.4) from Table 6.2 using a few metrics for ruleset performance measurement. Table 6.3 tabulates the metric *RS-percentage*, subdivision count as a percentage of Random-Selection subdivision count, for ruleset versus test case (each cell $t_{i,j}$ of this table is $d_{i+1,j}/d_{1,j} * 100$ where d is a cell from Table 6.2). The premise here is that if a rule cannot outperform the mindless Random-Selection rule (i.e, its RS-percentage is greater than 100), it probably is not an effective rule. The last row of Table 6.3 is the lowest RS-percentage achieved for each test case.

Table 6.4 measures a ruleset’s overall performance across all test cases. Specifically, it tabulates the difference between RS-percentage and the lowest RS-percentage achieved for ruleset versus test case (each cell $t_{i,j}$ of this table is $d_{i,j} - d_{lastrow,j}$ where d is a cell from Table 6.3); this was done to compare how close a ruleset’s performance is to the best performing rule. The rightmost column contains the sum of all results (a *summation value*) for each row. The lower a ruleset’s summation value, the more likely it is to be good. Of course, such a metric fails to take into account poor performance for particular examples; the data of Table 6.4 should not be ignored.

The first seven cases (*complex cases*) are complex enough to require a large number of subdivisions no matter what combination of rules is applied. In contrast, the remaining cases (*simple cases*) require few subdivisions. Very good or bad heuristic performance is more likely to be experienced with simple cases. Thus, separate summation values are given for each type of case: the SC column is the sum of all complex results, and the SS column is the sum of all simple results.

Note the extremely poor performance (i.e., RS-percentage > 100) of ruleset (CTD, RS) when applied to the following test cases: CS, 143%; D1, 933%; D2,

Name	CS	D1	D2	DS1	DS2	MA1	MA2	NA	A1	A2	A3
RS	2893	528	508	2448	911	707	321	31	4	16	1
CTD, RS	4143	4928	4642	16816	10946	683	150	43	3	14	1
SPT, RS	1390	276	269	1547	843	584	147	22	1	11	1
FFS, RS	282	159	132	1246	286	562	224	12	10	5	2
FFS, CTD, RS	285	165	134	1216	284	538	209	12	10	5	2
FFS, SPT, RS	279	155	133	1203	286	543	210	12	10	5	2
FFS, CTD, SPT, RS	285	165	134	1231	284	539	201	12	10	5	2
FFS, SPT, CTD, RS	279	157	133	1203	284	539	202	12	10	5	2
FESF, RS	410	176	149	1208	313	641	240	15	8	22	1
FEFS, CTD, RS	366	193	101	1151	356	626	138	15	8	19	1
FEFS, SPT, RS	363	143	123	1052	278	543	148	10	1	13	1
FEFS, CTD, SPT, RS	358	120	96	1163	307	560	128	12	1	12	1
FEFS, SPT, CTD, RS	342	130	125	1057	285	531	122	10	1	14	1
FESF-FFS, RS	345	169	142	1212	314	637	174	15	8	17	1
FEFS-FFS, CTD, RS	314	172	79	1148	339	635	174	15	8	17	1
FEFS-FFS, SPT, RS	384	147	125	1067	279	552	161	10	1	14	1
FEFS-FFS, CTD, SPT, RS	314	103	81	1109	293	550	163	12	1	12	1
FEFS-FFS, SPT, CTD, RS	379	131	125	1059	289	547	158	10	1	15	1
SPT, FEFS-FFS, RS	929	220	152	1228	399	554	136	28	1	15	1
SPT, CTD, RS	709	478	373	1919	1393	537	122	30	1	15	1
SPT, FEFS-FFS, CTD, RS	539	205	127	1228	458	549	133	28	1	15	1
SPT, CTD, FEFS-FFS, RS	701	468	370	1906	1373	540	120	30	1	15	1

Table 6.2: The total subdivision count for ruleset versus test case.

Name	CS	D1	D2	DS1	DS2	MA1	MA2	NA	A1	A2	A3
CTD, RS	143	933	914	687	1202	97	47	139	75	88	100
SPT, RS	48	52	53	63	93	83	46	71	25	69	100
FFS, RS	10	30	26	51	31	79	70	39	250	31	200
FFS, CTD, RS	10	31	26	50	31	76	65	39	250	31	200
FFS, SPT, RS	10	29	26	49	31	77	65	39	250	31	200
FFS, CTD, SPT, RS	10	31	26	50	31	76	63	39	250	31	200
FFS, SPT, CTD, RS	10	30	26	49	31	76	63	39	250	31	200
FESF, RS	14	33	29	49	34	91	75	48	200	138	100
FEFS, CTD, RS	13	37	20	47	39	89	43	48	200	119	100
FEFS, SPT, RS	13	27	24	43	31	77	46	32	25	81	100
FEFS, CTD, SPT, RS	12	23	19	48	34	79	40	39	25	75	100
FEFS, SPT, CTD, RS	12	25	25	43	31	75	38	32	25	88	100
FESF-FFS, RS	12	32	28	50	34	90	54	48	200	106	100
FEFS-FFS, CTD, RS	11	33	16	47	37	90	54	48	200	106	100
FEFS-FFS, SPT, RS	13	28	25	44	31	78	50	32	25	88	100
FEFS-FFS, CTD, SPT, RS	11	20	16	45	32	78	51	39	25	75	100
FEFS-FFS, SPT, CTD, RS	13	25	25	43	32	77	49	32	25	94	100
SPT, FESF-FFS, RS	32	42	30	50	44	78	42	90	25	94	100
SPT, CTD, RS	25	91	73	78	153	76	38	97	25	94	100
SPT, FESF-FFS, CTD, RS	19	39	25	50	50	78	41	90	25	94	100
SPT, CTD, FESF-FFS, RS	24	89	73	78	151	76	37	97	25	94	100
Lowest RS-Percentage	10	20	16	43	31	75	37	32	25	31	100

Table 6.3: For each test case, the percentage increase/decrease of the total subdivision count achieved by each ruleset set compared to the Random-Selection rule. Each cell $t_{i,j}$ of this table is $d_{i+1,j}/d_{1,j} * 100$ where d is a cell from Table 6.2. The last row is the greatest percentage decrease achieved for each test case.

Name	CS	D1	D2	DS1	DS2	MA1	MA2	NA	A1	A2	A3	SC	SS	ST
CTD, RS	134	914	898	644	1171	21	9	106	50	56	0	3791	213	4004
SPT, RS	38	33	37	20	62	7	8	39	0	38	0	207	76	283
FFS, RS	0	11	10	8	1	4	32	6	225	0	100	67	331	298
FFS, CTD, RS	0	12	11	7	1	1	28	6	225	0	100	59	331	290
FFS, SPT, RS	0	10	11	6	1	2	28	6	225	0	100	57	331	289
FFS, CTD, SPT, RS	0	12	11	7	1	1	25	6	225	0	100	57	331	289
FFS, SPT, CTD, RS	0	10	11	6	1	1	26	6	225	0	100	54	331	286
FESF, RS	5	14	14	6	4	16	37	16	175	106	0	95	297	393
FESF, CTD, RS	3	17	4	4	9	13	6	16	175	88	0	56	279	335
FESF, SPT, RS	3	8	9	0	0	2	9	0	0	50	0	30	50	80
FESF, CTD, SPT, RS	3	3	3	5	3	4	2	6	0	44	0	24	50	74
FESF, SPT, CTD, RS	2	5	9	0	1	0	1	0	0	56	0	18	56	74
FESF-FFS, RS	2	13	12	7	4	15	17	16	175	75	0	69	266	336
FESF-FFS, CTD, RS	1	13	0	4	7	15	17	16	175	75	0	56	266	323
FESF-FFS, SPT, RS	4	8	9	1	0	3	13	0	0	56	0	37	56	94
FESF-FFS, CTD, SPT, RS	1	0	0	2	2	3	13	6	0	44	0	22	50	72
FESF-FFS, SPT, CTD, RS	3	5	9	0	1	2	12	0	0	63	0	33	63	96
SPT, FESF-FFS, RS	22	22	14	7	13	3	5	58	0	63	0	88	121	208
SPT, CTD, RS	15	71	58	35	122	1	1	65	0	63	0	368	127	430
SPT, FESF-FFS, CTD, RS	9	19	9	7	20	3	4	58	0	63	0	71	121	192
SPT, CTD, FESF-FFS, RS	15	69	57	35	120	1	0	65	0	63	0	297	127	424

Table 6.4: Except for the three rightmost columns, each cell $t_{i,j}$ of this table is $d_{i,j} - d_{i,astrow,j}$ where d is a cell from Table 6.3. Each cell of column SC is the sum of all complex case results, each cell of column SS is the sum of all simple case results, and each cell of column ST is the sum of the sums from column SC and SS.

914%; DS1, 687%; and DS2, 1202%. The reason why is the following: Closest-Toolcut-Direction tends to produce numerous extremely thin cells if the tool's direction gradually changes. To avoid this problem, Closest-Toolcut-Direction should be assigned lower priority than Furthest-From-Side (or Far-Enough-From-Side, explained later). Thus, I did not consider any rulesets that assign Closest-Toolcut-Direction the highest priority. As an illustration, Figure 6.12 shows a cellular view of two multidirectional heightmap representations of case D2 (see Figure 6.3); one was produced using ruleset (CTD, RS) (part (a)), and the other was produced using ruleset (FEFS-FFS, CTD, SPT, RS) (part (b)). Case D2 was produced via rotation of an angled tool 360 degrees and thus, the tool's direction gradually changes. Ruleset (FEFS-FFS, CTD, SPT, RS) clearly helps avoid skinny cells in this case (and it also decreases the subdivision count by a factor of 57).

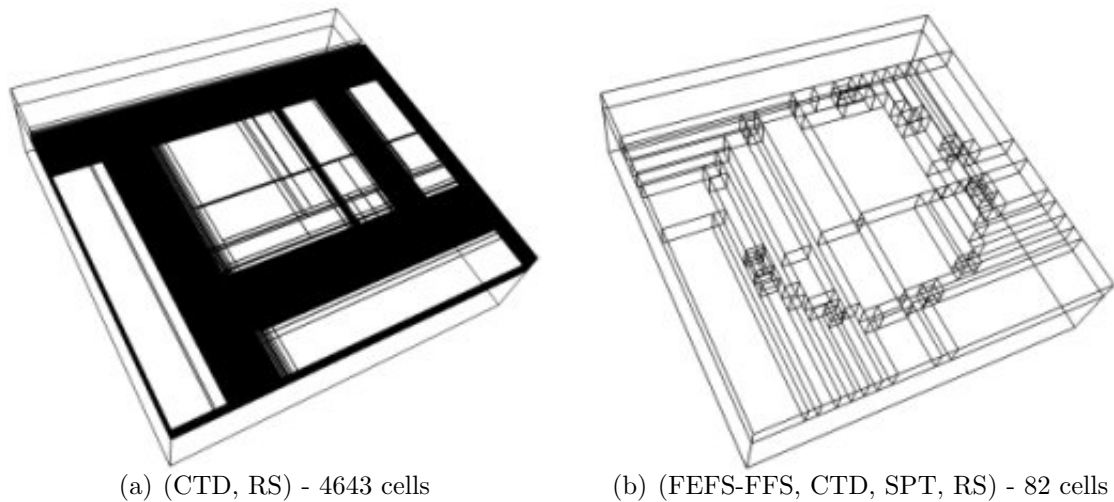


Figure 6.12: Two cellular representations of case D2 (see Figure 6.3).

Far-Enough-From-Side (FEFS) is meant to be a less restrictive version of Furthest-From-Side. Far-Enough-From-Side assigns high priority to each split plane that is further than a threshold distance R (the tool's radius) from both surrounding cell boundaries; low priority is assigned to all other split planes. All results obtained when Furthest-From-Side has the highest priority are roughly the same or identical for each test case, which suggests that the other rules are not being used to break ties very often. This is not the case when Far-Enough-From-Side is used in its place.

I also tried some rulesets that use both Furthest-From-Side and Far-Enough-From-Side, but not at the same time. The idea here is to use Furthest-From-Side whenever Far-Enough-From-Side fails to partition the split planes into two sets

(perhaps because all cell dimensions are less than $2R$), since close together split planes may not be avoided if this happens.

Overall, the data of Table 6.4 certainly does suggest the heuristics are helpful: every ruleset except for (CTD, RS) works better than Random-Selection. Here the six best performing rulesets are listed with their summation values:

- (1) (FEFS-FFS, CTD, SPT, RS), 72
- (2) (FEFS, CTD, SPT, RS), 74
- (3) (FEFS, SPT, CTD, RS), 74
- (4) (FEFS-FFS, SPT, RS), 94
- (5) (FEFS, SPT, RS), 80
- (6) (FEFS-FFS, SPT, CTD, RS), 96

From the above results I suggest the following:

- (1) Either FEFS or FEFS-FFS should be used and have the highest priority.
- (2) The SPT rule is also useful. The results are often much better when this rule is included.
- (3) The CTD rule might be useful. Sometimes it helps, and sometimes it does not; the results are too close to tell for sure.

When ruleset (FEFS-FFS, CTD, SPT, RS) was used, on average, 40.20% of the splits selected were indirect splits, 44.76% of the splits selected were direct splits and finally, 14.94% of the splits selected were sign conflict splits.

6.3 CPU and Memory Usage

This section experimentally addresses the CPU and memory usage of simulations with multidirectional heightmaps; the Section 6.1 test cases, and the simulation parameters of Table 6.1, were used once again. First I will consider CPU usage. All simulations utilized a 2.40 GHz Intel Core 2 Duo CPU with access to 2.0 GB of 333.9 MHz DDR2 RAM, and a NVIDIA GeForce 7600GT graphics card; but note that my implementation is not optimized for multiple cores. Table 6.5 tabulates simulation run time (in seconds) for rulesets (FEFS-FFS, CTD, SPT, RS), (FEFS-FFS, SPT, CTD, RS) and (RS). Run time results for each simulation pass as well as the total were computed; for reasons I note in Chapter 7, a second simulation pass is necessary to eliminate discontinuous seams. In addition, since sometimes only the final result of the simulation is required, run time results were computed with and without rendering enabled; however, there are no benefits to rendering the first pass so those numbers are not included in the table.

There are two important general trends to note from the results of Table 6.5:

- (1) For the most part, whichever ruleset achieves the lowest subdivision count also achieves the shortest run time. This fact is particularly apparent when comparing ruleset (FEFS-FFS, CTD, SPT, RS) or (FEFS-FFS, SPT, CTD, RS) with the RS rule.
- (2) The second pass run time is much slower when rendering is enabled. However, the rendering code was not optimized for speed and can likely be made to run much faster.

Table 6.6 tabulates numerous memory usage metrics for rulesets (FEFS-FFS, CTD, SPT, RS), (FEFS-FFS, SPT, CTD, RS) and (RS). The most obvious metric is the **Total**, which equals the total amount of memory used to store all aspects of the multidirectional heightmap data structure (kD-tree, heightmap grids, height/normal samples, etc.) in megabytes.

A few of the other metrics are intended to measure the memory efficiency of the data structure. First of all, the **Unique Height/Normal Samples** metric is the percentage of total memory occupied by “unique” height/normal samples (embedded and parent); larger values are better. The unique part of the definition refers to the fact that duplicate height/normal samples sometimes occur along a common boundary between cells (see Chapter 4).

The **Total as % of 3D Pointer Grid** metric also measures memory efficiency, but before this metric can be defined, the 3D pointer grid stock representation must be defined. As a multidirectional heightmap can be viewed as a more memory-compact version of a 3D pointer grid, but is likely more complex to implement and more computationally expensive, it would be good to know whether usage of a multidirectional heightmap is likely to significantly reduce total memory usage. That said, a *3D pointer grid* is a 3D array of pointers. Each grid entry is either null or points to a triplet of point/normal samples; each height/normal sample is calculated along a distinct cube edge, and these edges share a common point and are orthogonal.

The **Total as % of 3D Pointer Grid** metric equals the total memory as a percentage of the memory a 3D pointer grid would use to store the same design surface; smaller values are better. The metric is approximate because I did not actually machine any of the test cases using a 3D pointer grid stock representation. Thus, this metric does not measure height/normal pairs that are unused (a triplet is allocated even if only one sample is necessary).

Observe that if the relevant results of tables 6.2 and 6.6 are compared, it is apparent that as the subdivision count decreases, the height/normal samples percentage tends to increase, and the total memory as a percentage of a 3D pointer grid tends to decrease.

The **Embedded Height/Normal Samples** metric is the percentage of total memory occupied by embedded height/normal samples. The result of this metric is toolpath dependent, and is high for every test case considered in Table 6.6. For example, when ruleset (FEFS-FFS, CTD, SPT, RS) was used, results obtained from this metric range from 42.14% to 74.93%. However, the memory cost for embedded 1D heightmap usage is not as high as it appears. Since these extra surface samples significantly reduce approximation error, a lower sampling density can be used (see Section 6.4).

All other metrics measure the various uses of the memory a multidirectional heightmap uses organizing the sample data. These metrics are the following:

- **Null Grid Entries** - the percentage of total memory occupied by null sampling grid pointers.
- **Null Embedded Heightmaps** - the percentage of total memory occupied by null embedded heightmap pointers.

- **Unused Parent Height/Normal Samples** - the percentage of memory occupied by solid and empty parent heights/normals.
- **Duplicate Height/Normal Samples** - the percentage of total memory occupied by duplicate sample heights/normals.
- **Display List Management** - the percentage of total memory used for display list management.

The “waste” metric data of Table 6.6 suggests that null grid pointers are mostly responsible for wasted memory. For example, when ruleset (FEFS, CTD, SPT, RS) was used, results obtained from this metric were in the range of 5.08% to 32.96%. Thus, data structures for sparse arrays could possible help reduce memory consumption further.

FEFS-FFS, CTD, SPT, RS	CS	D1	D2	DS1	DS2	MA1	MA2	NA	A1	A2	A3
first pass, no rendering	95.14	12.09	6.70	21.80	15.33	10.52	8.64	0.80	1.63	9.61	3.72
second pass, no rendering	102.69	11.92	6.17	18.58	12.44	9.88	8.42	0.63	1.61	10.02	3.73
total, no rendering	197.83	24.02	12.88	40.38	27.77	20.39	17.06	1.42	3.23	19.63	7.45
second pass, rendering	486.23	68.34	35.88	136.36	99.06	25.22	18.16	3.56	3.72	35.91	4.69
total, rendering	589.09	81.89	43.52	161.36	117.15	36.52	27.07	4.48	5.42	45.85	8.41
FEFS-FFS, SPT, CTD, RS	CS	D1	D2	DS1	DS2	MA1	MA2	NA	A1	A2	A3
first pass, no rendering	95.56	12.59	7.53	20.48	14.34	10.47	8.67	0.77	1.63	9.64	3.72
second pass, no rendering	102.94	11.94	6.24	18.30	12.25	9.83	8.41	0.59	1.63	9.75	3.73
total, no rendering	198.50	24.53	13.77	38.78	26.59	20.30	17.08	1.36	3.25	19.39	7.45
second pass, rendering	456.50	73.48	43.69	132.94	92.20	24.89	17.34	3.53	3.77	33.73	4.75
total, rendering	562.28	87.84	52.39	157.14	108.89	35.88	26.29	4.44	5.49	43.79	8.48
RS	CS	D1	D2	DS1	DS2	MA1	MA2	NA	A1	A2	A3
first pass, no rendering	125.83	16.13	10.31	31.78	20.59	11.11	9.30	0.86	1.69	9.89	3.72
second pass, no rendering	147.86	12.91	7.34	23.34	15.05	10.06	8.67	0.64	1.66	10.48	3.73
total, no rendering	273.69	29.03	17.66	55.12	35.64	21.17	17.97	1.50	3.34	20.38	7.45
second pass, rendering	1122.08	83.83	67.59	241.36	156.27	35.05	27.53	4.30	4.28	45.14	4.64
total, rendering	1266.00	102.78	79.86	282.44	185.30	47.03	37.44	5.33	6.06	55.53	8.38

Table 6.5: Run time results (in seconds) for rulesets (FEFS, CTD, SPT, RS), (FEFS, SPT, CTD, RS) and (RS) when applied to the test cases of Section 6.1.

FEFS-FFS, CTD, SPT, RS		CS	D1	D2	DS1	DS2	MA1	MA2	NA	A1	A2	A3
Total (Mb)		17.80	16.59	5.64	14.74	11.39	2.87	2.61	0.59	0.54	3.11	0.55
Total (% of 3D pointer grid)		31.82	5.92	2.13	5.38	4.22	24.62	22.90	6.92	6.24	4.66	6.35
Unique Height/Normal Samples (% of Total)		87.85	89.41	75.80	73.77	71.19	84.82	86.76	57.93	72.43	75.47	80.77
Embedded Height/Normal Samples (% of Total)		64.74	70.05	42.14	44.31	43.17	73.72	74.93	36.07	43.46	44.61	53.30
Null Grid Entries (% of Total)		8.19	5.25	14.55	15.52	19.15	7.99	7.91	33.47	21.74	17.51	18.45
Null Embedded Heightmaps (% of Total)		1.04	1.15	1.80	1.69	1.66	0.83	0.66	2.12	0.19	2.22	0.30
Unused Parent Height/Normal Samples (% of Total)		1.05	1.11	1.04	1.17	1.28	2.38	1.96	1.84	0.98	0.74	1.23
Duplicate Height/Normal Samples (% of Total)		0.33	0.31	0.90	1.61	1.70	1.20	1.03	0.82	0.17	0.22	0.00
Display List Management (% of Total)		0.05	0.03	0.07	0.10	0.09	0.12	0.07	0.15	0.09	0.07	0.07
FEFS-FFS, SPT, CTD, RS		CS	D1	D2	DS1	DS2	MA1	MA2	NA	A1	A2	A3
Total (Mb)		17.75	16.69	5.82	14.08	10.95	2.80	2.54	0.57	0.54	3.20	0.55
Total (% of 3D pointer grid)		31.70	5.95	2.20	5.13	4.05	24.15	22.35	6.68	6.24	4.79	6.35
Unique Height/Normal Samples (% of Total)		88.22	89.20	74.20	76.97	73.85	85.56	87.89	59.47	72.43	73.60	80.77
Embedded Height/Normal Samples (% of Total)		64.30	70.50	43.60	47.45	44.24	71.68	74.92	36.97	43.46	61.26	53.30
Null Grid Entries (% of Total)		7.60	5.49	16.37	12.17	16.21	6.74	6.58	31.72	21.74	23.09	18.45
Null Embedded Heightmaps (% of Total)		1.05	1.11	1.66	1.61	1.64	1.00	0.73	2.18	0.19	1.00	0.30
Unused Parent Height/Normal Samples (% of Total)		0.88	1.05	1.08	1.01	1.07	2.13	1.64	1.67	0.98	0.90	1.23
Duplicate Height/Normal Samples (% of Total)		0.42	0.39	1.30	1.65	1.64	1.13	0.96	0.77	0.17	0.20	0.00
Display List Management (% of Total)		0.05	0.03	0.08	0.09	0.08	0.12	0.07	0.15	0.09	0.09	0.07
RS		CS	D1	D2	DS1	DS2	MA1	MA2	NA	A1	A2	A3
Total (Mb)		28.39	19.63	8.53	20.82	16.32	3.45	3.30	0.72	0.67	3.76	0.55
Total (% of 3D pointer grid)		46.33	6.98	3.22	7.55	6.02	28.90	28.34	8.43	7.76	5.62	6.35
Unique Height/Normal Samples (% of Total)		67.59	78.40	54.34	56.52	52.84	76.06	73.67	49.12	60.55	63.96	80.77
Embedded Height/Normal Samples (% of Total)		47.69	64.23	34.30	37.54	34.05	62.29	60.34	32.92	36.75	36.41	53.30
Null Grid Entries (% of Total)		27.49	16.59	38.36	35.10	39.75	15.51	19.17	43.69	33.95	28.85	18.45
Null Embedded Heightmaps (% of Total)		1.26	1.11	1.40	1.42	1.28	1.22	1.13	1.62	0.28	2.34	0.30
Unused Parent Height/Normal Samples (% of Total)		4.01	2.94	3.04	3.25	2.78	3.77	4.20	2.53	2.12	1.77	1.23
Duplicate Height/Normal Samples (% of Total)		1.73	0.73	2.21	2.72	2.10	1.64	1.62	1.15	0.35	0.20	0.00
Display List Management (% of Total)		0.25	0.10	0.26	0.29	0.23	0.17	0.15	0.21	0.14	0.11	0.07

Table 6.6: Memory usage for rulesets (FEFS, CTD, SPT, RS), (FEFS, SPT, CTD, RS) and (RS) when applied to the test cases of Section 6.1.

6.4 Maximum Error of Surface Approximation

A multidirectional heightmap is a triangular mesh approximation of a surface. The purpose of this section is to establish the relationship between maximum approximation error and stock density. Provided that my implementation is correct, we should expect the maximum approximation error to decrease as stock density increases. However, it would be difficult to establish that this relationship holds in all cases. Instead, similar to Israeli's work with regular heightmaps [10], the maximum approximation error for one particular design surface was calculated at numerous stock densities. The design surface used was created by rotating a sphere-tipped tool, angled by θ with respect to its rotational axis \vec{z} , 360° (Figure 6.13).

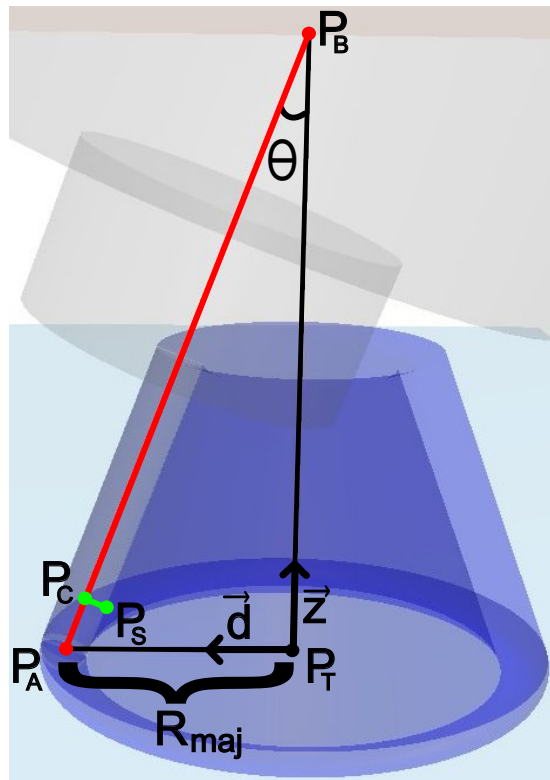


Figure 6.13: Closest error calculation for the error at a sample point P_S that approximates a design surface created by rotation of a θ -angled tool 360° .

The maximum approximation error for a design surface is the largest error that occurs between a sample point lying on a mesh triangle and the point on the design surface closest to that sample point (*closest point error metric*). For a design surface similar to the one I use here, Israeli found that the maximum approximation error for a triangle occurs at the centroid [10]. Thus, I only considered one sample point, the centroid, per triangle.

The equation necessary to compute the error for a triangle sample-point approximating the design surface of Figure 6.13 is somewhat complex. Thus, as a first step, I will derive an error equation for a simpler design surface created by a single drilling operation (see Figure 6.9). A simple bounding volume primitive called a capsule approximates the design surface in this case. A capsule is a sphere-capped cylinder defined by a radius R and a line segment (P_A, P_B) called the medial axis [7].

Let $P_C = (x_c, y_c, z_c)$ be the point on the medial axis closest to a triangle sample-point $P_S = (x_s, y_s, z_s)$. Then the error between P_S and the closest point on the design surface is given by the following equation:

$$error = |\sqrt{(x_s - x_c)^2 + (y_s - y_c)^2 + (z_s - z_c)^2} - R| \quad (6.1)$$

The following equations are used to compute point P_C :

$$\vec{m} = P_B - P_A \quad (6.2)$$

$$t = \frac{(P_s - P_A) \cdot \vec{m}}{\vec{m} \cdot \vec{m}} \quad (6.3)$$

$$P_c = \begin{cases} P_A + t\vec{m} & \text{if } t > 0 \\ P_A & \text{otherwise} \end{cases} \quad (6.4)$$

Note that only a cap centered at P_A is necessary in this work because no sample points will occur such that $t > 1$ in Equation 6.3.

Equation 6.1 can also be used to calculate the error for the Figure 6.13 design surface if an appropriately-placed capsule is first calculated. Specifically, the medial axis should lie in the plane formed by the sample point P_S and the tool's axis of rotation \vec{z} ; the capsule will intersect the design surface along a grazing curve if this condition holds. It is also important to note that the spherical tip of the tool sweeps out a torus with minor radius R , major radius R_{maj} , and center $P_T = (x_t, y_t, z_t)$, which coincides with the tool's axis of rotation \vec{z} . Thus, the medial axis (P_A, P_B) of the capsule required to calculate the error at a sample point P_S is given by the following set of equations:

$$\vec{d} = \frac{(x_s, y_s, 0) - (x_t, y_t, 0)}{\|(x_s, y_s, 0) - (x_t, y_t, 0)\|} \quad (6.5)$$

$$P_A = P_T + R_{maj}\vec{d} \quad (6.6)$$

$$P_B = P_T + (R_{maj}/\tan\theta)\vec{z} \quad (6.7)$$

The error data calculated for the Figure 6.13 design surface is given in Table 6.7; see Table 6.8 for the simulation parameters used. As was expected, approximation

error decreases as stock density increases. Also, if we compare a multidirectional heightmap composed of regular heightmaps with density d to another multidirectional heightmap composed of 3-maps with density $d/2$, the latter multidirectional heightmap has less approximation error and uses less memory (see Table 6.9).

There is a theoretical bound on maximum approximation error as measured by another metric referred to as vertical error [10]. This theoretical bound predicts factor of 4 improvement in maximum vertical error as the stock density doubles. Unfortunately, the theoretical bound only applies to regular heightmaps. However, it is interesting to note that as the stock density doubles, the maximum closest point error of a multidirectional heightmap has close to factor of 4 improvement (see Table 6.10), probably because the maximum vertical error is close in value to the maximum closest point error.

Stock Density	4	8	16	32	64	128
Heightmap Error	0.1586498	0.0891680	0.0601754	0.0383265	0.0500396	0.0086811
3-map Error	0.1082255	0.0278254	0.0069804	0.0020188	0.0005377	0.0001459

Table 6.7: Closest error data for the Figure 6.13 design surface.

Simulation Parameter	Value
stock dimensions (W, L, D)	(4 mm, 4 mm, 1 mm)
tool (and torus minor) radius (R)	0.15 mm
tool angle (θ)	25°
torus major radius (R_{maj})	1.69 mm
torus center (P_T)	(2 mm, 2 mm, 0.6748 mm)
grazing curve density	256
number of in-between steps	100

Table 6.8: Simulation parameters used when computing Table 6.7.

Heightmap $_{i+1,j+1}$ /3-map $_{i,j}$ Density	8/4	16/8	32/16	64/32	128/64
heightmap error/3-map error	0.82	2.16	5.49	24.77	16.07
heightmap memory/3-map memory	1.83	1.40	1.64	1.65	1.73

Table 6.9: Heightmaps versus 3-maps with half the density. Each cell $t_{i,j}$ of the first row is $d_{i+1,j+1}/d_{i,j}$ where d is a cell from Table 6.7.

Stock Density $_i$ /Stock Density $_{i+1}$	4/8	8/16	16/32	32/64	64/128
Heightmap Error	1.78	1.48	1.57	0.77	5.76
3-map Error	3.89	3.99	3.46	3.75	3.69

Table 6.10: The ratio between successive stock densities. Each cell $t_{i,j}$ of this table is $d_{i,j}/d_{i+1,j}$ where d is a cell from Table 6.7.

Chapter 7

Conclusion

This thesis has described in detail a new stock representation that can handle objects with overhangs, and its performance was experimentally verified. Here I will summarize the important results in this thesis and note the limitations of those results. Next, the relationship between multidirectional heightmaps and other stock representations will be established. The final section outlines several promising opportunities for optimization.

7.1 Summary and Contributions

In response to an overhang, an mdh-map recursively subdivides itself until each piece can be represented by an axis-aligned heightmap. To solve the when-to-split problem, I developed an additional data structure called a good-bad map. Regarding the where-to-split problem, I argued for the importance of minimizing the total subdivision count, and derived several split plane types and high-level heuristics with this goal in mind. My simulation results suggest that my techniques significantly help reduce memory and CPU usage.

Besides the challenge of deciding when and where to split, another challenge I encountered was the problem of rendering with continuous seams. This problem was solved using another new data structure called a 3-Way Heightmap (3-map). Since each cell of an mdh-map is represented using a 3-map, it can be viewed as a more memory-compact version of a 3D pointer grid, and can also be rendered using a variant of the well-known marching cubes algorithm.

The data obtained via my memory performance metrics suggest that the mdh-map is a memory efficient stock representation. The test cases considered suggest

that, when an mdh-map is used, simulations with reasonable accuracy and speed are possible. My error analysis data suggest that, as the stock density doubles, the maximum closest point error of an mdh-map has close to factor of 4 improvement. As this improvement comes from the 3-map data structure used for separating surface representation, a single 3-map data structure may be a good replacement for the heightmap as a functional surface representation scheme. In summary, the mdh-map successfully combines the accuracy of marching cubes with the memory efficiency of heightmaps.

7.2 Limitations

This work is not without its limitations. I did not simulate any industrial 5-axis toolpaths with overhangs. Thus, mdh-map performance in real practice is unknown. In particular, more splitting will be required when machining objects with many thin, non-axis-aligned features (e.g., the impeller of Figure 1.2(a)). Storing line segments rather than heights would minimize the number of splits in such cases. Another problem is that due to uniform tool sampling, cusps are not always represented well. A third problem is that large cells can be expensive to split. However, this expense will be subject to a reasonable bound if the mdh-map is sufficiently presplit into cubes (also, tree traversal can be accelerated using a 3D array in this case).

Since I did not implement 3-map resampling, a second simulation pass is necessary to eliminate discontinuous seams. Besides the obvious disadvantage of longer simulation duration, this approach does not guarantee accurate representation of the design surface for every in-between frame.

7.3 Other Simulation Techniques

Several researchers have represented the stock using uniformly-sampled, parallel line segments [9, 15, 21]; here I will refer to these stock representation schemes as *single direction sampling (SDS)* representations. SDS representations are very fast when GPU accelerated and are easy implement. In contrast, an mdh-map cannot be implement for modern GPUs, since dynamic memory allocation is not possible from shader (GPU) programs. Thus, better run time performance can likely be achieved if an SDS representation is used instead of an mdh-map. However, the mdh-map

representation is more accurate since three perpendicular sampling directions are used (long and skinny triangles are avoided). Also, the memory requirements of an mdh-map representation are not necessarily higher; such simulations can use a lower sampling density due to the lesser degree of approximation error.

The Octree is another stock representation that has been investigated [19, 11]. Roy and Xu [19] use a hybrid approach where each partial cell contains one face, one edge, or one vertex. An mdh-map will often experience higher splitting costs than a hybrid Octree would, but it will also require fewer splits (an mdh-map stores an unbounded number of triangles per cell). Also, note that the hybrid Octree uses a simplified tool-cube intersection calculation that assumes 3-axis machining (overhangs cannot be represented); calculating a tool's intersection with a cube is a complex task in general.

Karunakaran and Shringi [11] developed a “pure” Octree approach (i.e., no extra surface information is stored in leaf nodes). The pure Octree approach has significantly lower splitting costs than an mdh-map, but will often require a huge number of splits to obtain the same level of accuracy. Extremely tiny cubes are necessary to smoothly represent a curved surface with a pure Octree; smooth shading is not possible because surface normals are not stored.

A few simulation techniques that represent the stock using parallel 2D slices have been developed [6, 1]. Both approaches represent the slices using points interconnected by line segments. An mdh-map can also be thought of as a parallel-2D-slice-based stock representation. However, unlike the previous slice-based approaches, an mdh-map samples slices along three perpendicular directions rather than just one, and these slices are uniformly sampled (i.e., each slice is a marching squares curve). There are likely cases where an mdh-map is more accurate than these approaches, and vice versa. The supporting algorithms of the three stock representation schemes differ greatly. Thus, it is difficult to say which one would be faster.

Some solid modelling systems support simulation of toolpaths (including 5-axis toolpaths). This approach involves calculating the boolean subtraction of the tool (a combination of geometric shapes) from the stock at numerous positions along the toolpath. The solid modeller must record all boolean operations and transformation operations in a constructive solid geometry (CSG) tree [19]. Thus, this simulation method is much slower than any other approach (the simulation cost is $O(n^4)$ where n is the number in-between steps [6]), but is highly accurate when a small step size is used.

Another approach for 5-axis machining would be to represent the surface of the stock as a triangular mesh at the desired machined triangle size, and then to perform polyhedral CSG operations on the stock mesh by subtracting away the swept surface volume of the tool. Using a meshing library such as CGAL [8], the code for this simulation should be fairly compact. However, the computational costs should be similar to that of the solid modeling approach, and the memory costs would be significantly higher than my method. In particular, this mesh approach would have to store the x and y coordinates of the triangle vertices as well as the connectivity information of the mesh, both of which are stored implicitly with my method.

7.4 Optimizations

Two passes are currently necessary for simulations (that lack discontinuous seams). Two modifications are necessary to eliminate this multi-pass approach. Firstly, all heights, whether embedded or parent, should be measured relative to one of six stock bounding faces (i.e., the bounding face whose direction is the same). With this change in place, 3-map resampling would only require an adjustment to the pointer organization data of the surface samples; the surface sample data would not be modified. Thus, numerical problems would be avoided. Second, the solid segment construction algorithm outlined in Section 5.2.1 should be extended to exploit the extra information 3-maps contain.

Currently, simulations are significantly slower when in-between frame rendering is enabled. One reason why is referred to as the *partitioning problem*. To avoid the CPU-GPU data exchange bottleneck, the data of a 3-map is uniformly partitioned into cells by a 2D grid, and the sampling data within each cell is stored in a display list. It would be better to partition the sampling data with a 3D grid instead; if the difference between two heights is large, a large amount of data will be embedded between them.

A closely related rendering problem is referred to as the *rebuilding problem*. Since the triangular mesh (interpretation of an mdh-map) is only stored on the graphics card, all triangles must be recalculated when a display list is rebuilt. As the tool gradually passes through the cell of a display list, multiple rebuild operations will occur. To eliminate this problem without using a lot of extra memory, all triangles of such a cell should be stored in a specialized cache until the cell is no longer being cut. Another potential solution to the rebuild problem is to offload the

triangle creation stage to the GPU. However, this solution may not be possible due to the current limitations of geometry shaders (only a limited number of primitives can be generated).

There is no reason why a 3-map (or heightmap) cannot store both backfacing and frontfacing surface portions. Moreover, rendering and solid segment construction is probably not much harder with this extension in place. Since sign conflicts appear to be common, such a feature may significantly reduce the number of cell subdivisions required.

I developed a more general DSP series calculation algorithm (it does not assume a convex tool), but did not have time to implement it. Here I will briefly describe the main ideas of this algorithm. The algorithm takes as input several stacks of intersection points, one for each ray. Collectively, the top elements of the stacks compose the 1st level intersections, and below these are the second level intersections. The main step of the algorithm involves cutting away the largest functional piece of intersections possible. More precisely, first a split plane is placed at the 2nd level intersection closest to the first level intersections. Next, all intersections above this plane are popped from their respective stacks. Besides being more general, I suspect the formentioned algorithm may be less expensive computationally.

Finally, the mdh-map supporting algorithms can likely be adjusted to take advantage of additional CPU cores.

References

- [1] L. Perez-Vidal A. Puig and D. Tost. 3D simulation of tool machining. *Computers and Graphics*, 2003. 74
- [2] Karlo Apro. *Secrets of 5-Axis Machining*, chapter 7. Industrial Press, Inc., 2008. 1, 2
- [3] M.C. Leu D. Blackmore and L.P. Wang. Applications of flows and envelopes to NC machining. *Annals of the CIRP*, 41(1):493–496, 1992. 7
- [4] F. Ismail D. Roth, S. Bedi and S. Mann. Surfaces swept by a toroidal cutter during 5-axis machining. *Computer Aided Design*, 33:57–63, January 2001. 7
- [5] Jackie Neider Dave Shreiner, Mason Woo and Tom Davis. *OpenGL Programming Guide, Fifth Edition*. Addison-Wesley, 2005. 8
- [6] B. Kiatsrithanakorn P. Natasukon H. Ruei-Yun L. T. Son E. L. J. Bohez, N. T. H. Minh. The stencil buffer sweep plane algorithm. *Computer-aided Design*, 35:1129–1142, 2003. 74
- [7] Christer Ericson. *Real-Time Collision Detection*. Morgan Kaufmann, Elsevier Inc, 2005. 70
- [8] P. Hachenberger and L. Kettner. Computational geometry algorithms library: 3d boolean operations on nef polyhedra. http://www.cgal.org/Manual/latest/doc_html/cgal_manual/Nef_3/Chapter_main.html. 75
- [9] Tim Van Hook. Real-time shaded NC milling display. *Proceeding of the 13th annual conference on Computer graphics and interactive techniques*, pages 15–20, 1986. 73
- [10] Gilad Israeli. Software simulation of numerically controlled machining. Master’s thesis, University of Waterloo, 2006. 2, 5, 7, 8, 69, 71

- [11] K. P. Karunakaran and R. Shringi. Octree-to-brep conversion for volumetric NC simulation. *International Journal of Advanced Manufacturing Technology*, 32:116–131, 2007. 74
- [12] W. E. Lorensen and H. E. Cline. Marching cubes: A high resolution 3D surface construction algorithm. *Computer Graphics*, 21(4):163–169, 1987. 40, 44
- [13] S. Mann and S. Bedi. Generalization of the imprint method to general surfaces of revolution of NC machining. *Computer Aided Design*, 34(5):373–378, April 2002. 7
- [14] Martti Mantyla. *An Introduction To Solid Modeling*, chapter 5. Computer Science Press, 1988. 13
- [15] J. P. Menon and D. M. Robinson. Advanced NC verification via massively parallel raycasting. *Manufacturing Review*, 6(2):141–154, 1993. 73
- [16] G. M. Nielson and B. Hamann. The asymptotic decider: Resolving the ambiguity in marching cubes. *Proceedings of the 2nd conference on Visualization*, pages 83–91, 1991. 40, 41, 44
- [17] T. Ochotta and D. Saupe. Compression of point-based 3D models by shape-adaptive wavelet coding of multi-height fields. *Eurographics Symposium on Point-Based Graphics*, 2004. 8, 9
- [18] R. D. Toledo P Santos and M. Gattass. Solid height-map sets: modeling and visualization. *ACM Symposium on Solid and Physical Modeling*, pages 359–365, 2008. 8
- [19] U. Roy and Y. Xu. Computation of geometric model of a machined part from its NC machining program. *Computer Aided Design*, 31(6):401–411, 1999. 74
- [20] Peter Smid. *CNC Programming Handbook*, chapter 54. Industrial Press, Inc., 3rd edition, 2008. 1, 6
- [21] B. Tukora and T. Szalay. Fully GPU-based volume representation and material removal simulation of free-form object. *Advanced Research in Virtual and Rapid Prototyping - Proceedings of VR@P4*, 2010. 73