Accelerated, Collaborative & Extended BlobTree Modelling

by

Herbert Grasberger
Bakk. tech., Vienna University of Technology, 2006
Dipl. Ing., Vienna University of Technology, 2009

A Dissertation Submitted in Partial Fulfillment of the
Requirements for the Degree of

DOCTOR OF PHILOSOPHY

in the Department of Computer Science

Accelerated, Collaborative & Extended BlobTree Modelling

by

Herbert Grasberger
Bakk. tech., Vienna University of Technology, 2006
Dipl. Ing., Vienna University of Technology, 2009

Supervisory Committee

Dr. Brian Wyvill, Supervisor
(Department of Computer Science)

Dr. Melanie Tory, Departmental Member
(Department of Computer Science)

Dr. Ryan Budney, Outside Member
(Department of Mathematics and Statistics)

Supervisory Committee

Dr. Brian Wyvill, Supervisor
(Department of Computer Science)

Dr. Melanie Tory, Departmental Member
(Department of Computer Science)

Dr. Ryan Budney, Outside Member
(Department of Mathematics and Statistics)

ABSTRACT

BlobTree modelling has been used in several solid modelling packages to rapidly prototype models by making use of boolean and sketch-based modelling. Using these two techniques, a user can quickly create complex models as combinations of simple primitives and sketched objects. Because the *BlobTree* is based on continuous field-values, it offers a lot of possibilities to create and control smooth transitions between surfaces, something more complicated in other modelling approaches. In addition, the data required to describe a *BlobTree* is very compact. Despite these advantages, the *BlobTree* has not yet been integrated into state of the art industrial workflows to create models. This thesis identifies some shortcomings of the *BlobTree*, presents potential solutions to those problems and demonstrates an application that makes use of the *BlobTree*'s compact representation.

A main criticism is that the evaluation of a large *BlobTree* can be quite expensive, and, therefore, many applications are limited in the complexity of models that can be created interactively. This work presents an alternative way of traversing a *BlobTree* that lowers the time to calculate field-values by at least an order of magnitude. As a result, the limit of model complexity is raised for interactive modelling applications.

In some domains, certain models need more than one designer or engineer to be created. Often, several iterations of a model are shared between multiple participants

until it is finalized. Because the description of a *BlobTree* is very compact, it can be synchronized efficiently in a collaborative modelling environment. This work presents *CollabBlob*, an approach to collaborative modelling based on the *BlobTree*. *CollabBlob* is lock-free, and provides interactive feedback for all the participants, which helps with a fast iteration in the modelling process.

In order to extend the range of models that can be created within *CollabBlob*, two areas of *BlobTree* modelling are improved in the context of this thesis. CAD modelling often makes use of a feature called *filleting* to add additional surface features, which could be caused by a manufacturing process. Filleting in general creates smooth transitions between surfaces, something that the *BlobTree* can do with less mathematical complexity than approaches needed in Constructive Solid Geometry (CSG), in the case of fillets between primitives. However, little research has been done on the construction of fillets between surfaces of a single *BlobTree* primitive. This work outlines *Angle-Based Filleting* and the *Surface Fillet Curve*, two solutions to improve the specification of fillets in the *BlobTree*.

Sketch-based implicit modelling generates $3D$ shapes from $2D$ sketches by sampling the drawn shape and using the samples to create the implicit field via variational interpolation. Additional samples inside and outside the sketched shape are needed to generate a field compatible with *BlobTree* modelling and state of the art approaches use offset curves of the sketch to generate these samples. The approach presented in this work reduces the number of sample points, thus accelerating the interpolation time and improving the resulting implicit field.

Contents

List of Tables

List of Figures

ACKNOWLEDGEMENTS

I would like to thank:

Brian Wyvill, for all his support, guidance and encouragement for me to research what I was interested in.

GRAND and NSERC, for funding my research.

my parents, Elfriede and Herbert, for accepting my decision to pursue a PhD.

my wife, Marlies, for tolerating my moods, supporting me and proofreading this thesis.

everyone else who helped me further my research, for all our fruitful discussions. You know who you are.

DEDICATION

To my parents, for all their support.

Chapter 1

Introduction

1.1 Motivation

The *BlobTree* by [Wyvill et al., 1999] is a hierarchical approach to solid modelling based on Skeletal Implicit surfaces. Complex models can be created by combining Skeletal Implicit primitives or sketched shapes using a set of operators. These operators include all Boolean operations found in Constructive Solid Geometry (CSG) [Sabin, 1968] and more advanced and controllable blending operators to create smooth transitions between primitives. While Skeletal Implicit primitives provide a user with standard shapes to create a complex model, the incorporation of sketch-based modelling extends the modelling possibilities significantly. Consequently, it is very easy to prototype shapes of arbitrary topology. In addition to the modelling capabilities, a very important property of the *BlobTree* is that its description is very compact, resulting in a small memory footprint of a model, even when there are a lot of details. The *BlobTree* needs very little memory, and changes to the model can be described incrementally. For these reasons, the *BlobTree* has been used in research for many years, but due to some disadvantages the industry has not widely adopted it in their products.

Solid modelling based on CSG, on the other hand, is the standard methodology in the industry, despite some of the disadvantages resulting from its Boolean nature. Standard CSG only provides a limited set of operators to combine primitives, all of which are also present within the *BlobTree*-environment. Filleting, the process of "rounding" an edge, has to be realized using, for example, medial axis surfaces [Whited and Rossignac, 2009]. The *BlobTree*, in comparison, provides this function-

ality based on its continuous field-values and, as a result, additional surfaces are not necessary to create smooth transitions between objects. Therefore, the *BlobTree* is capable of recreating a standard CSG-tree using Boolean operators, whereas a CSG-tree cannot build the same model as a *BlobTree* including blend operators. Some *BlobTree* models can be built in CSG, using additional filleting surfaces, but at additional computation cost. Consequently, CSG can be considered inferior to *BlobTrees* in terms of fully integrated modelling capabilities, but it can be visualized faster than *BlobTrees*.

The work documented in this thesis is motivated by the desire to improve *BlobTree* modelling and to reduce the number of disadvantages compared to CSG which prevent its usage in state-of-the-art applications. Three properties of *BlobTree* modelling are addressed, which can be divided into three areas.

Firstly, the *BlobTree* is known to be slower at rendering (using polygonization and ray-tracing) than traditional modelling methods. With the improvement in computing power in recent years, the *BlobTree* can now be rendered at interactive frame rates, depending on the overall scene complexity (e.g. [Shirazian et al., 2012]). So far, the *BlobTree* has not really made use of the raw computing power GPUs provide, due to the complexity of the algorithms and the underlying structure of the *BlobTree*. In order to provide an OpenCL-based, interactive working experience for the user, even with models of high complexity, the algorithms traversing the *BlobTree* and calculating field-values have to be adapted to a GPU's unique hardware constraints.

Secondly, the underlying mathematical description of Skeletal Implicit Surfaces can make it harder for users to build the object they want. While user interfaces that hide the math from the usere have been created (e.g. [Schmidt et al., 2005a]), the advantage that the *BlobTree* is a very compact description of a model [Bloomenthal, 1997] has not yet been utilized fully. Collaborative user interfaces require instant transmission of model changes and interactive user feedback, something a compact model description can provide.

Thirdly, the *BlobTree* is known to create "blobby" shapes due to a lack of precise modelling tools. For example, a lot of different methods for blending exist, but most of them are hard to control and create undesired surface changes, e.g. blending at distance. These problems were solved when blend methods got revisited by [Bernhardt et al., 2010] and improved by [Gourmel et al., 2013]. Blending, however, only solves the first half, smooth transition between objects, of the so called *filleting* problem. The second half of this problem relates to smooth tran-

sitions between surfaces within an object. A first attempt at solving this problem by [Grasberger et al., 2010] does not create fillets of the required continuity. Sketch-based modelling on the other hand extends the *BlobTree* with non-standard shapes that are based on hand drawings. It is important for precise modelling that the implicit field of such shapes is well-formed, something achieved by post-processing of the drawn shape by [Schmidt and Wyvill, 2005b].

This work improves each of the aforementioned issues in *BlobTree* modelling. The next few sections outline the contributions of this thesis and show how *BlobTree* modelling can be *accelerated*, how it can be used in a *collaborative* modelling environment(*CollabBlob*) and how the *BlobTree* can be *extended* to give the user better tools to create models.

1.2 Contributions

1.2.1 Accelerated *BlobTree* Traversal

Most of the previous work on accelerating *BlobTree* visualization focussed on the optimization of the rendering and/or mesh generation algorithm. Methods to accelerate those algorithms are based on the reduction of the number of *BlobTree* traversals that calculate the field values, normals and colours at certain points in space.

Little work, however, has been done on speeding up a single tree traversal to calculate such a field-value. There has been research on accelerating the tree traversal of Constructive Solid Geometry (CSG) trees (e.g. [Hable and Rossignac, 2005], [Romeiro et al., 2006] and [Hable and Rossignac, 2007]), but, only parts of these approaches can be used when traversing *BlobTrees*. This work shows which parts of the CSG traversal improvements can be applied to the *BlobTree* and how to make *BlobTree* traversal more efficient.

On modern hardware, memory bandwidth and lack of locality are often the limiting factors of algorithm performance. By changing the storage layout of the *BlobTree*, memory transfers during the tree traversal can be reduced and the traversal performance improved.

This work enables the execution of many *BlobTree* traversals in parallel on modern GPUs and CPUs. The traversal algorithm presented in this thesis makes use of the floating point vector architecture during the field-value calculation, so no processing power is wasted. While OpenCL is not available on current mobile devices, the

optimized traversal algorithm still shows significant performance improvements so that *BlobTree* modelling is possible on modern mobile devices.

1.2.2 Collaborative *BlobTree* Modelling Environment

Since the *BlobTree* is a very compact representation of a model, even when the actual model is fairly complex, it lends itself to be used in an application where data needs to be transmitted. Even though modern networks are increasing in speed, having a small representation of a complex model can be an advantage when trying to synchronize this model between devices connected via wireless networks. An application can utilize the small memory footprint of the *BlobTree* to synchronize models between several users in a collaborative modelling environment. As a result, many users can work on a single model at the same time, and there is no need to lock parts of the model during editing. Moreover, every participant involved receives immediate feedback on what every other user is attempting to do, which is rendered interactively by making use of the accelerated *BlobTree* traversal described above.

CollabBlob makes use of WebSockets [W3C, 2013] to transmit the data between the participants and shows that a simple, time-stamp-based mechanism is sufficient to synchronize the work of many users on one single model.

The approach presented separates the communication between the devices involved in several layers of varying importance. Communication modifying the model is separated from communication providing interactive feedback on the actions of other users. Depending on the desired granularity, a user can even choose to use the stored communication messages, in order to re-play or recreate the model at any time. Due to the minimal memory usage of the *BlobTree*, storing the whole modelling history needs a lot less memory than a comparable approach transmitting mesh changes.

1.2.3 Extended *BlobTree* Modelling

Once many people can work together, the complexity of the resulting models increases. In order to provide the users with greater control over the resulting shapes, two areas of *BlobTree* modelling are extended by this work.

Filleting

The creation of controllable smooth transitions between surfaces (sometimes also referred to as the rounding of a corner) is known as *filleting*. While some modelling packages refer to the result of a filleting operation as *bevel* or *chamfer*, fillet is the more general term and used throughout this work. Extensive work has been done on filleting in the context of traditional CSG modelling (e.g. [Middleditch and Sears, 1985], [Adzhiev et al., 1999] and [Pasko et al., 2005]), where creating smooth transitions between surfaces of a model is desired for example for artistic reasons or to mimic the result of a manufacturing process. In many cases a fillet is built by inserting a separate surface, often being of high polynomial order.

Unlike more conventional solid modelling methods, creating fillets in the *BlobTree* does not require the placement of additional objects or surfaces. Fillets between objects resulting from blend operators that provide a lot of control to the user on the resulting shape has been heavily researched by [Barthe et al., 2004], [Bernhardt et al., 2010] and [Gourmel et al., 2013]. On the other hand, little work has been done on fillets between surfaces within one *BlobTree* primitive.

This thesis presents *Angle-Based Filleting*, a solution to this problem based on an existing controllable blend operator by [Gourmel et al., 2013] and that allows the creation of fillets within *BlobTree* primitives. These fillets extend the modelling capabilities so that a designer has more control over the final object.

Warping a *BlobTree*, as presented by [Sugihara et al., 2010], has been a large improvement in the context of sketch-based modelling. This work proposes the *Surface Fillet Curve* approach, an extension to the *BlobTree* that makes it possible to create arbitrary fillets on any *BlobTree* surface by drawing on top of the model.

Sketching

Sketch-based modelling is a more recent addition to *BlobTree* modelling and enables users to quickly prototype shapes. The main premise of sketch-based modelling in the context of the *BlobTree* is that drawn shapes need to be sampled at discreet points. The the samples are then used as control points of a variational interpolation method. For the implicit field to be generated properly, additional control point samples need to be generated inside and outside the drawn shapes. While previous approaches (e.g [Schmidt and Wyvill, 2005b]) used offset curves for these additional control points, this new improved method reduces the number of points to interpolate by altering

the placement of the interior sample points. This approach also positively affects the generated implicit field because there are no regions of field compression where the change in the field values doesn't correlate with the change in distance to the sketched shape.

1.3 Combining the Contributions

To summarize, the four contributions of this thesis are:

- A more efficient *BlobTree* traversal algorithm (see Chapter 3)
- A method to use the *BlobTree* in a collaborative modelling environment (Chapter 4)
- A method to specify fillets on *BlobTree* primitives (see Chapter 5) and
- An improvement to the algorithm generating implicit field from drawn shapes (Chapter 6).

By combining these main contributions a *BlobTree* modelling application that runs on both MacOS and iOS was created. Mobile devices running iOS have limited processing power compared to laptops or desktop devices that can offload intensive work to the CPU. However, the efficient traversal algorithm also improves running times on mobile device CPUs. This makes it possible to include mobile devices into the collaborative *BlobTree* modelling system.

In addition, being able to add surface detail through filleting without actually increasing the *BlobTree* complexity of a model, improves the performance on mobile devices. Without the capabilities to control filleting, fillets would have to be created by a number of additional primitives that are added and/or subtracted from the *BlobTree*.

Finally, sketch based modelling lends itself to usage on touch screen devices. The improved method to generate *BlobTrees* from sketched shapes needs fewer control points to generate a better implicit field. While this improves visualization times on mobile devices, every model benefits from the improved implicit field.

1.4 Outline

This work is structured as follows: In Chapter 2 the state of the art in *BlobTree* modelling, including recent developments, such as better blend operators and the inclusion of sketch-based modelling approaches, is discussed. In Chapter 3, a unique *BlobTree* traversal algorithm that provides significant speed-ups compared to previous approaches is presented. Chapter 4 discusses a collaborative modelling environment based on the *BlobTree*. The extension of the *BlobTree* to have more control on fillets is shown in Chapter 5, while the improvements in sketch-based modelling are described in Chapter 6. An overall conclusion of this work is given in Chapter 7.

Chapter 2

The *BlobTree*

This chapter gives an in-depth look at the modelling paradigm known as the *BlobTree*. As a hierarchical solid modelling technique it is very similar to Constructive Solid Geometry (CSG) which is briefly introduced below. The *BlobTree* is based on *Skeletal Implicit Surfaces* and *Sketched Objects*, which can be combined using a large set of *Blend Operators*. Additional sketch-based methods for *Warping* are also part of the *BlobTree*'s modelling features. The models created using the *BlobTree* can be visualized using mesh-generation or ray-tracing techniques.

2.1 Constructive Solid Geometry (CSG)

Constructive Solid Geometry (CSG), as first described by [Sabin, 1968] and later applied to implicit modelling by [Ricci, 1973], is often considered a solid modelling technique that the *BlobTree* by [Wyvill et al., 1999] is related to, while in reality the *BlobTree* is a more general and powerful approach. In CSG, the basic building blocks are simple primitive objects that partition space into *inside* and *outside* regions, with the surface lying at their borders. Because of this binary classification of surfaces, the evaluation of CSG models can be reduced to Boolean *true/false* (e.g. [Rossignac, 1999]). More complex models are created in CSG by combining these simple primitives (e.g. sphere, cube, cylinder, etc.) using the Boolean operators (notation as per [Rossignac, 2012]) :

- union, using Boolean **or** ("+")
- intersection, using Boolean **and** ("•")

- difference, using a combination of Boolean **and** and the **complement** ("•!").

This results in a solid model tree structure, with the primitives being the leaves of the tree, and the operators the interior nodes. CSG models can be visualized with a variety of approaches, of which ray-tracing and mesh-generation (polygonization) are the most common. Other approaches like [Hable and Rossignac, 2005] rely on depth-peeling or use Graphics Processing Units (GPUs) (e.g. [Hanniel and Haller, 2011] and [Romeiro et al., 2006]) to improve visualization times on modern hardware.

Despite appearing to be a very simple and restricted modelling environment, CSG has become a de-facto standard for solid modelling in the industry and much work has been done to accelerate visualization (e.g. [Rossignac, 1999], [Hable and Rossignac, 2007] and [Rossignac, 2012]) and extend the resulting surfaces, such as [Elber and Cohen, 1997], [Elber, 2005] and [Whited and Rossignac, 2009].

2.2 Skeletal Implicit Surfaces

BlobTree modelling from a user's perspective is very close to CSG in that models are built by combining primitives using operators. While both approaches fall into the category of solid modelling, the basic building blocks of the *BlobTree* are based on continuous field-values as opposed to Booleans in CSG. This fundamental difference allows the *BlobTree* to use additional combination operators, resulting in a wider variety of transitions between surfaces, some of which are emulated in CSG using mathematically complex surface descriptions as shown in [Whited and Rossignac, 2009].

Compared to the binary space classification, the basic building block of the *BlobTrees* parametric modelling approach are so called *Skeletal Implicit Surfaces* described in [Bloomenthal, 1997]. These Skeletal Implicit Surfaces can be found in a variety of available modelling packages, such as *BlobTree.net* [de Groot, 2008] or *ShapeShop* [Schmidt et al., 2005b], which are often used to prototype shapes of arbitrary topology (see [Bloomenthal, 1997]). In general these works conclude that the use of skeletal primitives can lead to a simple and intuitive user modelling methodology.

The basic building block of a skeletal primitive is a skeleton S. Usually the skeleton itself is a very simple shape such as a point or a line, but also more complex skeletons can be used as described by [Grasberger et al., 2010]. Figure 2.1 shows the most common used *BlobTree* primitives.

To create a skeletal primitive, the distance-field has to be computed as described

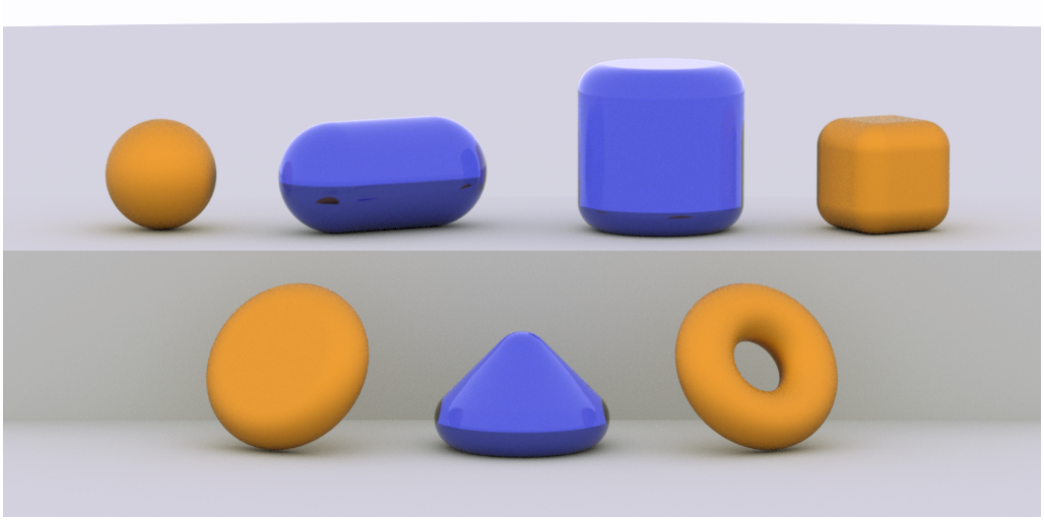


Figure 2.1: The most common used *BlobTree* skeletal primitives. (Image from [Grasberger, 2009])

by [Barbier and Galin, 2004]. This is done by calculating the distance to the skeleton for each point in the volume encapsulating the final shape. The distance function is defined as $d_S : \mathbb{R}^3 \to \mathbb{R}$. As a result, the distance field is a volume of scalar values, which is not bounded as a distance can be infinitely large. In most cases, the euclidean distance to the skeleton is used, creating a smooth distance field surrounding every skeleton. However some approaches (partly) rely on other distance metrics to create non-continuous distance fields in order to introduce sharp surface features. Having a non-continuous distance field can, however, result in significant problems when combining objects, as will be discussed in Section 2.4.

In the next step, the distance field d_S has to be modified by a filter fall-off function (often called field function) (see [Shirley and Marschner, 2009] for an in-depth discussion) to bound the field to a finite range. This field function is defined as $g : \mathbb{R} \to \mathbb{R}$, and the final skeletal implicit primitive is formed by applying this function to the given distance d_s. Usually the function maps the distances from the range $[0, r]$ to $[1, 0]$, where points having field values of 1 are on the skeleton and points having field values of 0 have distances $d \geq r$. In most cases r is chosen to be 1.
In general, the implicit function of one skeletal primitive is:

$$f(p) = g(d_S(p))$$

The Wyvill field function [Shirley and Marschner, 2009], was developed as a simpli-

fication of the original Soft Objects field function [Wyvill et al., 1986], which is more complex to compute and not C^2 continuous. It maps a distance d to the field value $g(d)$ using the following formula

$$g(d) = \left(1 - \frac{d^2}{r^2}\right)^3$$

(see Figure 2.2 for a plot of the function). In this formula, r is a constant value that states the distance where the field value equals zero. The main advantage of this field function is that it is C^2 continuous. A discussion of different field functions appears in [Shirley and Marschner, 2009].

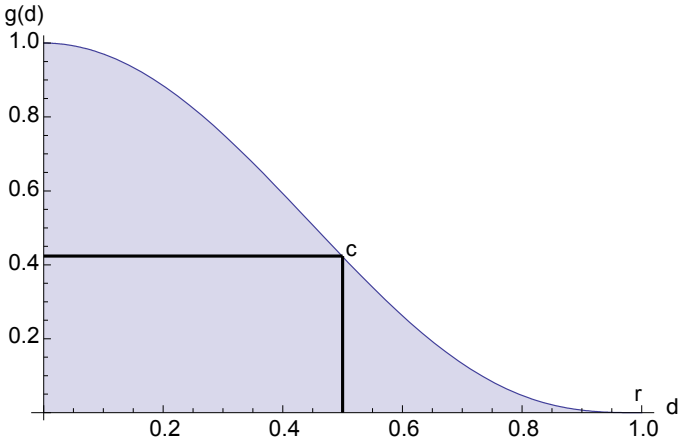


Figure 2.2: The commonly used Wyvill field function. The black line marks the iso value $c = 27/64$ to place the surface at $d = 0.5$.

After applying the field function to the distance field the resulting field is called the potential field. The surface is defined as the locus of points that have a field-value equal to the chosen iso-value. By defining an *iso*-value c, it is possible to construct the surface of the shape and classify the surrounding space into points inside ($f(p) > c$) and outside ($f(p) < c$) the surface. The chosen iso-value depends on the exact form of $G(d)$, for example $c = 0.5$ ([Bloomenthal, 1997]). Since the Wyvill field-function is not symmetrical ($G(0.5) \neq 0.5$), it can be an advantage to choose $c = G^{-1}(0.5)$ as the iso-value c. With this chosen iso-value, the resulting surface lies at distance $d = 0.5$ to the skeleton, which helps with precise modelling.

A Skeletal Implicit Primitive is formed by the surface along the given iso-value within the continuous implicit field. This differs from the Boolean inside and outside regions that define CSG primitives. These field-values decrease with the distance to

the original skeleton, a property that is the main advantage over CSG. Blends between surfaces can be created with little computational cost, compared to the expensive solutions found in standard CSG which cause problems in some blend situations (e.g. a blend of blend comes to mind as described in [Middleditch and Sears, 1985]). In CSG, the Boolean inside/outside classification is done at every primitive and passed to each operator, whereas the *BlobTree* uses the continuous field-values until the root node of the tree is reached. Only then the surface is created based on the iso-value c.

2.3 Sketching using the *BlobTree*

Traditional sketch-based modelling differs from CSG based modelling, since it does not combine primitives together using boolean logic. Moreover, it does not rely on primitives at all, but requires the user to draw (parts of) the models. As a drawing is only two dimensional, there are several ways to interpret the contour the user has drawn:

- surface of revolution
- extrusion along a spline
- inflation

Some more advanced sketching approaches, such as demonstrated by *iLoveSketch* [Bae et al., 2008], even allow for sketching in multiple planes, but only create curve data.

Several approaches to sketch based modelling exist, some of them working with meshes [Igarashi et al., 2007], others using implicit surfaces [O'Brien and Turk, 2002], with both approaches having advantages and disadvantages. Mesh-based methods usually have difficulties supporting blending between objects, which, however, is very easy to do using implicit surfaces. A *BlobTree*-based sketching system, such as *ShapeShop* [Schmidt and Wyvill, 2005a], on the other hand supports all operations of the *BlobTree*, including blending, warping and tapering, even precise contact modelling (PCM), as shown by [Cani, 1993]. This is done by creating *BlobTree* compatible shapes, using the three interpretations of a curve outlines above, that can be integrated as primitives into the exiting *BlobTree*. The interface of ShapeShop is based on the same paradigms as *Teddy* [Igarashi et al., 2007], which allows the cre-

ation of complex sketched shapes that can be combined with others using the *BlobTree* operations.

The workflow of such a sketch based modelling interface can be divided into the following stages, which will be described below:

1. query hand drawn shape
2. compute flattened curve from drawings
3. generate field values for the resulting shape
4. save this 2D field and modify it to create a 3D volume.

Since the complete dataset of the drawn shape can be fuzzy due to input lag and human motion, the data has to be filtered. Usually, it is sufficient to filter the positions to reduce high frequency motion, but in some cases, it is desired to detect overlapping curves or a beginning and end that are parallel to some extent. In the latter case, only connecting the start and end point to create a closed shape, which is necessary to properly create the field values, can lead to undesired shapes as shown by [Schmidt et al., 2005a].

In the sketch-based modelling method by [Schmidt et al., 2005a], the shape sketched by the user is sampled at a lower resolution, and an implicit approximation is created from the sample points. This is done by fitting a thin-plate spline to the sampled points using variational interpolation as demonstrated by [Turk and O'Brien, 1999a]. A continuous $2D$ scalar field is created from several samples (p_i, v_i), where p_i describes the position of the sample and v_i its field-value.

If the sample points are on the control polygon the field-value is the set iso-value. The thin-plate spline used to create the variational implicit field $f(p)$ is defined in terms of these points weighted by corresponding coefficients w_i combined with a polynomial $P(p) = c_1 p_x + c_2 p_y + c_3$.

$$f(p) = \sum_{i \in N} w_i (\|p - p_i\|)^2 ln(\|p - p_i\|) + P(p) \tag{2.1}$$

One advantage of creating the base shape using variational interpolation is that the resulting implicit field is C^2 continuous, a property needed when the shape is involved in several blending operations [Barthe et al., 2004].

A thin plate spline, created by sampling the sketched shape and setting the v_i values to the desired iso-value, will return the iso-value for every interpolation result. In order to create a thin-plate spline which returns values in the range of $[0, 1]$ (and to avoid the interpolation-overshoot), additional points p_i with their field-values v_i have to be computed, both inside and outside the base shape. Chapter 6 presents methods to determine the placement of these additional sample points. The field-values v_i assigned to these sample points are proportional to the displacement along the normal. This can lead to problems in cases involving concave polygons, since the displaced polygon could intersect at a polygon notch (a non-convex feature [Lien and Amato, 2006]), thus producing sample points with wrong polygon distances. In addition, it is possible that the inside displaced points actually lie outside the original polygon. A solution to this problem is presented in Chapter 6.

2.4 Blend Operators

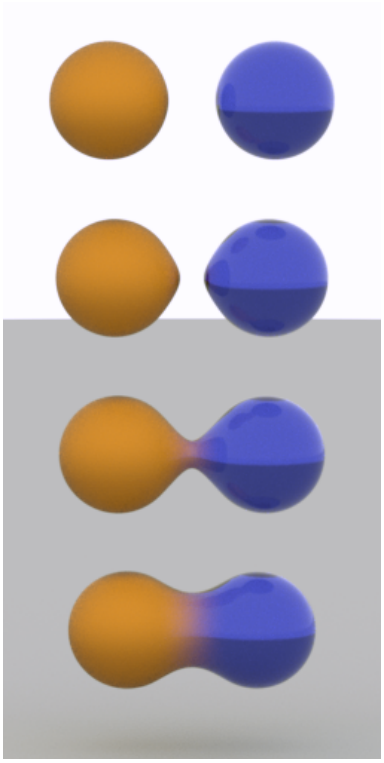


Figure 2.3: The summation blend between two spheres using the Wyvill Field function. (Image from [Grasberger, 2009])

In *BlobTree* modelling, primitives are combined to form complex models, similar

to CSG. Instead of doing Boolean operations on the binary space classification found in CSG primitives, *BlobTree* operators use the continuous field-values f generated by the Skeletal Implicit Primitives. This makes it possible to go beyond the classical Boolean operators, and define general *blend operators* that e.g. create smooth transitions between shapes, in addition to the operators found in CSG (as per [Ricci, 1973]):

- union: the max of a set of n field-values f_n
- intersection: the min of a set of n field-values f_n
- difference: defined as a binary operator $min(f_1, 1 - f_2)$.

The most common blend operator that creates a smooth transition between several values is called the *summation blend* [Bloomenthal, 1997]

$$f_R(p) = \sum_{n \in N} f_n(p)$$

where the resulting field-value at a point p in space $f_R(p)$ is the sum of the field-values of all the objects involved.

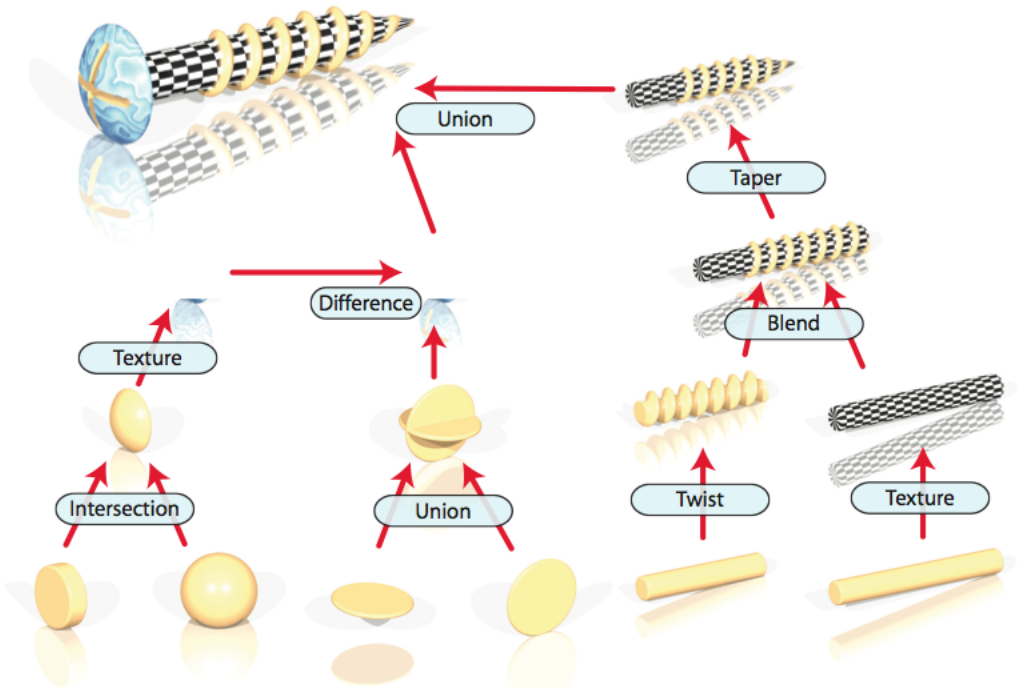


Figure 2.4: An example of a *BlobTree* with all the transformation nodes shown. Image courtesy of *Erwin de Groot* [de Groot, 2008]

Other complex operators, such as those described in [Barthe et al., 2003] and [Barthe et al., 2004], or the blending functions that are based on R-functions (e.g. [Shapiro, 1994] and [Pasko et al., 1995]), allow for a fine control on the resulting blend shape. By using them, it is possible to create complex blended shapes similar to the ones proposed for CSG in [Elber, 2005].

Figure 2.3 shows a blend between pairs of spheres with decreasing distance. It can be seen, that the two objects start to bulge towards each other at a certain distance before actually touching each other. This may not be desired and has been a problem in *BlobTree*-modelling for a long time. It was solved by extending the simple sum to a more complex and controllable operator that also takes the gradients of the objects at each point into account when calculating the new field-value after the operator by [Gourmel et al., 2013].

2.5 WarpCurves

WarpCurves [Sugihara et al., 2010] is the current state-of-the-art approach to Free-form Deformation (FFD) [Sederberg and Parry, 1986], first introduced to the *BlobTree* in [Sugihara et al., 2008]. Whereas traditional modelling can only influence the object surfaces by placing primitives and combining these primitives with various operators, FFD enables the user to alter any object without using the traditional CSG approach. Instead, the user can "grab" a part of an object and modify it by hand.

In the first approach by [Sugihara et al., 2008], surface features are selected by drawing a small line on them. The line is then used as a "hook" to move the surface around. In order to comply with a very popular *peeling* [Igarashi et al., 2005] technique of deforming surfaces, the farther the curve is moved, the bigger the *Region of Influence* (ROI) of the deformation is. Two underlying techniques are used to deform the model, and the whole implicit field:

- coarse voxelization to interactively approximate the deformation of the model
- variational warping, a more complex deformation algorithm that properly deforms the whole implicit field.

In the voxelization approach, a coarse voxelgrid is placed around the object to be deformed. When the deformation curve is drawn on the underlying shape, the voxels closest to the curve are calculated. Once the curve is moved, the closest

voxels are deformed accordingly (and the deformation is propagated to neighbouring voxels). As a result, a deformation vector for every voxel can be calculated, and used to approximate the exact deformation when regenerating the mesh of the deformed model. This, however, creates discontinuities in the resulting implicit field, as the field is defined outside the surface boundaries. The voxel grid is only built as long as voxels intersect the *BlobTrees*. Creating the proper deformation field is mathematically more complex and takes longer to compute, thus the approach using the voxel grid allows for faster interactive feedback, while the final deformation is calculated.

A proper deformation of the whole implicit field is very important for any technique to work properly within the *BlobTree* modelling system, as any deformed surface can be used in blending situations later on. For these blending situations, a proper continuous field is needed, as otherwise cracks and undefined surface behaviour could occur. Variational warping is the basis for the WarpCurves approach, which largely influenced the work on the Straight-Edge-Warp technique presented later and, for this reason, it is explained in more detail below.

2.5.1 Variational Warping

Variational Warping is a technique based on variational implicit surfaces, as used in the sketch based approach described in Section 2.3. For the sketching case, the outline of the sketched shape is sampled, and additional sample points are taken outside and inside the shape. Each of these points are assigned a weight based on their location:

- samples on the shape have weight corresponding to the iso-value $v = c$
- samples inside have weights $1 > v > c$
- sample outside have weights $0 < v < c$.

The samples are then used to fit a thin-plate spline to the sample points using variational interpolation [Turk and O'Brien, 1999a], which then creates a 2D field.

Variational warping extends the above $\mathbb{R}^2 \rightarrow \mathbb{R}$ approach to work in $\mathbb{R}^3 \rightarrow \mathbb{R}^3$, where $3D$ points in space interpolate $3D$ displacement vectors. The formula to achieve this approach as given by [Sugihara et al., 2008] is very similar to the $2D$ case presented earlier:

$$d_f(p) = \sum_{i \in N} w_i (\|p - p_i\|)^3 + P(p) \tag{2.2}$$

where

$$P(p) = c_1 p_x + c_2 p_y + c_3 p_z + c_4 \tag{2.3}$$

and w_i, p_i, p and $c_i \in \mathbb{R}^3$.

In order to calculate the weights w_i and the coefficient c_1, c_2, c_3 and c_4, the linear system has to be solved at any known solution. These known solution are taken from the grid points of the voxel grid, with their corresponding displacement vectors used as v_i. Once the coefficients of the polynomial and the general weights are calculated, this function maps any $p \in \mathbb{R}^3$ to a corresponding displacement vector, with C^2 continuity. This variational warp can be used as a unary node in the *BlobTree* hierarchy, placed above any node, similar to any affine transformation. In order to evaluate the variational warp operator at a point p, the displacement vector $d_f(p)$ has to be calculated. Then, p has to be modified by $d_f(p)$ before the result of this modification is then passed on to the child node of the variational warp for further calculations.

Since this node cannot be represented by a $4x4$ matrix, it prevents an important optimization method, which is normally used in many applications to accelerate the evaluations. Affine invariant transformations, represented using $4x4$ matrices on arbitrary locations within the *BlobTree*, can be pushed to the leaf nodes in a preprocessing step. As a result the aggregated transformations are available at every leaf node, avoiding multiple matrix - matrix multiplications when the tree needs to be traversed multiple times to calculate field-values. In the case of the variational warp, this is not possible, but a potential solution to this problem is outlined in Section 3.5.

2.5.2 WarpCurve User Interface

The Free Form Deformation (FFD) approach described in [Sugihara et al., 2008] differs from the WarpCurves approach [Sugihara et al., 2010] in the interface provided to the user to describe the deformation. In WarpCurves, the shape of the deformation is controlled by a curve drawn on the surface. This curve can have any shape along the model's surface, and provides a certain number of control points to the user. Compared to only having one handle to control the deformation field, WarpCurves are able to use any of these control points as deformation handles. Each of these control points can be moved in different directions, with the restriction, that neighbouring control points can be influenced depending on the length of the deformation added. Using the Wyvill function to control influence fall-off [Shirley and Marschner, 2009]

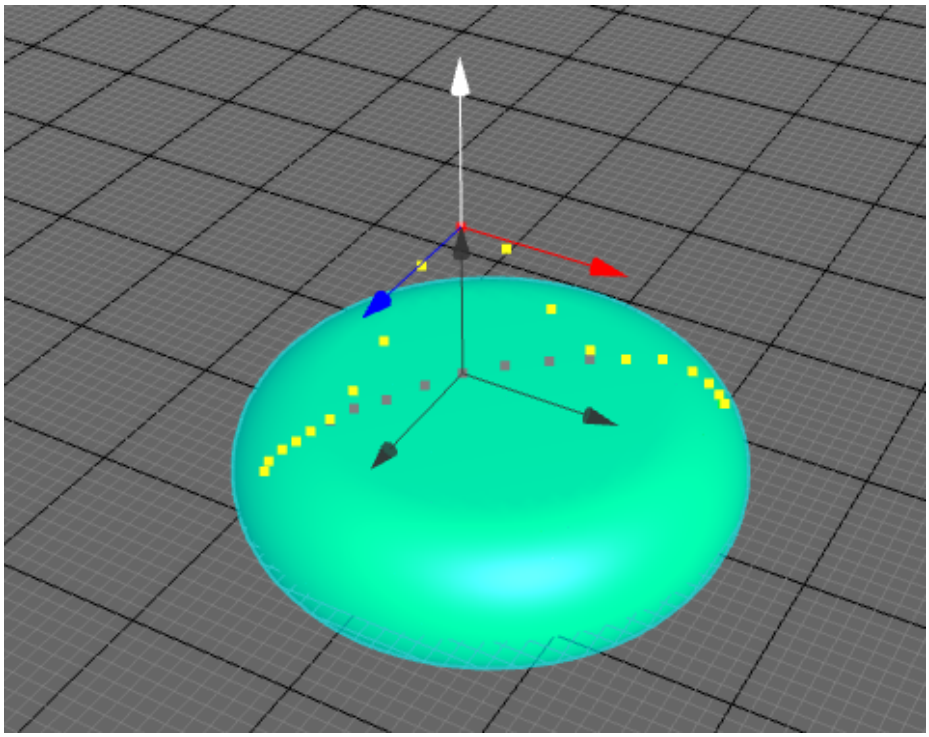


Figure 2.5: Propagation of the warp vector to neighbouring control points.

$g(d) = (1 - \frac{d^2}{r^2})^3$, any non deformed control point of the curve c_i, is altered by

$$v_i = \frac{v}{|v|} * g\left(\frac{d_i}{v}\right) \tag{2.4}$$

where d_i denotes the distance of the undeformed control point c_i to the currently modified control point along the warp curve, and v is the current deformation vector at this control point.

Figure 2.5 shows how the displacement at the selected point (original position represented through grey widget, current position represented through coloured widget) is propagated to the neighbouring control points (grey and yellow). The continuity of the displacement is preserved, while still allowing every control point being moved in different directions. It helps the user when only a smooth bump is desired, since only one control point needs to be moved to achieve this effect. In case more control is desired, several control points can be moved instead. This provides a very simple and efficient way to customize deformations along complex surfaces.

2.5.3 Creating the WarpCurve Deformation Field

Given the difference in the UI control, WarpCurves construct the deformation field differently than the previous voxel-grid based approach. It still uses the same variational implicit interpolation approach; however, the samples used to create the linear system are calculated in a different way. There is already a set of points and displacements defined using the curve control points, where the displaced control points should map to the original points before the transformation. In addition, a region surrounding these control points has to be defined, which bounds the displacement field to 0. In order to do this, so called *off-curve constraints* oc_i have to be added to the given displacements. The displacements at these oc_i values have to be set to 0 to ensure the amount of displacement decreases from the WarpCurve. However, this doesn't create a displacement field of local support, a property desired in the *BlobTree* modelling context. An approach to convert the displacement field to enable local support is described below in Section 2.5.4.

Every control point along the WarpCurve defines four additional off-curve constraints. Two of these control points are aligned with the normalized deformation vector $\hat{d}_i$ at each control point c_i, while the other two are in line with the normal of the plane at the point defined by the displacement vector and the tangent at ci_i.

$$oc_i = \begin{cases} c_i \pm \triangle l_i * \hat{d}_i \\ c_i \pm \triangle l_i * (\hat{d}_i \times \mathcal{T}(c_i)) \end{cases} \tag{2.5}$$

In this case, $\triangle l_i$ is the distance between the off curve constraint oc_i and the control point c_i, which according to [Sugihara et al., 2010] is defined as $2||d_i||$, resulting in the oc_i values being closer or farther away from the control curve depending on the displaced distance, effectively controlling the ROI of the deformation this way, as shown in Figure 2.6. Because the WarpCurve can be approximated by a 3D polyline, $\mathcal{T}(c_i)$ can be calculated using

$$\mathcal{T}(c_i) = c_{i+1} - c_{i-1} \tag{2.6}$$

and normalizing the result.

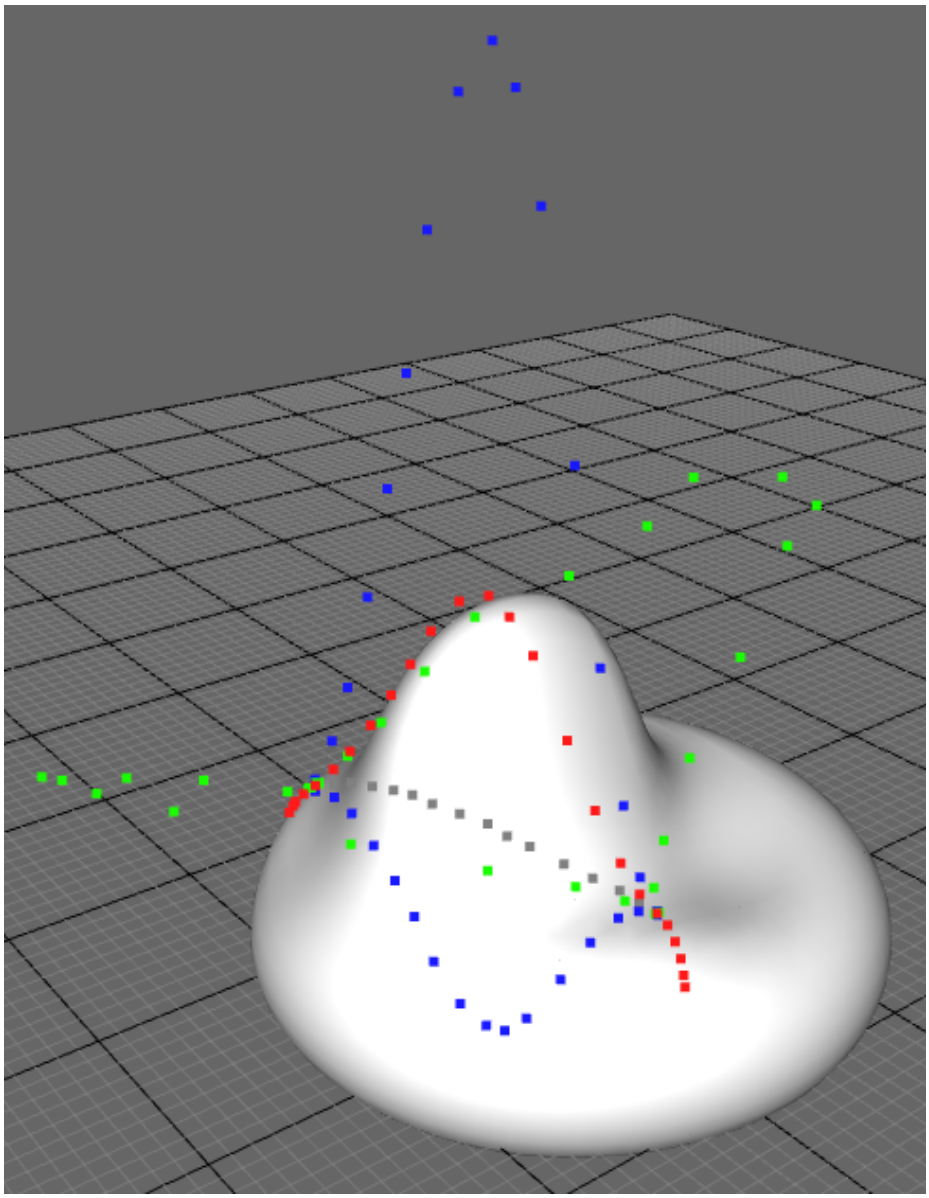


Figure 2.6: Off curve constraints (blue for top/bottom, green for left/right) are of varying distance to the displacement curve (original in grey, displaced in red).

2.5.4 Bound the Displacement Field using a Convolution Surface

The variational implicit interpolation is not bounded outside the off curve constraints but returns values other than zero instead (see areas marked − in Figure 2.7). The black dashed shape in Figure 2.7 is created by the off curve constraints of the orange displacement curve and illustrates where the displacement weights are set to zero.

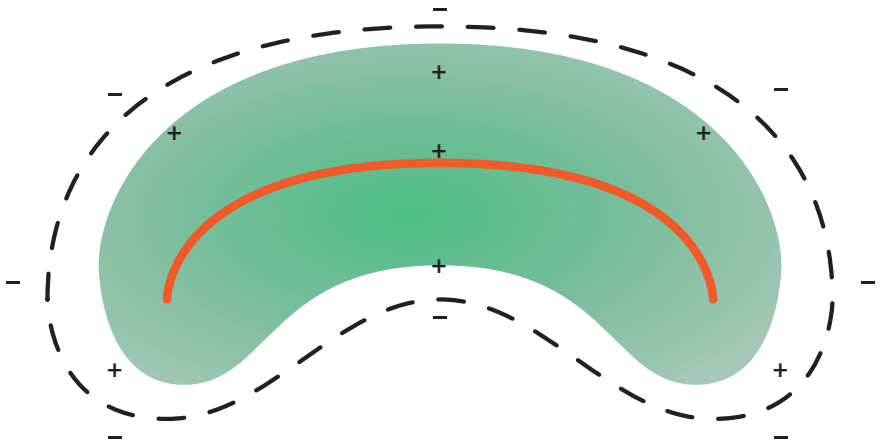


Figure 2.7: The relation between displaced line (orange), the convolution primitive (green) and the field created by the variational interpolation (black dashed line and + and − regions).

Within this shape, the displacement weights are "positive", whereas in the region outside the shape, the displacement weights are "negative". Because the *BlobTree* is based on bounded fields, the displacement field requires an additional modifer, which has a maximum value of one at the deformed curve, and decreases to zero at a finite distance. For this reason, a function that takes a sample point as an input and returns values in the range $[0, 1]$, depending on the distance to the deformed WarpCurve is needed. A convolution surface, with the WarpCurve as a skeleton, avoids the problems of other line based approaches that create bulges at joints [Bloomenthal, 1997]. The green region in Figure 2.7 illustrates the region $f > 0$ of the convolution field generated by the displacement curve.

In general, a convolution field is generated by convolving a skeleton $\mathcal{S}$ with a kernel function h (as per [Sugihara et al., 2010]):

$$f(p) = \int_{\mathcal{S}} h(p, \mathcal{S}) d\mathcal{S} \tag{2.7}$$

WarpCurves uses the *Cauchy* kernel [McCormack and Sherstyuk, 1998] to create the bounded field from the given polylines. Depending on the maximum displacement length along the WarpCurve, the kernel width is adjusted in order to create a convolution field of the right size, spanning the extents of the deformation. During these calculations, the field-values at the polylines can be greater than one, a property not desired in the *BlobTree* context. As a result, the maximum field value v_{one} within

the convolution field is calculated, and the final convolution field is modified by the Wyvill function:

$$f_{bounding}(p) = g(d) = (1 - \frac{f(p)^2}{v_{one}^2})^3 \tag{2.8}$$

The full warp within the WarpCurve approach is then calculated as follows.

$$d_{warp} = f_{bounding}(p) * d_f(p) \tag{2.9}$$

To calculate the field-value of the WarpCurve node's child value, the new warped position $p_{warp} = p - d_{warp}$ is used.

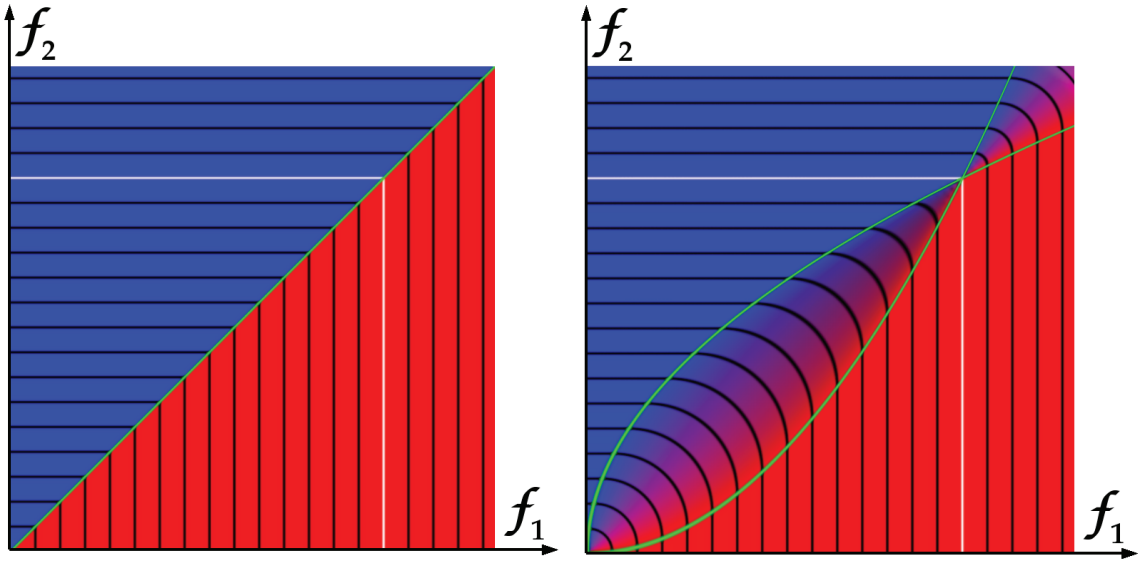
2.6 Gradient Based Blend

One major improvement in *BlobTree* modelling, and the foundation of Chapter 5, is the *Gradient Based Blend* (GBB) by [Gourmel et al., 2013] that includes a blend operator for a "continuous" union. Moreover, GBB defines a continuous operator function that smoothly interpolates between the standard summation blend

$$f_R(p) = \sum_{n \in N} f_n(p)$$

and the union operator. The latter is parametrized using an opening angle α, which defines the influence region of the blend. At an opening angle of $\alpha = 0$, the blend has a similar result as the standard summation blend and at an opening angle of $\alpha = 45$ the operator creates the clean union. Normally, the union operator is defined as $max(f_1, f_2)$, which produces a straight edge on the surface where the two objects meet. This is the desired behaviour of a union operator at $f = c_{iso}$, however the rest of the field ($f \neq c_{iso}$) is discontinuous, which is not desired.

Any binary implicit operator can be defined in the so-called *implicit extrusion field* $\mathbb{I}^2$ [Barthe et al., 2001] (see Figure 2.9), where the fields of both operands are mapped to the x, y axes of the graph. The origin of the graph is the region, where both of the operands are bounded in $3D$ space. The surface of the operands are defined by the iso value along the operands axis in $\mathbb{I}^2$. An operator in $\mathbb{I}^2$ defines a function that calculates a resulting field-value for any pair $(x, y) = (f_1, f_2)$. For example, the standard summation blend connects the two iso-value points along each axis with a quarter arc of a circle. By altering the function combining f_1 and f_2, new operators



(a) max union (b) clean union

Figure 2.8: Comparison between max union (left) and clean union (right) in the $\mathbb{I}^2$ space. Images from [Gourmel et al., 2013].

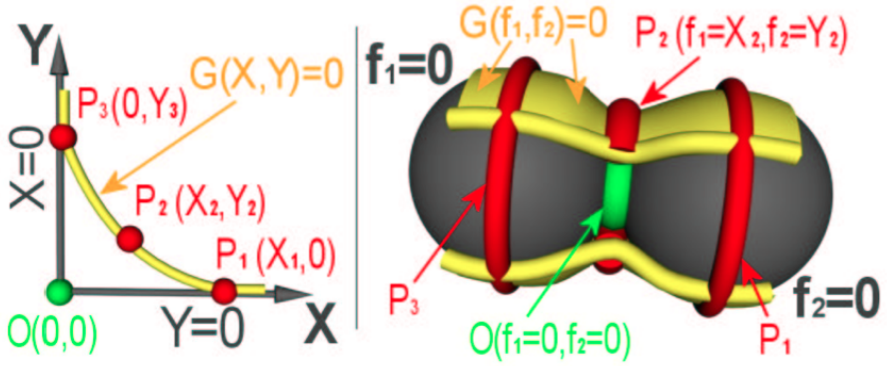
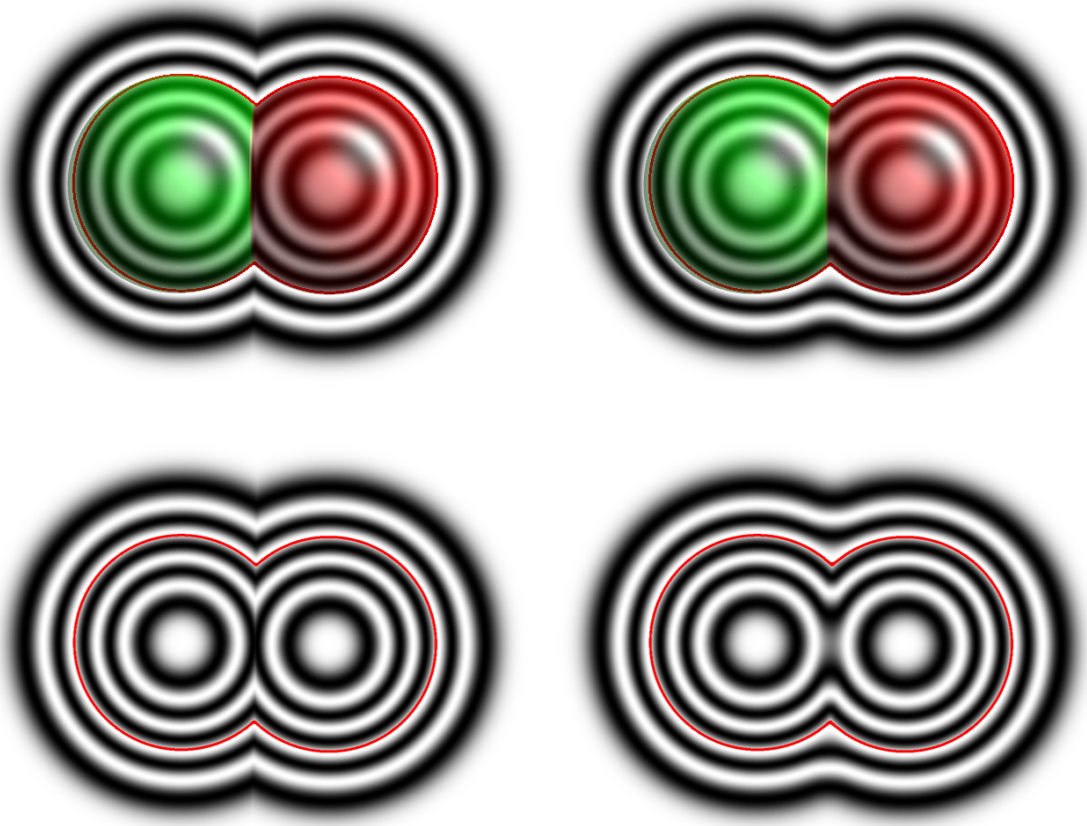


Figure 2.9: The implicit extrusion field $\mathbb{I}^2$ on the left and the corresponding blend between two implicit fields f_1, f_2 in $\mathbb{R}^3$ on the right. Image from [Barthe et al., 2002].

can be designed. Figure 2.8 shows how the standard $max(f_1, f_2)$ union differs from the $gbb(f_1, f_2)$ union when looked at in implicit space.

In general, implicit fields should be continuous in order to maintain the capability to blend properly, even after repeated combination through operators. A discontinuous field, as produced by the standard union max (or the intersection min), can produce undesired surface features when used in consecutive blend situations.



(c) max union (d) clean union

Figure 2.10: Comparison between max union (left) and clean union (right).
Top row: the context of the model (the same on both sides) and outside field (different)
Bottom row: shows the field inside the surface. Surface boundary marked red.

This problem has been first addressed by [Barthe et al., 2004], which defines a clean union operator based on complex numbers in $\mathbb{I}^2$. GBB simplifies this operator, while improving continuity from C^1 to C^∞, and provides a framework for precise user control to create blends of arbitrary "radius" between $f_R = f_1 + f_2$ and union $f_R = f_1 \vee f_2$.

Figure 2.10 compares the fields of the standard union operator with the *clean union* presented, and also shows the context of the surrounding field. It can clearly be seen how the field is different inside/outside the objects in both solutions. The clean union closely resembles a standard blend on the inside and outside, with the

transition being continuous.

The full Gradient Based Blend operator uses an additional control curve to customize the blending behaviour based on the angle of the gradients at both operands (thus the "gradient based" blend name). This, together with an optional shape function, solves a wide variety of problems (blending at distance, blend control, deformation at contact) within one unique framework. For example, by carefully choosing the the control curve (also called shape function $s(\varphi)$), blending of objects at distance can be removed. The same control curve also removes the problem that a blend between a large and a small object will not show the fine features of the small one.

2.7 Rendering the *BlobTree*

A *BlobTree* model can be visualized using two distinct approaches:

- ray-tracing [Bloomenthal, 1997] for high quality offline rendered images
- polygonization [Wyvill et al., 1986] for rendering the mode at interactive frame rates.

Both approaches rely on calculating the field-values ("evaluation of the tree") on multiple points p in space. A desired point can be the corner of a cube used in one of the common polygonization approaches or it can be the current point along a ray when ray-marching is used to render an image.

Both algorithms require traversing the full *BlobTree* and evaluating the distance functions of each skeletal primitive at p or the given range. Since all the *BlobTree* primitives are bounded, points outside the primitive boundaries can be omitted and *BlobTrees* can be rebuilt to avoid unnecessary field-value evaluations of nodes within the tree, as partly demonstrated by *de Groots Turbo Charged BlobTree* in [de Groot, 2008].

There is a long history of polygonization algorithms (the conversion of a *BlobTree* to a mesh) accelerating and improving the method by [Wyvill et al., 1986]. *Bloomenthal* [Bloomenthal, 1994] published a popular implementation of this uniform grid method , which overcomes ambiguities using tetrahedral decomposition. A more efficient algorithm was published in [Akkouche and Galin, 2001]. Various adaptive methods have been proposed, e.g. [Bloomenthal, 1988] and [van Overveld and Wyvill, 2004],

which convert an implicit surface into a triangle-mesh that has an optimized, non-uniform triangle distribution. ShapeShop [Schmidt et al., 2005a] (a sketch-based system) stores cache nodes in the *BlobTree* [Schmidt et al., 2005b] to allow for interactive feedback in his modelling system

In Ray-tracing the *BlobTree* is evaluated along a ray segment using interval analysis as described by [Mitchell, 1990] and [Snyder, 1992]. Alternatively, Lipschitz approaches, as demonstrated by [Kalra and Barr, 1989], can be used for faster ray-tracing. *RaySkip* [de Groot and Wyvill, 2005] builds on these ray-tracing approaches with an intelligent way to reuse rays for faster visualization times.

2.8 Summary

The *BlobTree* is the combination of the Skeletal Implicit Primitives with the blend operators described above, stored as a unified tree structure. In this tree, interior nodes can represent arbitrary blends between objects, as well as Boolean operations and warps at a local and global level (see Figure 2.4). The Skeletal Implicit Primitives or any Sketch-based shape, on the other hand, are stored in the leaves. Furthermore, the *BlobTree* stores transformations within the tree structure. When the tree is traversed, all operations, including visualization of the final *BlobTree*, depend on the field-value and a gradient (usually calculated using central differencing [Yagel et al., 1994]) returned for an arbitrary point in space.

Chapter 3

Efficient Data-Parallel Tree-Traversal for *BlobTrees*

Implicit modelling has several advantages over other solid modelling systems, but unfortunately, complex implicit models are often slow to render. While previous acceleration approaches reduce the number of field-value calculations (*BlobTree* traversals) and employ multi-threading to parallelize the visualization problem, little work has gone into accelerating the tree traversal itself.

Despite *BlobTrees* being seemingly similar to Constructive Solid Geometry (CSG) trees, as both store hierarchical models, some CSG tree traversal optimizations do not apply to the *BlobTree* due to the underlying differences. This chapter demonstrates how to optimize the *BlobTree* traversal by re-interpreting the tree as a mathematical expression in reverse polish notation. The performance impact due to memory use on modern Single Program Multiple Data (SPMD) devices is investigated and options to minimize it are proposed. The conclusion is that tree traversal is done faster bottom-up, resulting in an order of magnitude speed up for large trees.

3.1 Introduction

3.1.1 Motivation

Despite its unique and advanced features, the *BlobTree* by [Wyvill et al., 1999] as a data structure for implicit models is often considered too slow for interactive modelling purposes. Several researchers, such as [Schmidt, 2006] and [Shirazian et al., 2012], have shown that visualization methods can indeed be fast enough to re-create a new

mesh object from a *BlobTree* at interactive frame-rates. Other approaches have shown that ray-tracing times can be reduced, for example, by using interval-arithmetic(see [Knoll et al., 2009]) to find the intersection of a ray with the *BlobTree* surface. A main commonality of all these approaches is that they speed up the visualization by reducing the number of implicit (*BlobTree*) evaluations at points in space. Moreover, many acceleration approaches also rely on the fact that both ray-tracing and polygonization can be done in parallel, since each field-value calculation is independent. Thus the problem lends itself to an application that makes use of the Single Program Multiple Data (SPMD) paradigm as described by [Darema et al., 1988].

Nevertheless, little work has been done to actually improve the time a single tree evaluation takes, despite recent advances in *BlobTree*-modeling that introduced more complex operators to expand the capabilities of the *BlobTree* by [Bernhardt et al., 2010] and [Gourmel et al., 2013] and offer greater control to the user. Since every acceleration approach is based on these tree evaluation calculations, all of them would benefit from improving the time it takes to evaluate the *BlobTree* at a point in space.

3.1.2 The *BlobTree*

A *BlobTree* is a representation of the syntactic parse tree Φ of the evaluation of a scalar value $f(R, P)$ at a query point P, where R is the root of Φ. The leaves of Φ represent shapes. For a leaf, L, $f(L, P)$ is a scalar value calculated by taking the distance $d(P)$ of P to a skeleton $L.S$, that then is modified by a filter-fall-off function g. An internal (non-leaf) node, N, may have one or more children in Φ. When N has a single child $N.C$, the value $f(N, P)$ returned by N is $N.f(N.C, t^{-1}(P))$ where $N.t$ is a transformation associated with N that may define an affine transformation (translation, rotation, scale, shear), a bending or twist, or other space warps. When N has several children $N.C_i$, the value it returns is $N.g(f(N.C_0, P), f(N.C_1, P)\dots)$, where function $N.g$ combines scalars and may be used to implement certain forms of blends that smoothly join surfaces. Figure 3.1 illustrates how a model is built using this concept.

BlobTree modelling is related to Constructive Solid Geometry (CSG) [Sabin, 1968], where much work has been done to accelerate tree traversal (e.g. [Jansen, 1991], [Hable and Rossignac, 2005] and [Rossignac, 2012]), as well as novel rendering techniques (e.g. [Romeiro et al., 2006]). Unfortunately, approaches improving tree traversal for CSG are not necessarily applicable in this case, due to the different mathemat-

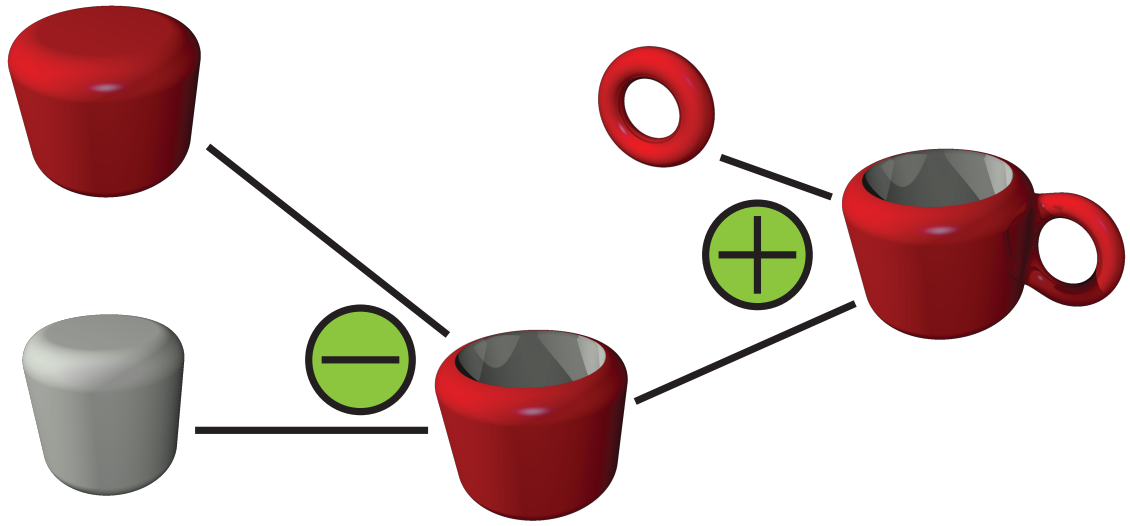


Figure 3.1: A simple example of a *BlobTree* building a mug. − describes a *difference* operator and + a blend.

ical formulation of the *BlobTree* compared with pure Boolean CSG trees. Depending on the application, CSG evaluation algorithms classify points, line segments, or surfaces, but always return point sets (possibly augmented with set membership maps for their neighbourhood), while *BlobTree* evaluation algorithms return a scalar value at a query point. Direct CSG classification algorithms classify (i.e., trim) these candidate sets against the leaves of the CSG tree (i.e., the primitive solids) and then merge the classified subsets up-the-tree, according to the Boolean operations (see [Mäntylä, 1987]). CSG trees that support such algorithms are limited to have nodes that represent regularized Boolean set operators (Union, Intersection, Difference) and affine transformations. Offsetting, blending, and Minkowski operations require evaluating a boundary representation of the solids associated with the argument nodes of such operations and hence do not lend themselves to a direct CSG evaluation. In contrast, *BlobTrees* evaluate scalar values at the query point, one per *BlobTree* leaf (primitive) and then blend, filter and combine these values according to various formulae (e.g. [Barthe et al., 2004] and [Gourmel et al., 2013]), which may reproduce Boolean operations and also their blended versions.

3.1.3 Contributions

In this chapter the tree structures of CSG trees will be compared with *BlobTrees*, and why the acceleration methods for CSG tree traversal are not applicable is explained.

It is shown how to accelerate the tree traversal for the *BlobTree* by traversing bottom-up that results in an $O(n)$ traversal time. This results in predictable memory access patterns which are important for performance on modern SIMD architectures, such as GPUs using OpenCL, or using vector instructions on CPUs. In this approach the tree information is stored in a linear memory pattern and can improve traversal speed by as much as an order of magnitude, compared to previous approaches running on an SPMD architecture.

3.1.4 Outline

The remainder of this chapter is structured as follows. Section 3.2 discusses related work in SPMD programming, as well as *BlobTree* and CSG acceleration. A summary of the most efficient CSG traversal accelerations is given in Section 3.3 and Section 3.4 describes how some of these changes can be applied to the *BlobTree*. The *BlobTree* supports non-affine transformation within the tree structure, which need special care in this traversal approach, outlined in Section 3.5. Section 3.6 introduces the implementation, the results are discussed in Section 3.7 and future work is stated in Section 3.8.

3.2 Related Work

3.2.1 The SPMD programming model

Current computer architectures provide two main paths for accelerating floating-point heavy workloads: SIMD (Single Instruction Multiple Data) units, which evaluate the same floating point operator on multiple (typically 4, 8, or 16) elements of data; and GPUs (as described by [Fatahalian and Houston, 2008]), re-purposed to general computation using a large set of SIMD-like processors, with hardware predication for divergent control flow. This last model is better known as the Single Program Multiple Data (SPMD) programming model described by [Darema et al., 1988]. In an SPMD program a single execution stream is applied over a large number of independent data elements; any dependencies between these elements cannot be expressed directly in the model, and have to be managed outside of it. Applying the SPMD model usually involves rethinking the algorithm with respect to optimal memory transfer and problem blocking (e.g. [AMD, 2011]) to allow maximum data independence.

Architectures embodying the SPMD programming model usually also have significant memory bandwidth compared to execution speed to allow the transfer of the many independent data streams. In many cases recalculation can be substantially more efficient than sharing.

Several high-level programming languages provide facilities to help the programmer create code to run on SPMD architectures, such as OpenCL [Munshi, 2011] for GPUs, or ISPC [Pharr and Mark, 2012] for CPUs. OpenCL, the industry standard for SPMD on programmable Graphics Processing Units (GPUs), currently uses a high-level programming language derived from C, where the only custom data structure is a struct that does not support inheritance or member functions. This means significant rethinking of the data structures used to represent a hierarchical tree structure, as used in both CSG and the *BlobTree*, since a naive implementation typically makes use of the non-existent features class inheritance and virtual functions.

3.2.2 Accelerating *BlobTree* rendering

Previous work on improving the traversal time of *BlobTrees* include aggregating nodes in the tree to reduce overall node count [Fox et al., 2001]. Furthermore, a simple approach using spatial subdivision together with pruning the tree for each subdivision node is also suggested and explored further in [de Groot, 2008]. Caches within the tree structure are introduced by [Schmidt et al., 2005b], in order to reduce the number of field value calculations, in favour of interpolation of field-values as soon as enough values are found within the cache structure. This work enabled interactive editing through polygonization but for edit operations it still relied on polygonizing at a coarser resolution to allow for fast interactive feedback. Polygonization itself is an algorithm that lends itself to a parallel implementation using SIMD, as shown by [Shirazian et al., 2012], which makes use of linearizing a tree structure into continuous memory and calculating multiple field-values in parallel, since each calculation is independent.

All of the above approaches improve rendering times significantly but none of them investigated methods to reduce the time a single tree traversal takes, while still calculating the exact field-value. It is not obvious that improving the time taken by the calculation of the field-values is possible. The formulas for primitives and operators are constant since they cannot be changed without changing the resulting values. Thus, it is a very important condition that any acceleration approach does not alter

the results of the field-value calculation at a point in space. Throughout this chapter it will be shown that it is still possible to accelerate the field calculations without altering the underlying formulas by taking certain hardware properties, especially those of modern GPUs, into account.

In another GPU approach by [Reiner et al., 2011], the model is built by altering the GLSL shader code that renders models directly to the screen, but is less practical than the approach presented in this Chapter, as a new shader is required whenever the model changes. Other approaches, such as [Gourmel et al., 2010], present a method to ray-trace meta-balls on the GPU. Neither of the above deal with tree structures similar to the *BlobTree*, and are not comparable systems because they cannot produce the model complexity of the *BlobTree*.

3.2.3 Accelerating CSG rendering

Most of the current state of the art GPU rendering techniques for CSG models are based on the formulation of a CSG tree as a *Boolean list (Blist)* [Rossignac, 1999]. *Blister* [Hable and Rossignac, 2005] and subsequently [Hable and Rossignac, 2007] use this Blist formulation of a CSG tree in a rendering technique based on depth peeling. The depth peels are classified for each primitive based on stencil bit true/false values that are then combined using the Boolean expression given by the CSG tree. Several optimization methods are found in these approaches that eventually resulted in the formulation of *Ordered Boolean Lists* [Rossignac, 2012], which shows that this Boolean List based approach can be used in other areas other than CSG rendering as well. The work on OBL can be considered related to earlier work on boolean expressions, such as Reduced Ordered Binary Decision Diagrams (ROBDD) (see [Bryant, 1986] and [Bryant, 1995]).

Other approaches to render CSG models on GPUs include [Romeiro et al., 2006], where CSG objects are subdivided until each child is simple enough (one primitive or boolean operator of two primitives) for rendering on the GPU. This subdivision is done on the CPU, whereas every node that only contains one primitive is rendered on the GPU. It is also possible to use a face representation precomputed on the CPU to directly render the CSG objects using boundary representations as described by [Hanniel and Haller, 2011].

Using a tree structure efficiently in an SPMD context requires linearizing the tree into a continuous block of memory. Many approaches for linearizing tree structures

store the tree nodes in top down order, with offset pointers used to do the traversal ([Smits, 2005] and [Bunnell, 2005]). Some approaches insert additional offset pointers so it is possible to directly go to the in-order predecessor or successor by one single offset pointer (threaded tree [Wyk, 1991]) instead of reading the parent pointer again in order to find the neighbouring child node. Linearizing and traversing trees has been important in the context of acceleration structures (e.g. [Wald, 2004], [Smits, 2005], [Bunnell, 2005], [Benthin, 2006] and [Popov et al., 2007]).

3.3 Methods to accelerate CSG tree traversal

For any binary tree structure Φ, the top node of the tree is called the *root* node. Any interior node represents a subexpression of Φ, which for a node N is often described as $L \circ R$, where L and R are called the *left* and *right children*, $\circ$ the operator to combine them and N is called their *parent* and a *link* is the connection between parent and child nodes. A *walk* between two nodes within the tree is defined as the minimum connected subgraph containing both nodes, with its *distance* being the number of links in the walk. The *height* of a node is defined as its maximum distance to its leaves and the *path* to a leaf is called its walk to the root node.

The most efficient methods to accelerate the traversal of CSG trees involve writing the tree as a boolean expression. In such an expression, the leaves of the tree correspond to the literals, which at any 3D point either evaluate to *true* or *false*. The operators (notation as per [Rossignac, 2012]) in the expression can be of the limited set of union, expressed as a Boolean OR ("+"), intersection as AND ("•") and the difference operator as AND NOT ("•!"), where "!" denotes the complement. Operators that can create more complex transitions between surfaces, such as the ones described in [Elber, 2005] and [Whited and Rossignac, 2009], are usually not considered in these approaches. Any tree only consisting of the three simple operators stated above can be described using such a Boolean expression which can be evaluated in parallel for any input points and classifies this point against the CSG model surface.

A CSG tree that uses only these three operators ("+", "•", "!") and that has the "!" operator pushed to the literals using the de Morgan laws is called a *Positive Form Expression* (PFE). In this expression, which is the basis of the *Blist* wiring process for optimization, only the + and • operators exist; the "!"-operator is expressed as a parameter to the actual primitives. Both operators in this PFE are commutative, so they can often be swapped to make the tree left-heavy, which can reduce the

footprint of the expression (see section 3.4.2 for a discussion on left-heavy trees). The Blist wiring process uses the metaphor of an electrical circuit, where every literal is expressed as a switch reading its boolean value. Each switch has a top output, representing true, and the bottom output, representing false. Depending on the value of the literal, the incoming "current" is directed to one of these two outputs.

Two switches, A and B, can be connected together to form either the expression $A+B$ or $A \bullet B$ by altering the connections between input and output. This allows the introduction of connections that can skip the evaluation of nodes, e.g. in the expression of $A + B$ the result is already set to *true* as soon as A evaluates to *true*. There is no need to evaluate B in this case. Similarly, for the $A \bullet B$ case, if A evaluates to *false*, the whole expression results in *false*, not requiring B to be evaluated. This also is a variation of a ROBDD wiring process [Bryant, 1986], since both approaches produce wiring where every literal appears only once, and connections between them don't cross. In an Ordered Boolean List, the Blist structure is then stabilized (reducing the nodes' width, as defined in [Rossignac, 2012]), by continually swapping nodes to effectively re-order the tree, resulting in the smallest memory footprint possible for each expression. This results in OBLs that can be created from any expression and can be evaluated in $O(loglog\ n)$ space.

To summarize, many efficient CSG visualization techniques (on the GPU) rely on the fact that any primitive can be reduced to a Boolean value. In addition, the operators are of a limited set and allow skipping the evaluation of child nodes for certain cases. Re-sorting the tree can reduce the upper boundary of the required memory footprint.

3.4 Techniques applicable to the *BlobTree*

Optimization approaches, which rely on the simplification of boolean expressions are not applicable to the *BlobTree*. Unlike boolean CSG operators, both sides of a *BlobTree* operator have to be evaluated. All nodes return continuous values that are combined as described above, potentially changing the inside/outside classification for a point P based on its numerical value. For example, using the Ricci operations, union is $\max(f_1, f_2)$. Even if $f_1 < c$, it still has to be evaluated, as f_1 may be greater than f_2. Only combining inside/outside information, such as in CSG, will disable the mathematical foundations behind all the blend operators, which add to the *BlobTrees* unique modelling features. Operators other than Ricci's can also be

used in the *BlobTree*, such as Pasko's [Pasko et al., 1995] R-functions, but the same principle applies.

Pushing an *invert* operator to the leaf nodes of a *BlobTree* does not work. Given that in the *BlobTree* the right child's field-value is not just negated, but the complement is calculated using $1 - f$, an inversion at the child node is not possible. This is due to the fact that at a leaf node, the field-value returned is within $f \in [0, 1]$, but once this field-value f is part of a binary blend node, it can theoretically result in $f = 2$, (i.e. when the point lies on both skeletons). In a situation where there are more blends involved, field-values at object interiors can become bigger than 1 and, subsequently, < 1 after a difference node. An inversion compatible with the aforementioned CSG acceleration approach, on the other hand, would calculate the compliment by $-f$, resulting in a different field-value than $1 - f$. For these reasons, only a subset of the aforementioned CSG acceleration methods can actually be applied to the *BlobTree*.

3.4.1 Hardware Considerations

Memory reads and writes can have a significant impact on the performance of modern processors, both for CPUs and GPUs. Modern hardware accelerates the performance of applications using prefetching mechanisms and hardware caches. In case the memory footprint of the tree is very small, it might be possible to fit a large portion or all of it into the hardware cache, and, more importantly, keep it there for most of the time. As a result, whenever tree information is read, the access is done from the representation in cache, avoiding the more expensive read from main memory (that, in addition, can produce a context switch on the hardware, etc.). Even if the whole tree does not fit into the cache, it is possible to exploit the behaviour that memory is read in cache lines (or equivalents on GPUs). Reading memory in cache lines means that independent of the size of the memory read, it is read in chunks that are sized equally to the cache line. In the case that the memory read is smaller than one cache line, the actual memory read consists of the desired size, plus the remaining memory until the cache line size is reached. This means that when an element is read from an array that is a fraction of the size of a cache line, succeeding array elements are read as well. If these elements are accessed shortly afterwards, they can still be read from the cache, instead of doing another expensive main memory transfer.

Consequently, designing an algorithm to make the best use of these techniques can

result in a significant performance improvement. The Bottom-Up traversal method relies heavily on these hardware characteristics because it produces a coherent memory access pattern to data-structures that are stored in a way to support cache aligned reads and writes. Two measures need to be optimized to have a well performing traversal algorithm of *BlobTrees*:

- reduce the memory footprint of the tree structure as much as possible and align it to the cache line size (see Section 3.4.2)
- reduce the reads/writes to temporary storage (see Section 3.4.3)

3.4.2 Linearizing a *BlobTree*

In order to use a tree data structure efficiently on modern SPMD architectures (GPUs or CPU vector instruction sets), it is necessary to store the whole tree in contiguous blocks of memory. The usual implementation, where links in the tree are represented by pointers, can lead to bad performance, since memory reads are harder to predict for the hardware, and, in many cases, the pointers won't lead to a cache aligned memory distribution. Previous approaches to improving tree traversal are based on *linearizing* the tree structure (information about parent - child relation) and data (information on the type of node) into arrays of contiguous memory (e.g. [Wyk, 1991], [Smits, 2005], [Bunnell, 2005], [Wald, 2004] and [Shirazian et al., 2012]).

Algorithm 1 Recursive solid model tree traversal

```
1: function DataAtRecursive(Point p, TreeNode curData)
2:     if n is a primitive then
3:         return dataAtPrim(p, curData)
4:     else
5:         childResults[0] ← DataAtRecursive(p,curData.child0)
6:         childResults[1] ← DataAtRecursive(p,curData.child1)
7:         return dataAtOp(p,curData,childResults)
8:     end if
9: end function
```

A simple application that traverses the tree top-down iteratively and stores child relations only in the parent node requires that the parent node is read after visiting the first child. More generally, any top-down traversal algorithm of a tree maintains two stacks of temporary memory:

- the traversal state (information about the previous visited node), m
- the temporary results at the nodes, to be used in the parents, t

Both of these stacks depend on the number of nodes n within the *BlobTree* and its structure.

Algorithm 1 illustrates the simplicity of a tree traversal approach done recursively, especially since intermediate results and the traversal stack are provided implicitly through the recursion stack. However, it has been proven numerous times that the recursive approach, despite being elegant, can in some cases not perform optimally. In such instances, rewriting the recursive traversal into an iterative approach can improve running times significantly, in addition to allowing it to run on architectures with limited recursion support (GPUs, IBM Cell, etc).

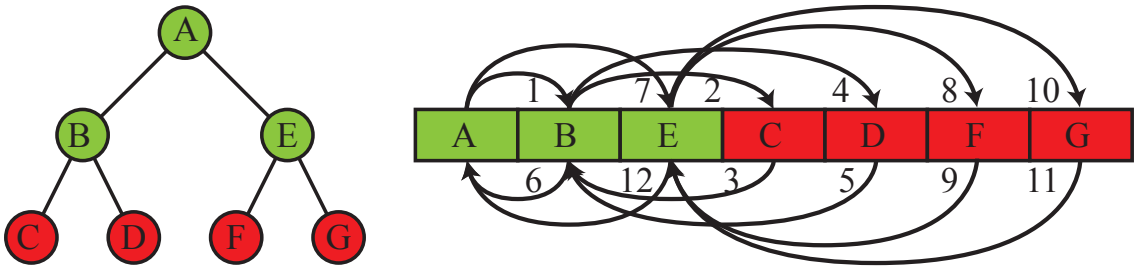


Figure 3.2: The order of the tree nodes in memory. The numbers describe the order of memory reads when the full tree is traversed. Top arrows: reads going deeper in the tree; numbers are closer to arrow head. Bottom arrows: reads going back up in the tree; numbers closer to arrow base.

Figure 3.2 shows how memory is read (not including the temporary storage stack) using the approach based on Algorithm 2 (recursive traversal rewritten iteratively using stacks), resulting in 12 memory reads for the traversal, with 5 changes in the read direction for this small example. In this case, the tree storage is already optimized according to [Wald, 2004] that stores child nodes in neighbouring array elements, removing the need to store 2 index offsets per parent node pointing to the children. Whenever the traversal moves deeper down the tree, the current node is pushed to the traversal stack m. In case of a primitive, the data is computed using the corresponding primitive function for p and stored it in the temporary stack t for later use. If the current node is an operator, a check if both children have been visited is needed and if a child is still to be processed the current node is pushed to the stack and the new current node is the child node. As soon as both children of an operator have

Algorithm 2 Iterative tree-structure-based traversal. Both the TraversalState m and the TemporaryResults t are stacks

```
1: function DataAt(Point p, Tree Φ, TraversalState m, TemporaryResults t)
2:     curNode ← Φ.root
3:     while true do
4:         if curNode is a Leaf then
5:             curResult ← dataAtPrim(p,curNode)
6:             if m not empty then
7:                 push curResult to t
8:                 curNode ← done
9:                 curNode ← pop from m
10:            else
11:                return curResult
12:            end if
13:        else
14:            if curNode.child0 ≠ done then
15:                push curNode to m
16:                curNode ← curNode.child0
17:            else if curNode.child1 ≠ done then
18:                push curNode to m
19:                curNode ← curNode.child1
20:            else
21:                childResults[1] ← pop from t
22:                childResults[0] ← pop from t
23:                curResult ← dataAtOp(p,curNode,childResults)
24:                if m not empty then
25:                    push curResult to t
26:                    curNode ← done
27:                    curNode ← pop from m
28:                else
29:                    return curResult
30:                end if
31:            end if
32:        end if
33:    end while
34: end function
```

been calculated their results stored on stack t are used to calculate the data values of the operator. These are written to the stack t and the information in m is to move back up the tree towards the root node.

Apart from the fact that the tree itself is read in an unpredictable way (Figure 3.2),

both stacks are involved in many reads and writes. Stack m has the size $|m|$ of the maximum height found in the tree. In order to quantify the size of stack t, the property *right-heavy-ness* needs to be defined, which can be calculated recursively. If an interior node N has two leaf nodes L, its right-heavy-ness is 2. In the case that N's *right child* is an operator node, the right-heavy-ness is the maximum of its right-heavy-ness of the left child and 1 plus the right-heavy-ness of the right child. The resulting value corresponds to the size of the temporary storage stack t. Dependent on the tree structure (see a more detailed discussion in Section 3.4.4), $|t|$ and $|m|$ have certain size characteristics, but for any tree $|m| \geq |t|$ (n is the number of leaf nodes in the tree):

1. left-heavy: $|t|$ is 2, whereas $|m|$ is n.
2. balanced: $|t|$ and $|m|$ are $log(2n) + 1$.
3. right-heavy: $|t|$ and $|m|$ are n.

Even though this approach based on a linearized tree is faster than a non-linearized version, it can only be considered the performance base-line. Better traversal performance can be achieved by optimizing the tree storage and, as a result, the number of stacks/stack frames needed. In this method, every interior node N is visited three times, every leaf L visited once, numbers that can be reduced for the interior node N case, as demonstrated below.

3.4.3 Eliminating the need for a traversal stack

Much work has been done on how to represent parent child relations of a tree in memory in order to avoid unnecessary memory reads, e.g. in theory it is not necessary to visit the parent N node to change from child L to child R. As a result, a *threaded* tree stores this none relationship directly in every child node by saving the offset to the in-order predecessor and successor [Wyk, 1991]:

- node L stores an array offset to N and to R
- node R stores an array offset to L and N.

In cases where L or R are trees and not leaves, these offsets are to the appropriate leaf nodes instead. Since for a *BlobTree*, the parent nodes combine the results of the

child nodes, these need to be visited after processing the right child as well, which is not needed for the general threaded tree approach. The memory layout of the tree and the memory access pattern of the threaded approach are shown in Figure 3.3, where the number of memory reads are reduced to 9.

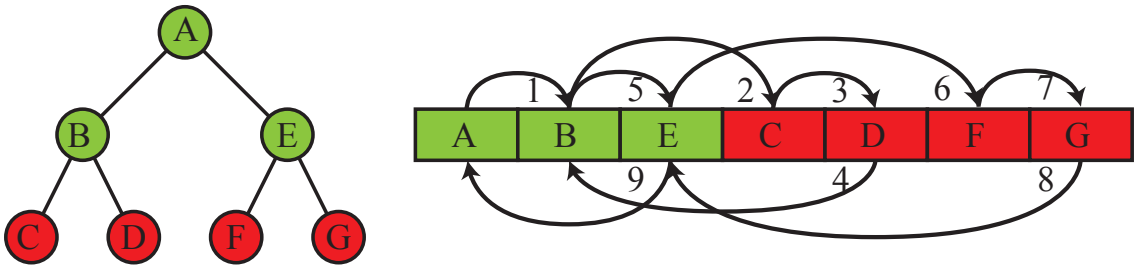


Figure 3.3: The order of the tree nodes in memory. The numbers describe the order of memory reads when the tree is traversed using a threaded tree approach. Top arrows: reads going deeper in the tree; numbers are closer to arrow head. Bottom arrows: reads going back up in the tree; numbers closer to arrow base.

Even though the threaded tree approach already eliminates the need to keep a traversal stack m, as presented by [Popov et al., 2007], it still relies on changing the read direction in memory 3 times, and is not optimal for all SPMD architectures [AMD, 2011].

Looking at the architecture of modern SPDM hardware, especially GPUs, non-cached reads from main memory can often be very slow (up to 500 cycles or more). This makes finding an approach that does not need to store offset pointers to traverse the tree a desired goal. In fact, not having to store offsets for parent-child relations means that only the tree data is needed.

A *BlobTree* can be treated as a mathematical expression, with the literals being the leaf nodes (primitives) that are combined using operators. As a result, rewriting the expression in the reverse polish notation is possible, as done i a Blist for CSG. In this notation, both operands precede the corresponding operator; thus, the expression $A \circ B$ would be rewritten as $AB\circ$. This corresponds to a post-order / bottom-up tree traversal, shown in Figure 3.4. Compared to a top-down approach, the memory access pattern of the tree data is simpler, reads are only done in one memory direction, independently of tree size and structure, and, as a result, are easier to predict for current hardware. This results in 6 reads from memory, compared to the 12 in the original approach, and 9 in the threaded optimization. If a certain tree node is of a fraction of the size of a cache line, it can happen that when one node is read into

memory, the following node is read as well, due to the cache line size.

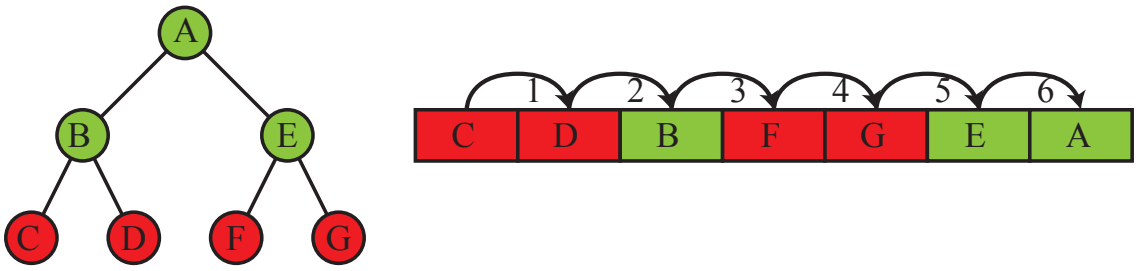


Figure 3.4: The order of how the tree nodes are stored in memory. The numbers describe the order of memory reads when the full tree is traversed. The only reads are going bottom up in the tree. It results in: less memory reads, memory is only read in one direction and only once.

For every field-value computation, the intermediate calculation results need to be stored in order to combine them at operator nodes. In a recursive approach, storage for these intermediate results is implicitly given by the recursion stack, which the bottom-up approach does not require. Since the traversal is based on the reverse polish notation of an expression, the storage layout for the temporary results is a stack. Every intermediate result is pushed on this stack and at an operator node the last two results are popped from the stack (first result is the right child), so they can be combined. This results in algorithm 3 (for a single thread/field-value calculation).

3.4.4 Optimize the tree to require less temporary storage

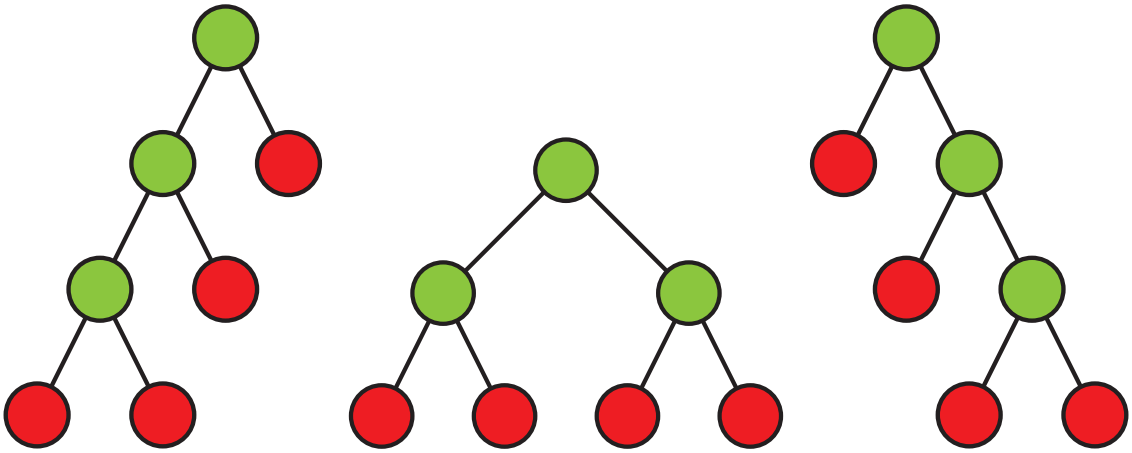
Given that the presented implementation targets SPMD architectures and multiple field-values will be calculated in parallel, the temporary results stack t is needed for every thread. The more threads are used, the more storage is needed. The maximum size of the stack depends on the structure of the tree. There are three extreme ways a tree with n_l leaf nodes can be combined to build a solid model tree:

- left-heavy (Figure 3.5a)
- balanced (Figure 3.5b)
- right-heavy (Figure 3.5c).

Depending on this structure, trees with the same number of leaves (primitives) can need a bigger or smaller stack to store the intermediate results. The best case is

Algorithm 3 bottom-up traversal

```
1: function DataAtLinear(Point p, TreeArray a, TemporaryResults t)
2:     n ← a.length
3:     for i = 1 → n do
4:         curData ← a[i]
5:         if curData is a primitive then
6:             curResult ← dataAtPrim(p, curData)
7:             push curResult to t
8:         else
9:             childResults[1] ← pop from t
10:            childResults[0] ← pop from t
11:            curResult ← dataAtOp(p,curData,childResults)
12:            push curResult to t
13:        end if
14:    end for
15:    return pop from t
16: end function
```



(a) Left-Heavy (b) Balanced (c) Right-Heavy

Figure 3.5: The three different extreme structures for a tree with n leaf nodes.

a left-heavy tree, as shown below in Section 3.7, since the extreme left-heavy example in Figure 3.5a can be traversed with a constant stack size of 2, independent of how many leaf nodes n there are in the tree. Optimizing a tree so that it is as left-heavy as possible is desired in order to reduce the size for the temporary variable stack and thus accelerating the traversal. The basis of this optimization is that the height at every node in the tree is computed. This can actually be done when the tree is built,

as shown by [Hable and Rossignac, 2005]:

1. any new primitive has height 0

2. any new node's height is the maximum of the height of its children, plus 1

Swapping the tree nodes can be integrated into the algorithm that traverses the tree and produces the reverse polish memory layout. If the height of the right child is bigger than the left, the algorithm has to traverse the right child first, essentially swapping left and right in this case. Every operator in the *BlobTree* apart from the difference operator is commutative, thus no additional changes are necessary for the traversal to still produce the same field-value. In the case of the difference operator, the easiest way to support swapped nodes is to set a flag if the node is swapped. When the traversal algorithm processes a flagged node, the algorithm assigns the child results in the opposite order.

This means that the absolute worst case arrangement of n_l leaf nodes, a perfectly right-heavy tree can be transformed into the best case. When a left-heavy tree is created from a perfectly right-heavy tree, the size of the temporary storage stack t is reduced to a constant size of 2. In general, any right-heavy representation can be converted to a corresponding left-heavy one, effectively making the previous average case, a balanced tree, to the new worst-case. A perfectly balanced tree has the size requirements of stack t (right-heavy-ness) of $log(n_l) + 2$, where n_l is the number of leaf nodes. Any other (already left-heavy transformed) tree with n_l leaves needs a stack size that is in between and 2 and $log(n_l) + 2$.

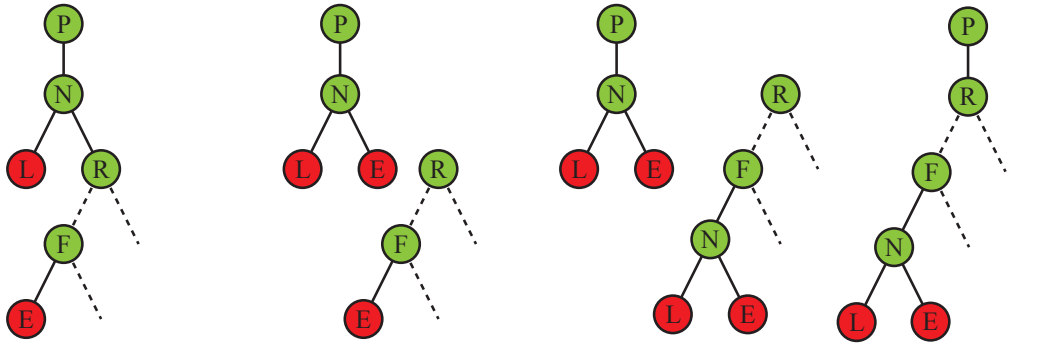
In some cases, the structure of the tree can be converted to a left-heavy version by reordering parent and child relations. This does not work for all combinations of operators, only for ones that are associative and commutative in limited arrangements. The algorithm starts at a node N with children L and R that have been pivoted to be left-heavy. The leaves of L and R may be primitives or roots of subtrees with non-compatible operators. In addition, E is the left most child of R, F the parent of E and N one child of P. To pivot the nodes, the following steps have to be applied to the tree:

1. Replace R by E in N.

2. Replace E by N in node F.

3. In P replace N by R.

See Figure 3.6 for an illustration of before and after. This algorithm can only be applied when the path between N and F only consists of the same operator type supporting the pivot: the union, intersection and summation blend operators.



(a) Before Rotation (b) After step 1 (c) After step 2 (d) After step 3

Figure 3.6: Conversion to left-heavy tree. Nodes N and F have to be compatible for allowing the rotation. Dashed lines represent either subtrees or nodes with operator types supporting the pivot.

3.5 Incorporating Non-Affine Transformations

In addition to affine transformations, the *BlobTree* also supports a series of non-affine transformation, such as the *Barr Warps* [Wyvill et al., 1999] and more recently *Warp Curves* [Sugihara et al., 2010]. Like affine transformations, these non-affine transformations are represented in the *BlobTree* as unary nodes at any point in the tree, transforming the whole subtree underneath. Figure 3.7 shows an example, where the blue nodes represent non-affine transformations. These nodes apply the inverse transformation to the specific input point coordinates, moving the input point according to the non-affine transformation to its new location used for any further calculations. This behaviour is perfectly acceptable as long as the tree traversal is done top down; however, the bottom-up approach discussed above needs more work to support non-affine transformations.

One major property for enabling the bottom-up traversal is that all affine transformations can be pushed to the *BlobTrees* leaf nodes during linearization. This makes it possible that the input point coordinates can be transformed into every leaf node's local coordinate system with only one matrix-vector multiplication, done

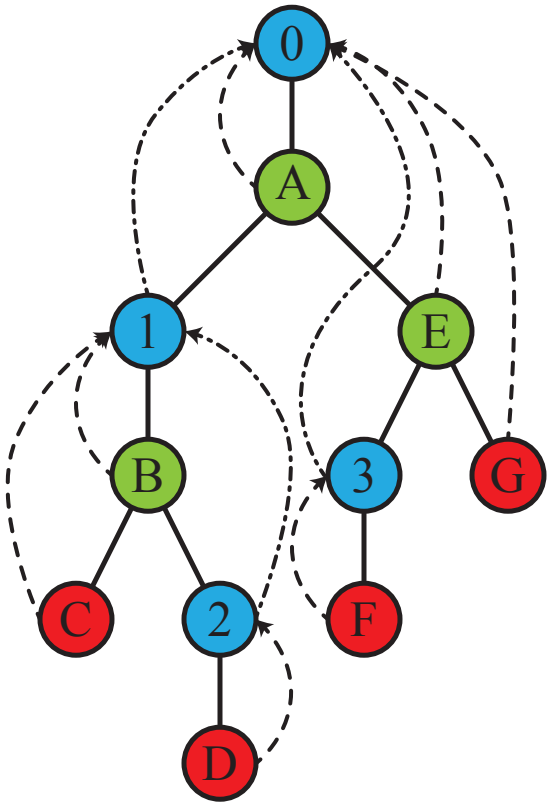


Figure 3.7: A *BlobTree* including non-affine transformations shown in blue. Stippled lines show the connections for each node to its parent non-affine transformations. Stippling with line-dots connect the non-affine transformations

once at every leaf node. It has been shown by [Wald, 2004] that pushing the affine transformations to the leaf nodes can cause a large performance improvement simply by reducing the number of matrix-vector calculations during a tree traversal (not limited to *BlobTrees*), even in other traversal situations not involving the bottom-up approach.

As soon as one non-affine transformation is along the path from the root node to the leaf, all the affine transformations can't be pushed to the leaf anymore. However, pushing affine transformations and aggregating them until the first non-affine transformation is reached is possible. Similarly, pushing and aggregating all affine transformations below a non affine transformation is possible until either another non-affine transformation or the leaf is reached. As a result, aggregated affine transformations can still be stored as parameters to the leaf, instead of nodes within the *BlobTree*, with the addition that the non-affine transformations potentially store such

a transformation as additional property.

In a top-down tree traversal, this optimization would result in a *BlobTree* that, in addition, can contain unary non-affine transformation nodes. As mentioned above, the inverse of these transformations is applied to the input points before the tree traversal continues, until the leaf nodes are reached. Field values are calculated and then combined bottom up to produce the final traversal result.

Looking at this, a full *BlobTree* traversal in this situation consists of two steps:

1. top-down tree traversal: transforming the input points
2. bottom-up tree traversal: combining the field-values and potentially gradients and colour.

In both cases, the full *BlobTree* information is traversed.

It has already been shown, that in the case of a *BlobTree* only containing affine transformation, leaving out step one can provide significant performance improvements, since the full traversal can be done only within the bottom-up part. Now that non-affine transformations are involved, a single leaf node does not have all the information in a bottom-up traversal situation to calculate the correct values yet. For this reason, a subset of the original *BlobTree*, the *Transformation Tree* can be used. It only contains the non-affine transformation nodes, to simplify the top-down traversal and transforms the input points and stores their respective results. Every leaf node stores a reference (index, offset, etc. depending on the implementation) to its closest non-affine parent node. The result of this non-affine transformation is then used as the input point for the bottom-up tree traversal, done without changing the algorithm described above. Figure 3.8 shows the tree of Figure 3.7 split into its *Transformation Tree* and the corresponding *BlobTree*. Their memory layouts are illustrated underneath, with memory traversals and lookups shown by arrows. The stippling of the arrows corresponds to Figure 3.7.

In the same way, any non-affine node stores a reference to its closest non-affine parent in order to pick the right input values in the case that several non-affine transformations occur along the path to a leaf. Linearizing this reduced *Transformation Tree* in depth-first-order into a contiguous array allows for a linear traversal algorithm. This traversal order ensures that at any node, the previous transformation node is already applied to the input points and the result can be read and used for the consecutive calculation. Applying the same data layout techniques described above for the bottom-up, field-value "gather" step, will also result in a cache-efficient data

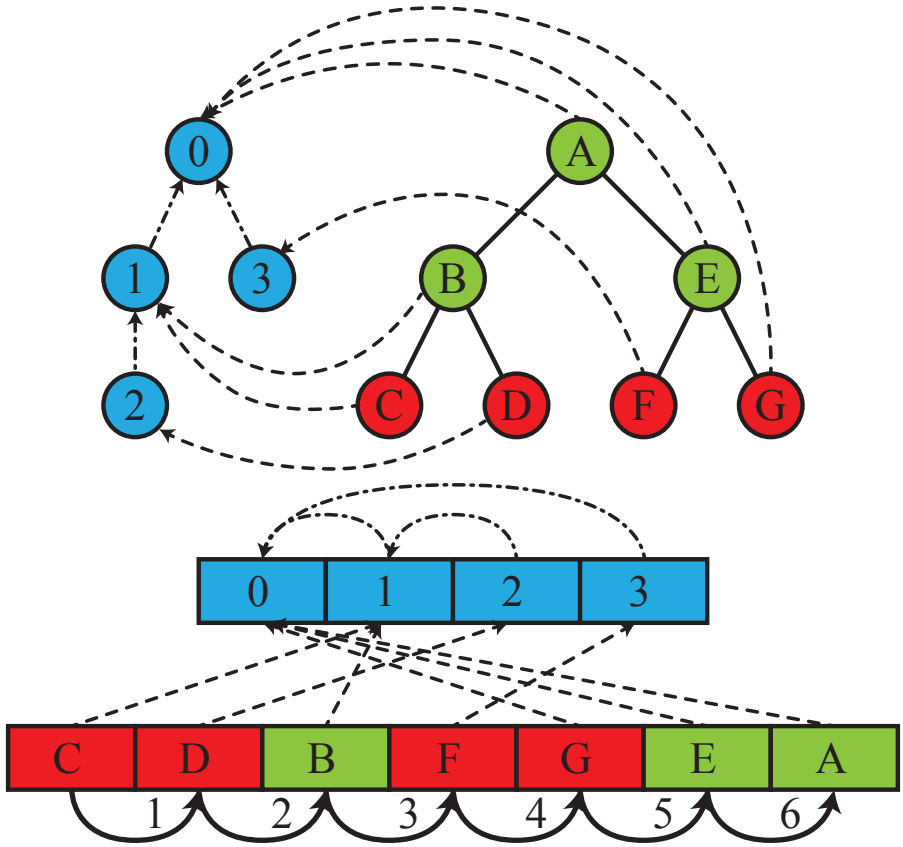


Figure 3.8: Separation of tree from Figure 3.7 into *Transformation Tree* and the *BlobTree* nodes, including their corresponding memory layout and access patterns.

layout to store the non-affine transformations, which, depending on their type, are more (Barr warps) or less (Warp Curves) trivial. In the same way, the metrics to optimize are memory usage, memory read direction and random memory access.

3.6 Implementation

The implementation of the algorithms use OpenCL 1.2 on MacOX 10.9.4, and the measurements are taken on an nVidia GeForce GTX 780M GPU .

As already stated in Section 3.4.2, the *BlobTree* is split into its structure, containing the links (for the top-down case only), and the actual tree data, containing the node information. The same node information data structures are used for both the top-down and bottom-up measurements. In order to support loading the *BlobTree* tree data into the current processing unit at a high speed, the size of a single cache-line

or the cache-line equivalent must be taken into account. To minimize the number of cycles it takes to load the nodes into variables, it is desirable that an integer number of tree nodes fit into a cache line.

The following data is stored for each node:

- type of the node,
- node-type specific data.

Since the node types are mutually exclusive, but are very close in size, they are stored as a union of types to make memory allocation, cache line alignment and array handling easier. The current size of the nodes are 64 bytes (an integer fraction of the cache line equivalent on the GPU). When the tree is traversed, node-type specific functions are called depending on the node type using a standard switch-case statement.

In the case of the base-line top-down traversal, the memory is stored as suggested by [Wald, 2004], where child nodes are stored next to each other (see Figure 3.2) to improve node locality and cache friendliness for ray-tracing. The advantage is that the data structure describing the tree structure only needs to store one index for the first child, and the second child can be found by incrementing this index, resulting in a 4 byte small value for every tree node. If needed, this approach can easily be extended to n-ary tree nodes as well, although in this work the tree is restricted to binary.

Because the field-value calculations are used by polygonization and ray-tracing, a single field-value calculation also returns additional information, similar to the *Tuple Tree* approach by [de Groot, 2008]:

- one float for the field value *BlobTree*
- two floats for gradient x and y. Assuming that gradients are normalized the z value can be omitted, since it can be recalculated from x and y.
- the sign of the z direction of the gradient, since gradients are stored in world space
- three bytes for storing the RGB colour values.

The main reason for this is that in general, the gradient and colour calculations rely on the field values at each node. Thus, calculating them in a separate step would require

a re-calculation of the field values at every node. Furthermore, the blend operator by [Gourmel et al., 2013] used in some of the performance cases blends based on the gradient at the child node as well.

A set of these values, a total size of 16 bytes, is stored for every intermediate calculation result. The number of intermediate results stored is dependent on the input model's tree structure (see Figure 3.5) and the chosen algorithm (top-down or bottom-up for comparison purposes). The size of the temporary stacks are calculated in advance (during linearization) and the required memory size is multiplied by the number of threads is allocated. Every thread is assigned a sub-block of the stack, so that the memory access between threads accesses parallel elements in the large block of memory. According to GPU vendors such as [AMD, 2011], this is one of the desired memory access patterns that increases performance. The top down approach has access to two memory blocks, one for each stack, with the traversal stack m having stack frames of size 4. For all the other approaches, the size of a t-stack frame is 16 bytes. In the implementation, the input values are stored in OpenCLs constant variable scope. Originally intermediate results (and the traversal stack when needed) should have been stored in local scope, since this is faster memory, but for many large *BlobTrees*, the local memory per thread was not big enough. As a result of these limitations, all the cases store the stacks in global memory for consistency reasons.

3.7 Results

3.7.1 Synthetic scene

For testing the performance of the algorithms and the improvements presented, a variable sized computer generated test scene is created. The model is created so that primitives will not overlap, which means that with an increasing number of leaf nodes, the volume occupied by the whole model increases. As a result, the density of primitives in the volume occupying the model is constant. This means that when the *BlobTree* is sampled at regular intervals, there is a good distribution of valid field-value results ($f > 0$) per primitive. It is expected that the distribution of *BlobTree* primitives will not have an impact on unaccelerated traversal algorithms, since all the tree branches have to be visited for the calculation. Once acceleration structures are added to the traversal algorithm the distribution of the primitives, and the structure

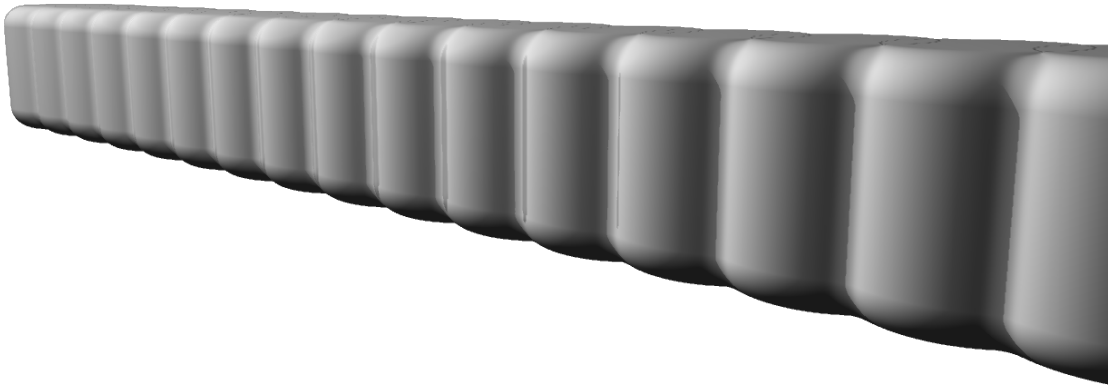


Figure 3.9: A sample test scene with several cylinders chained together.

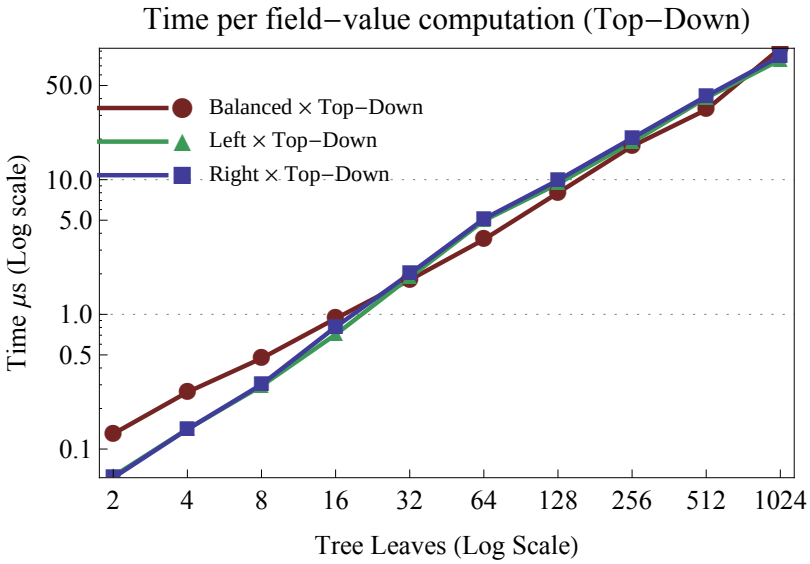


Figure 3.10: The running times for the computer-generated test scenes, traversed with the top down approach. Note that the "Left" and "Right" cases overlap.

of the tree will have a performance impact.

The synthetic model contains n_l primitives of the same type, all combined with the same type of binary operator, arranged to form a string of objects (see Figure 3.9 for an example with cylinders and an advanced blend operator[Gourmel et al., 2013]). To have a test case close to a real world scenario (polygonization), the bounding volume of this object is sampled 32^3 times along a regular grid, requiring the same number of stacks. Since the main focus of this algorithm is accelerating the tree traversal,

the rest of the polygonization algorithm is removed from the benchmark, since this is where previous work achieved their speed up. The same resolution is used for all the models and algorithms so that every test case has the same constant OpenCL scheduling overhead and the test cases between different sized trees are comparable. Every performance result is calculated as the average run time of 32 runs to get rid of outliers due to load spikes of other applications running or the OpenCL run-time optimizing the kernel based on the input data.

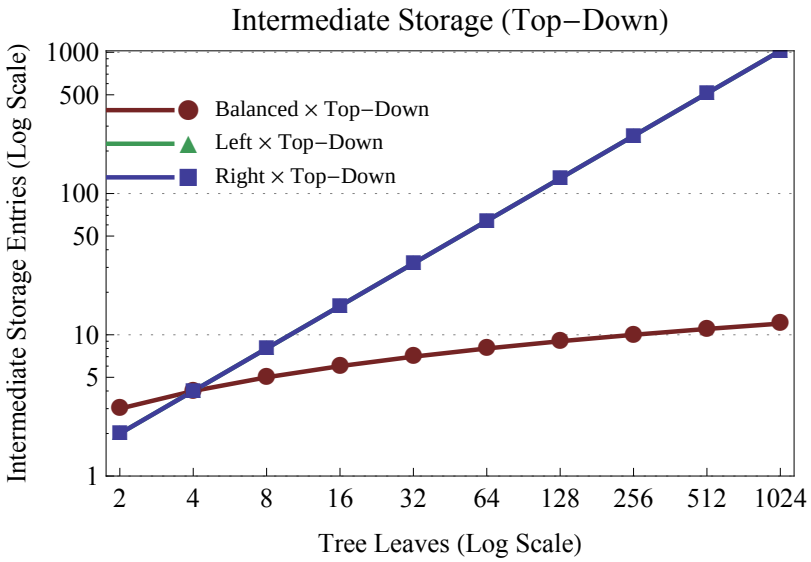


Figure 3.11: Number of temporary storage array entries for the three tree types and different numbers of leaf nodes. Note that the "Left" and "Right" cases overlap.

Top-Down Traversal

Two different types of graphs are shown for the computer-generated test scene: with an increasing number of leaf nodes, the average time in μs the algorithm spends in a single field-value calculation is plotted (including the tree traversal, and the actual evaluation of the nodes). In addition, each of the graphs is accompanied by the memory needed (plotted as the number of array entries) for the traversal to work. In this top-down case, this includes the emulated recursion stack frames. The intermediate results can still be stored on a smaller stack, which is independent of the recursion stack. See algorithm 2 for the implementation of both stacks.

Both graphs have their axes plotted using a logarithmic scale. Figure 3.10 shows that the left-heavy tree and the right-heavy tree have similar performance characteristics, with the balanced case being slightly faster. When this is compared to the

memory usage in Figure 3.11, it is easy to see why the performance of those two model cases are very similar: they need the same amount of memory, whereas the third one needs significantly less. In this case, the stack size stated is the size of the traversal stack m, since it is larger than t. This results in the balanced tree actually being preferable to any other tree representation in terms of storage needs and performance.

Bottom-Up Traversal

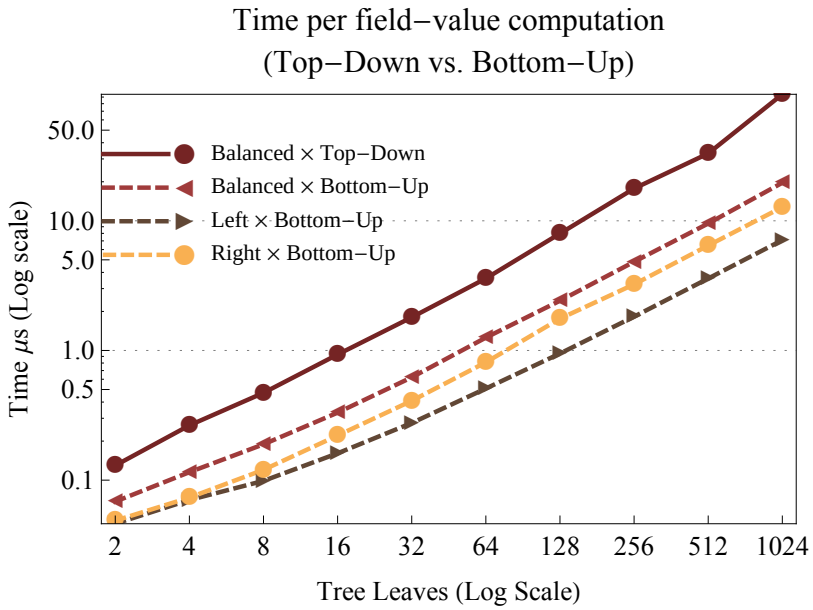


Figure 3.12: The running times for the bottom-up traversal algorithm, compared to the best top-down case as a reference.

Traversing the tree bottom-up creates a better memory access pattern, since every thread loads the tree array from start to end, potentially resulting in several memory load multicasts if threads try to access the same node at the same time. This approach does not require a stack for the tree-traversal recursion, as only the stack for the temporary results is needed. In this case, the number of intermediate-data stack frames are plotted in Figure 3.13. This is the size of stack t, as no m stack is needed any more.

Figure 3.12 includes the best case of Figure 3.10 (balanced) as a reference, and compares it to the run times for the bottom-up traversal. The best top-down case is the example of the balanced tree, requiring less storage for the recursion stack than the

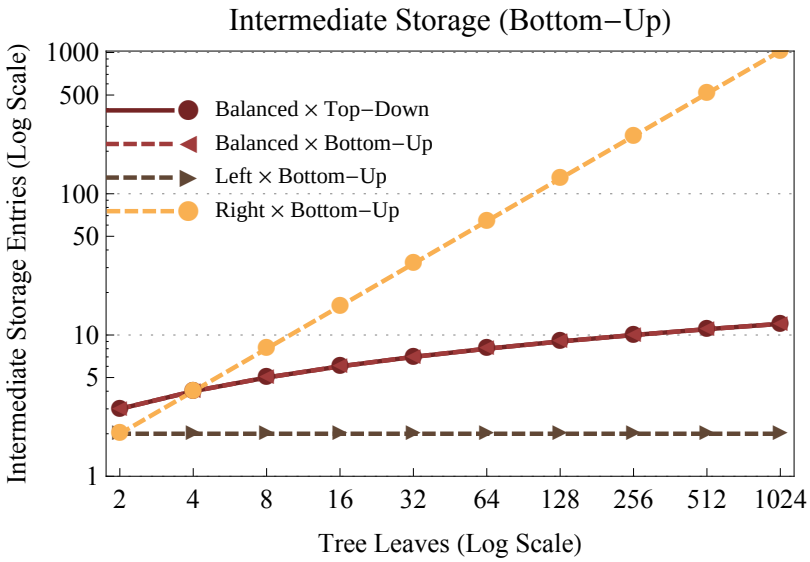


Figure 3.13: The memory usage for the bottom-up traversal algorithm, compared to the best top-down case. "Right Bottom-Up" and "Balanced Top-Down" overlap.

right-heavy tree in the bottom-up approach. Looking at this behaviour emphasizes that not just memory reads of the tree data are important for fast traversal, but that it is even more important to optimize the storage needs for intermediate results. On the other hand, the left-heavy tree case only needs a constant temporary result stack of 2 and shows the best performance of all of them.

However, despite the right-heavy variant needing the most amount of temporary storage entries, more specifically one storage entry for each leaf node, in this case it actually is the second fastest. The performance characteristics of this benchmark on different older hardware, an AMD Radeon HD 5870 (see Figure 3.14) shows that this hardware produces the run-time result directly dependent on the storage needs. One can assume that the newer nVidia GPU is better at predicting memory reads/writes (or at caching) for the following memory access pattern. The whole array is filled with temporary entries from left to right. Then, the direction changes and the array is read right to left. It seems that as long as the reading direction is not changed, the linear read works almost as efficiently as the case needing constant memory. In the latter scenario, the two storage entries will stay in the cache, assuming it is not filled by the other data involved in calculating the field-value, having repeated cheap read and writes to these two cache locations. This leads to the conclusion that one should favour the production of left-heavy trees for efficient traversal, as already shown for CSG by [Hable and Rossignac, 2005]. This produces a time difference of one order of

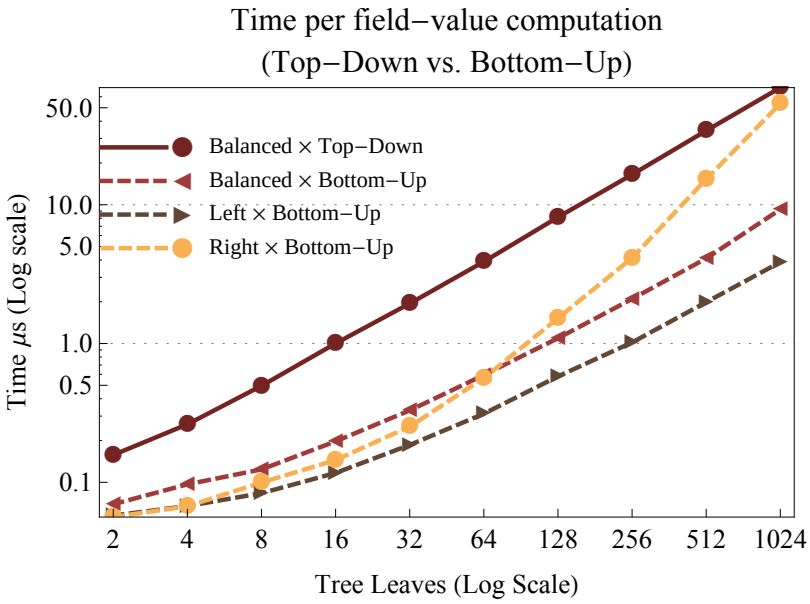


Figure 3.14: The running times for the bottom-up traversal algorithm, compared to the best top-down case, as run on an AMD Radeon HD 5870 GPU.

magnitude between the best top-down case and the best bottom-up.

Acceleration Structures

Rendering large models at interactive frame-rates on common hardware is often not possible due to their size and fidelity. Approaches to accelerate rendering operate on the principle of removing parts of objects that would not be seen from a current camera position and orientation, often referred to as *culling*. More generally, culling can be described as the process of removing/ignoring parts of an object that are not relevant for the current calculation. Acceleration structures operate on the principle of dividing the model into sub-parts. For rendering, traversing the acceleration structure determines the sub-parts of interest, so that only a sub-set of the whole model needs to be considered. Consequently, acceleration structures can be used to compare the performance of the Bottom-Up traversal algorithm (with and without an acceleration structure) to the accelerated Top-Down traversal.

Based on work by [Fox et al., 2001], the chosen acceleration structures are the Bounding Volume Hierarchy (BVH) [Rubin and Whitted, 1980] and Binary Space Partition (BSP) trees [Fuchs et al., 1980]. Both approaches recursively divide the volume surrounding an object into sub-volumes until a stop criterion is met (e.g. number of primitives in or certain size of the sub-volume). The BVH structure is

built from the leafs of the tree towards the root by combining the bounding volumes of the nodes. In the Top-Down traversal case, it is very easy to add a BVH to the traversal algorithm. The bounding boxes are already stored in world space for every tree node, so the only thing left is to add a point-in-bounding-volume check at line 5 in Algorithm 2.

In comparison, in the Bottom-Up traversal, the early discard property of a BVH cannot be used efficiently, since it is not possible to ignore sub-trees when traversing from the leaves up. An acceleration structure that prunes the *BlobTree* for each space subdivision leaf is the BSP Tree, which has been used widely to improve the visualization times of mesh scenes, resulting in real-time speed for raytracing on the CPU [Benthin, 2006]. A BSP Tree is built from the root node towards the leafs. At every build step the bounding box of the node is split into two, based on a given split strategy (for an evaluation of different split strategies see [Wald and Havran, 2006]). Each BSP-node only contains the parts of the model that are located within the node's volume, effectively building a sub-*BlobTree* for every BSP-leaf.

For the performance reasoning given by [Wald, 2004], axis aligned BSP trees (kD trees) are used, which are based on the implementation described. Since in this case only the kD node needs to be determined for a given point in space, there is no need to backtrack into neighbouring kD nodes, as demonstrated by [Popov et al., 2007]. To avoid storing duplicate nodes, the skip pointers from [Smits, 2005] are adapted, and index arrays for each kD leaf are created that work on the array storing the full linearized tree data. For the sake of simplicity only the best and worst running times of these two methods are compared with the previous, unaccelerated cases.

Figure 3.15 compares the four best cases of each algorithm(top-down, top-down plus BVH, bottom-up, bottom-up plus kD), showing two orders of magnitude difference between the worst of the best case algorithms and the absolute best. Adding an acceleration structure to any of the approaches changes the overall slope of the graph, whereas the pairs of accelerated and unaccelerated curves have approximately a parallel slope once the node count is higher than 16.

In the worst case, the difference in running time is even bigger. Figure 3.16 shows that the run-time of the worst case of the top-down approach for 1024 leaf nodes is close to $100\mu s$. On the contrary the worst run-time for the bottom-up traversal using a kD-tree is $0.19\mu s$, four orders of magnitude faster than top-down. Without the acceleration structures, there is still a one order of magnitude difference.

Figure 3.17 and Figure 3.18 compare the memory usage of the aforementioned

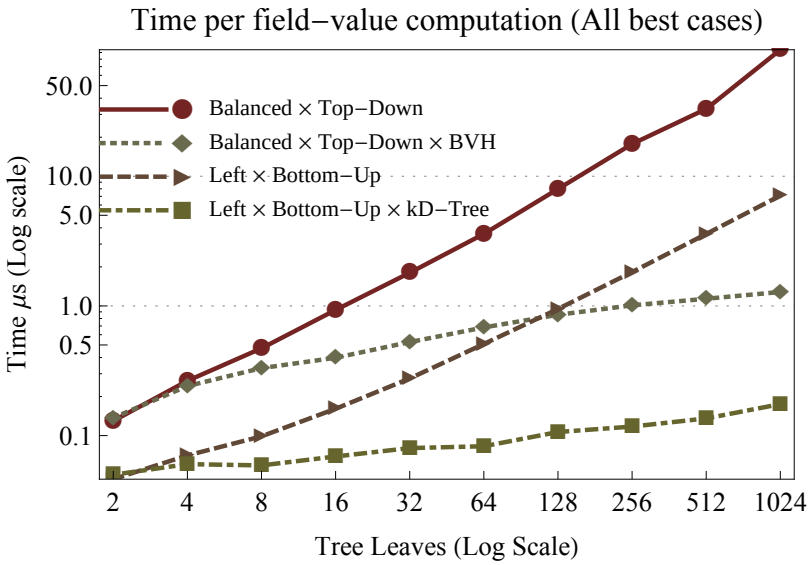


Figure 3.15: A comparison of the best case running times.

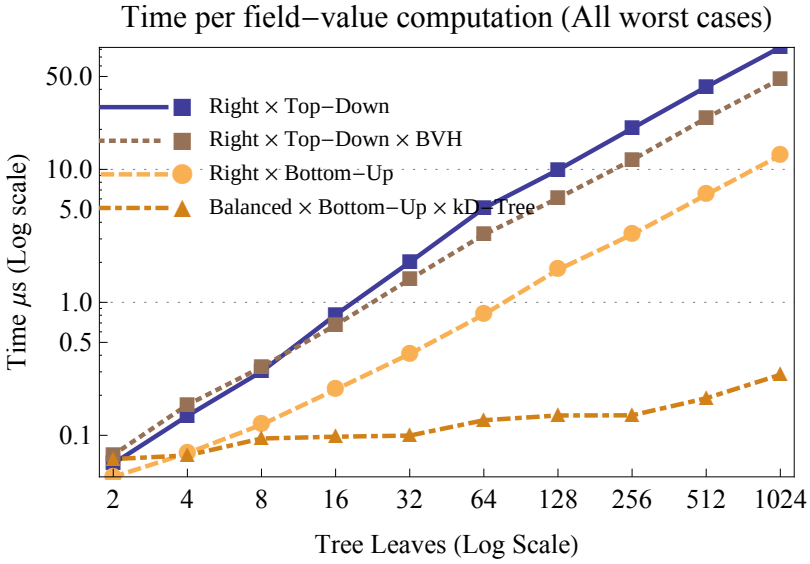


Figure 3.16: A comparison of the worst case running times.

time-based graphs. The trend here clearly is that the smaller the memory usage, the better performing the traversal algorithm. However, due to the special memory access pattern for a perfectly right-heavy tree, the bottom-up approach also performs almost as well as the best case. The size of trees that can be explored is limited by GPU memory, but one can assume that the top-down traversal will continue to increase its run time exponentially, whereas the bottom-up version, especially when using a kD-tree, will grow much slower. A top-down traversal of a solid model tree in SPMD

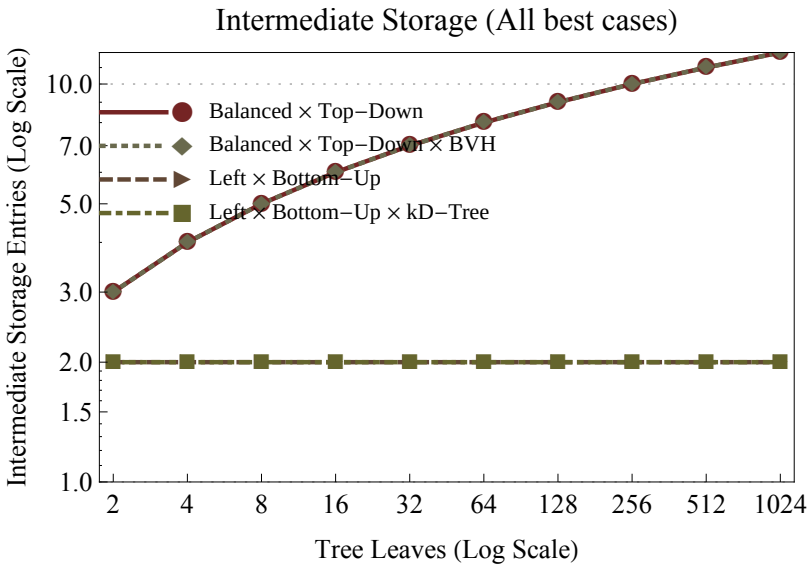


Figure 3.17: A comparison of the best case memory usage. Adding an acceleration structure does not change the memory usage for both traversal algorithms

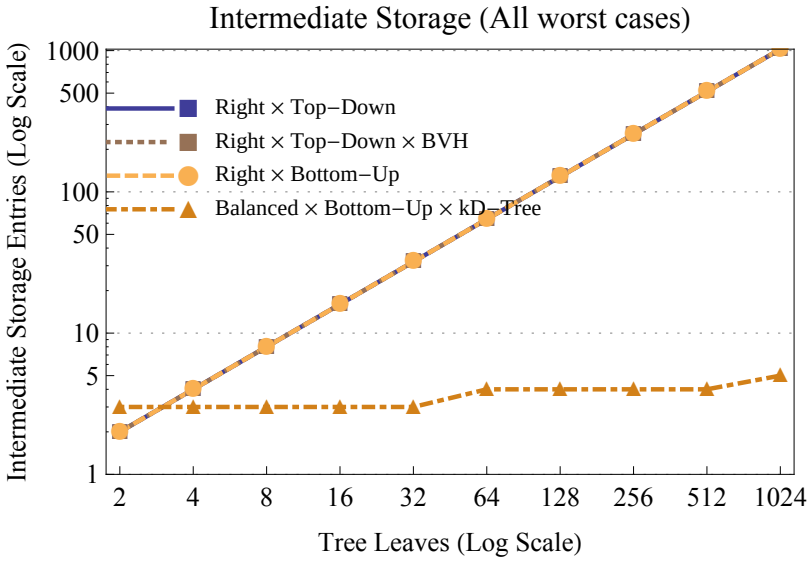


Figure 3.18: A comparison of the worst case memory usage. The two Top-Down cases and the unaccelerated Bottom-Up case have the same memory usage

is not the fastest. By modifying the traversal algorithm to use approaches presented for CSG [Rossignac, 2012], the traversal now works bottom-up. This improves the memory access pattern and usage and decreases the execution time of traversal algorithms significantly. With the top-down traversal, adding a BVH is fairly easy, and improves the algorithms run time; however, the trend of the performance still shows

an almost exponential slope (linear in a log graph), with increasing leaf nodes. The slope of the graph is a lot shallower for the bottom-up and especially the accelerated bottom-up algorithm, showing that it will lead to better performance in most cases.

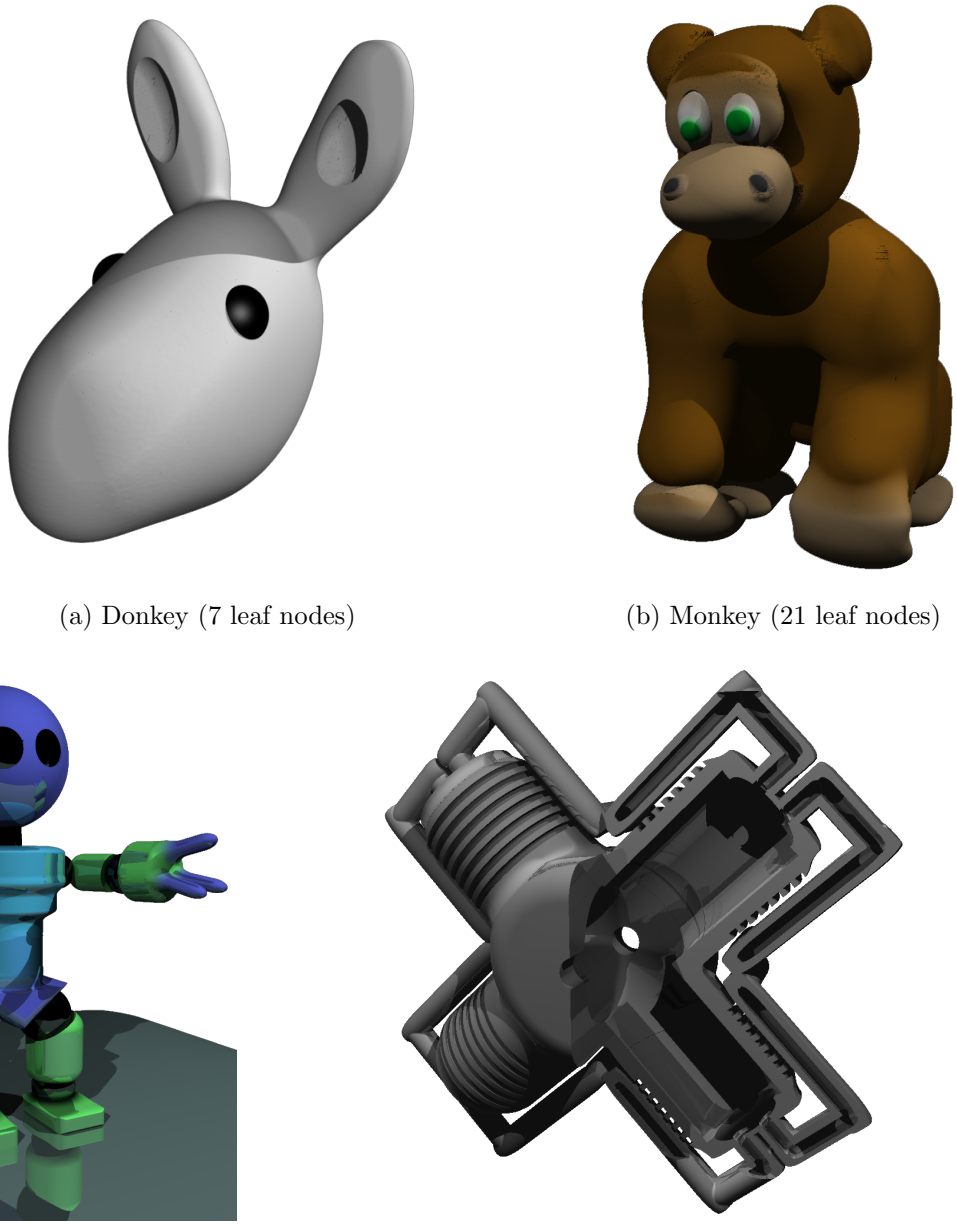
3.7.2 Models

The synthetic test scenes are built so that primitives are distributed equally in space, something very unlikely in real-world situations. While the synthetic models have a constant distribution of primitives in the surrounding bounding box, real world models will have their primitives clustered and distributed arbitrarily. Consequently, four real world models (see ray-traced images in Figure 3.19) of different sizes were used to investigate the performance of the Bottom-Up algorithm in a more realistic modelling scenario. The models were generated with an interactive modelling system that uses the Bottom-Up approach in the polygonizer. The two performance graphs, as explained for the synthetic models, can be found in Figure 3.20 and Figure 3.21. They show that the kD-tree is actually slightly slower than the basic bottom-up algorithm. One potential reason for this behaviour could be that the median split strategy for the kD-tree is not the best, as proven by [Wald, 2004]. Furthermore, since there is only a limited amount of splits (same maximum recursion as with the computer generated models), and all the objects are clustered at more or less the same point in space, the pruned trees contained in the leaf nodes will not be much more simplified than the original. As a result, more input data storage is needed without benefiting from the acceleration structure. It only adds traversal overhead to the running times.

Most of the trees used in the examples are very close to left-heavy; thus, adding this optimization did not add a significant performance impact to the models, especially the engine. The memory use for the left-heavy variants of the models stayed the same. Overall, the bottom-up traversal shows similar performance improvements with real-world models due to the reduced memory usage. Interestingly, these models don't benefit as much from the kD-tree as the synthetic *BlobTrees*.

These four models were chosen because of different distinct properties:

- The **donkey** model is built from a small number of nodes, but most of them are based on sketched primitives, combined using the summation blend. While the size of the tree is small, the sketched primitives need more time to compute than the cylinder primitives in the synthetic models of the previous section,



(a) Donkey (7 leaf nodes)

(b) Monkey (21 leaf nodes)

(c) Robot (37 leaf nodes)

(d) Engine Block (149 leaf nodes)

Figure 3.19: Three of the four real world models.

since they are dependent on the number of control points used to create the sketched field.

- Compared to the donkey, the **monkey** model is built from more sketched primitives that are combined using the same Gradient Based Blend used in the

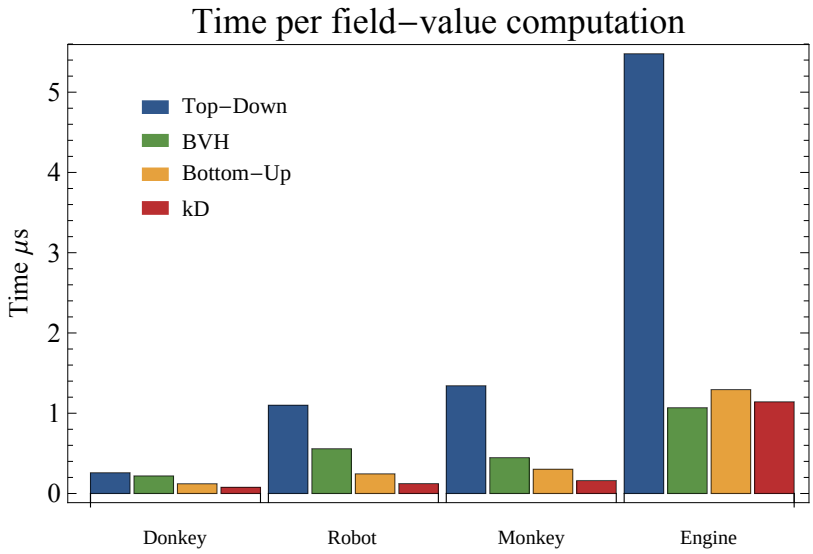


Figure 3.20: Average run times of a single field-value calculation using the algorithm variations for the four models, in μs.

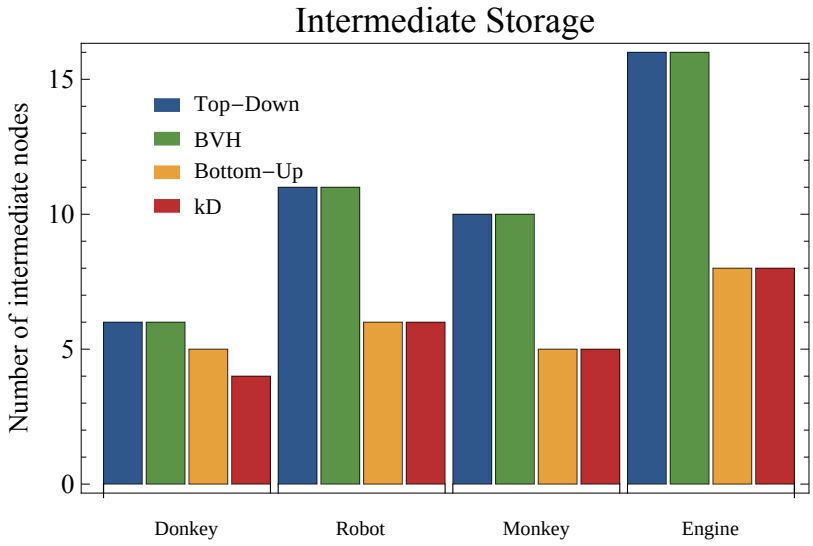


Figure 3.21: Intermediate storage entries for the four models.

synthetic case. There is a higher distribution of primitives in space compared to the donkey, which results in a better improvement of the accelerated test cases.

- The **robot** model uses a wider variety of *BlobTree* primitives and operators, with the primitives having a good distribution in space. As a result, this model can benefit from an acceleration structure similar to the monkey.

- Lastly, the **engine** model is the largest example model. It contains four large parts that are copies from each other, just positioned and rotated differently. Each of the four sub-*BlobTrees* are largely left-heavy trees which should show large performance improvements in the bottom-up case. Because these sub-*BlobTrees* are rotated, the axis-oriented bounding boxes, used in the acceleration approaches occupy larger regions of space, resulting in a non-optimal space subdivision. The performance graph shows that the bottom-up approach is almost as fast as the accelerated approaches. Because of the four similar trees the BVH case works very well at determining an early exit in the top-down traversal, whereas the kD-Tree case cannot improve performance as efficiently.

3.7.3 Non-Affine Transformations

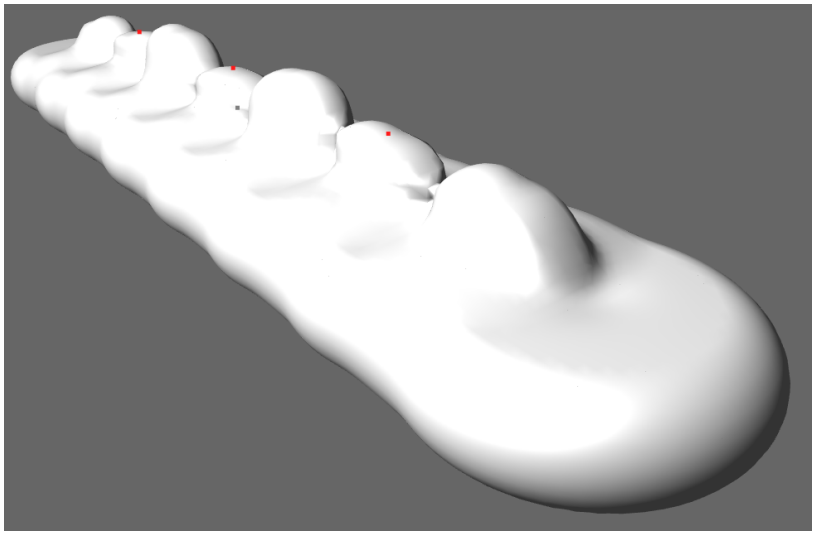


Figure 3.22: The warp-curve test scene. Notice the top-most three displaced points in red.

The performance test case for the inclusion of non-affine transformations in the tree is fairly similar to the synthetic test scene, where the different traversal methods are compared more generally. In this case, only the Top-Down traversal and the Bottom-Up traversal are compared. The models to be used for testing are based on the number of leaf nodes (primitives in the tree) but only for the balanced tree case. Above every blend node within this balanced tree, a WarpCurve [Sugihara et al., 2010] spanning the underlying model with three control points is inserted, where the middle one is displaced along the surface normal. The larger the number of leaf nodes, the larger

the number of interior operator nodes, which is equal to the number of warp curve nodes.

leaf nodes	Top-Down	Bottom-Up
2	1.007	0.821
4	2.027	1.839
8	4.364	3.985
16	8.990	8.018
32	17.259	16.317
64	34.640	32.772
128	70.156	64.634
256	140.729	129.534
512	279.599	260.756
1024	563.750	518.007
2048	2536.030	2360.930

Table 3.1: The performance numbers in μs, stating the average time for a single field-value calculation in μs for the test cases for *BlobTrees* that include various numbers of warp-curves.

Given that Disc primitives (slightly less expensive than the previously used Cylinder primitives) are combined using the Gradient Based Blend [Gourmel et al., 2013] (see Figure 3.22), the traversal times/calculation times for a single field value evaluation are fairly similar to the cases presented above. However, since each warp-curve has to calculate its values based on three thin plate splines, and there are $n/2$ warp curves in the test object, most of the calculation time is spent on the warp calculation. The same warp-curve calculations are done in both cases, so by looking at the difference in the traversal numbers, there is still a difference between the two traversal methods. As the warps need to be calculated for both cases, and the overhead of the warp calculation overshadows the traversal time in the first place, it can be seen, that the difference between Top-Down and Bottom-Up traversal is still there. However, most of the time is spent on calculating the warp, and not the tree traversal. Thus, the non-affine transformation case is computation bound, not memory bound as the previous cases were.

3.8 Conclusion and Future Work

Due to their mathematical differences, traversal of the *BlobTree* can't be optimized as much as a CSG tree. Simplifying the tree is not possible, since every tree node

needs to be evaluated and no short-circuit evaluation is possible for *BlobTree* operators. By investigating the memory usage of the traversal, and reducing its overall memory footprint, similar optimization techniques already applied in CSG rendering can be used to accelerate the *BlobTree* traversal. When reinterpreting the *BlobTree* as a mathematical expression and rewriting it in the reverse polish notation, the corresponding bottom-up tree traversal results in a performance improvement of one order of magnitude. Depending on the structure of the tree, additional performance improvements can be made by recursively swapping the child nodes of operators in order to make the tree left-heavy during the tree linearization process. Accelerating the bottom-up traversal with a kD-tree can result in an almost constant time tree-traversal.

Future work includes overcoming the memory limitations employed by current generation GPUs by further reducing the need for intermediate storage. This could require reordering the tree based on the operators to minimize temporary data stack push and pop operations. All current GPUs that support OpenCL have a region of very fast, but very small memory accessible from their compute units. Unfortunately, this fast memory could not be used for benchmarking since the temporary data stack does not fit into these memory regions. Once the required temporary memory size is reduced, the fast local memory can be used during traversal, resulting in additional performance gains.

An alternative to the presented data-drive approach would be a system similar to [Reiner et al., 2011], where new OpenCL kernels are compiled for the specific *BlobTree* configuration. Compiling the Bottom-Up traversal kernel needs several seconds, which would result in noticeable delays if such approach were to be used. A proper comparison, especially in terms of run times, is left to future work.

Chapter 4

CollabBlob: A Data-Efficient Collaborative Modelling Method using Websockets and the BlobTree for Over-the-Air Networks

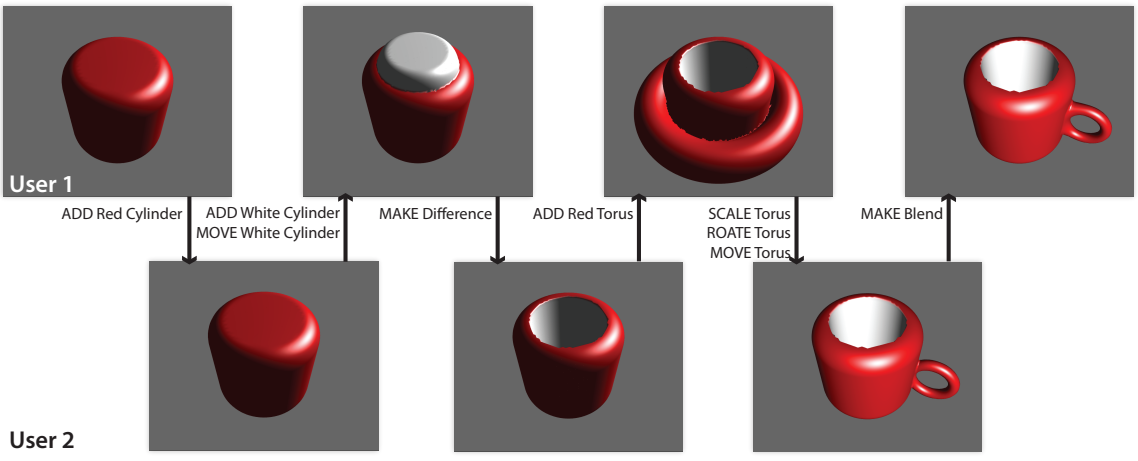


Figure 4.1: An example modelling session between two users.

Collaborative modelling has become more important in the past few years, especially now that mobile devices possess processing power to support 3D modelling in real-time. The problem with collaborative modelling using triangle meshes is that complex models are slow to synchronize and require large network resources, depend-

ing on the amount of data needed to update a model. Current mobile networks, such as 3G and LTE, unfortunately are not as fast as traditional wired internet and have higher latency. Synchronizing thousands of triangles wirelessly between all participating users can introduce a substantial lag between transactions, especially on wireless networks, making fine grained and rapid updates at interactive rates hard to achieve.

In contrast, the *BlobTree* is based on combining skeletal primitives and sketched-shapes using standard CSG and various blending operators. Using this methodology, complex models can be encoded with a smaller memory footprint than mesh-based systems, allowing for less traffic across a network to synchronize two or more workstations with one model. As a result, fine grained and rapid updates are possible, improving the visual communication between all participating users. *CollabBlob* demonstrates the advantages in interactivity and synchronization times over mesh modelling in a collaborative modelling environment.

4.1 Introduction

4.1.1 Motivation

CollabBlob is motivated by the desire to work collaboratively and share highly complex models across the network. With the introduction of touch-based tablet devices, a new way to share and model objects collaboratively using touch and sketch-based input is possible.

Outsourcing on a world wide basis, as studied by [Chu et al., 2006], is a good example of collaborative modelling in the industry. In this process companies rely on close collaborations between their suppliers and customers, often called Collaborative Product Development, where the protection of know-how (e.g. by hiding features) is very important [Chu et al., 2009].

Large models are very likely to be constructed by more than one person, particularly for product design where designers of different model-parts may be at different locations. The frame of a bike, for example, is sketched by a designer, whereas the linkages for the suspension are created by an engineer. Additional parts are added by another designer to create a final production rendering. Since network speed can be a limiting factor in collaborative design, one of the main criteria for *CollabBlob* is its small memory footprint and reduced amount of necessary synchronization messages. It provides the functionality to separate model features as a light-weight method for

access control.

4.1.2 Collaborative Modelling

In the WebGL [Parisi, 2012] strategy, the majority of the 3D geometric data shared and transmitted on the network is a polygon mesh. Mesh compression approaches and progressive meshes try to reduce the amount of information transferred, sometimes by reducing the overall quality of the mesh.

In comparison, *CollabBlob* minimizes network loads by transmitting updates to the hierarchical structure known as the *BlobTree* [Wyvill et al., 1999], where every participant receives the most precise description of the model. *CollabBlob* sends the information as typed messages with their associated parameters representing user modifications to the model. This strategy keeps the scene structures synchronized across multiple design stations. The *BlobTree* data-structure is modified by each participant using the commands, and visualization of the model is performed locally on each system using available processing resources.

4.1.3 Contributions

The contribution of this chapter is a synthesis of existing techniques from different disciplines. By using a hierarchical implicit modelling system network traffic is minimized and a sketch-based metaphor is used to directly manipulate a model. The implemented messaging system is server-less, does not require locks for synchronization and is based on Websockets [W3C, 2013]. Together, these improvement create a system contribution that can impact the way models are built in a collaborative environment as shown in Figure 4.1.

In situations where handheld devices are connected via 3G/4G networks, this method shows a big advantage over previous approaches: Fine grained and rapid updates of the scene are possible due to the *BlobTree*'s small memory footprint, enabling the system to be highly interactive. The approach is lock free and enables 'simultaneous modifications'. When several users want to change the same feature of a model in different ways, all of them can modify this feature (e.g. translate or rotate it) at the same time.

While *CollabBlob* presents a unique user interface to collaborative modelling, its main focus is to demonstrate the advantages of the *BlobTree* compared to mesh approaches in the context of collaborative modelling. For this reason, the current

user interface has not been evaluated in a user study. Improvements to the user interface and evaluating it are left to future work.

4.1.4 Outline

This chapter is organized as follows: The system of network messages is explained in Section 4.3.1, with an in-depth discussion of each message layer in Section 4.3.1, 4.3.1 and 4.3.1. Synchronization issues are discussed in Section 4.4 and the unique user interface features are explained in Section 4.5. The chapter continues with some example objects modelled using *CollabBlob*, including a discussion on the of data transmitted in Section 4.7. Finally, Section 4.8 concludes this chapter and future work is proposed in Section 4.9.

4.2 Related work

[Mouton et al., 2011] provides an in-depth analysis of current collaborative environments, mainly targeted to handle visual data sets. New applications should try to reduce their usage of bandwidth by using local client resources to increase an application's interactive performance. They advocate for a focus of new applications on transferring less data and calculating more information. In addition, they suggest that developers of new applications should try to use given standards instead inventing their own. The approach presented conforms with this idea, because it uses low bandwidth and uses HTML Websockets for transferring the information.

One early distributed virtual environment for engineering and manufacturing was CollabCAD [Mishra et al., 1997]. In this system, a mesh model is shared across the network among multiple designers. Previously designed models are imported for further manipulation and detailed modifications. Concurrent access to a common design is enabled for viewing and modification.

[Nishino et al., 1999] created a collaborative modelling environment to enable the design of implicit models. Each participant in the system can log into a session server to gain access to the part of the object being designed by other participants on that server. All session servers are managed by a centralized world server, which controls access rights and updates done by all participants. To make a modification to the model each participant requests an update right which is acknowledged by the session server. Each client holding an update right sends updated parameters to all

other participants connected to the same session server. Then it releases the update right and saves the tree data to the session server, not allowing simultaneous input from several nodes. *CollabBlob* uses the same basic idea of an implicit modelling system. It improves Nishino's system by using the *BlobTree*, which provides a variety of primitives including sketch-based shapes, CSG and different types of blending and deformations [Sugihara et al., 2010]. It avoids problems associated with having a central session server handling update rights (e.g. when the server fails) by offering a server-less system that allows multiple designers to make changes to the model simultaneously.

Many mesh-based approaches employ client-server architectures, such as the ones by [Han et al., 2003], [Ramani et al., 2003] and [Kim et al., 2006]. Meshes are transferred between the clients and the server, never between clients. All of these approaches use different ways to control access to parts of the model, all controlled by the central server. Approaches, such as the one presented by [Chu et al., 2009], try to compensate for the lack of synchronization speed of mesh approaches, by adding a mesh hierarchy or a level of detail method. One of their problems is that as the details and, therefore, the complexity of a model increase, updating the mesh hierarchy rapidly becomes the bottleneck in the system. This is in addition to the issues of client-server based approaches.

CollabBlob is based on Websocket [W3C, 2013], which is a standardized protocol to transfer messages across the internet, based on HTTP. [Marion and Jomier, 2012] use a Websocket implementation to transfer the scientific data to and between their clients. The data set is transferred when the program starts and only once it is finished, the collaboration process starts. In this process users can work on the data set concurrently, but the data set cannot be changed interactively. *Marion* highlights that the Websocket implementation can achieve lower latency and a higher synchronization rate than a comparable AJAX implementation.

Computer Supported Cooperative Work (CSCW) tries to find ways to present information to users in a collaborative environment and to optimize workflows between many users. The work by [Morris et al., 2004] identifies problems that occur in multi-user situations of cooperative system, especially when the number of users increases. Social protocol, while powerful for small groups, becomes less efficient in avoiding conflicts between a large number of participants. Sometimes actions by one user will change settings of others, or, for example, close documents that others are working on. Approaches, such as *Lark* [Tobiasz et al., 2009], try to overcome some

of the aforementioned issues by providing unique virtual views for every participant in the shared space. *CollabBlob*, in contrast, does not provide a single shared space, which avoids some of the issues found in co-located shared groupware systems. Social protocol, as discussed in more detail below, is still an important factor to provide a conflict free collaborative modelling session.

A different approach to collaborative work that is closely based on the workflow introduced by version control systems is *Branch-Explore-Merge* [McGrath et al., 2012]. It allows users to simultaneously work on the same data-set, however, there is the option of generating a local copy of the data (branch) to work independently from each other. Once the work task on the branch is completed the results need to be merged back to the shared space, in order to collaboratively finish a task. The option to branch and merge is not part of *CollabBlob* on purpose because it is designed so that all participants always see the current state of the model. Adding the capability for branching to *CollabBlob* requires that all branched models are shown in the viewport simultaneously. Visualizing all the branches at the same time poses its own set of challenges and is left to potential future work.

Distributed sketching has been a topic of interest in the CSCW and groupware community for a long time. While most reported systems are for simple 2D sketches, the human and social factors underlying distributed interaction apply equally to 3D modelling. These factors are perhaps best summarized by the mechanics of collaboration that cover the basic communication and coordination operations of teamwork - the small-scale actions and interactions that group members must carry out in order to collaborate within a shared workspace [Pinelle et al., 2003]. In brief:

- *Explicit communication* occurs not only through spoken and written messages, but by gestural messages, deictic references and actual actions that accompany talk (e.g., indicating, demonstrating, pointing, moving a pen to initial drawing, drawing actions).

- *Information gathering* includes fine-grained knowledge of what others are doing. This includes basic awareness (who is in the workspace, what they are doing, where they are working), feedthrough (changes to objects made by others), consequential communication (body position and location, gaze awareness).

- *Shared access* describes how people access tools and drawing objects, which covers how they reserve and obtain such resources, and how they protect their

work by (for example) monitoring others' actions in an area and negotiating access.

- *Transfer* covers how people physically handoff objects to others, and how they place objects in a space so others can use them.

Technically, *CollabBlob* require a few factors for the above to work in a real-time collaborative situation. Firstly, people need to communicate through words. This means a rich communication channel is necessary: in *CollabBlob*'s case, people are expected to use existing systems (e.g., telephones, VOIP, video conferencing) alongside while modelling. Secondly, people need to see rapid and fine-grained updates of the 3D sketch as it evolves, including transitional states that accompany object addition, deletion, movement, transformation, and so on. If delays are excessive, or if objects just 'shift' from one state to another without displaying in-between states, people have difficulty tracking what is going on, and have problems coordinating their talk with their sketching actions. This is the main motivation of *CollabBlob*: by using and transmitting only a small set of parameters, fine-grained and rapid updates are possible. Thirdly, people need to be embodied in the system in a way that others can see where they are, and what they are about to do. As common in most groupware, *CollabBlob* dos this through multiple cursors, implemented as arrows in 3D space and camera items implemented to show a miniature of the remote user's view of the scene.

One side effect of *CollabBlob* is that the construction history of the whole model is implicitly saved and can be used to replay the design process. Once the full history of a model is available, it can be used for demonstration purposes and even as a learning tool. *MeshFlow* [Denning et al., 2011] is a system that visualizes the construction of large mesh model. The history is filtered using a process which clusters operations within the construction history, and parts of the model can be a highlighted and annotated. As a result, MeshFlow can provide a good overview on the construction process and to focus important parts of the modelling process automatically. Because their approach is based on storing the full mesh at every step, the size of the history, while not stated explicitly, is likely to be bigger than in *CollabBlob*.

MixT [Chi et al., 2012] is a newer approach to generate tutorials automatically from a stored construction history. MixT is not limited to geometric modelling because it uses screen captures and input logs to generate the tutorial. By post-processing the videos, it generates hybrid tutorials that contain videos and static content (text and images). The reason for combining both types of tutorials is that

while video tutorials effectively describe interactions with the program, they are hard to navigate. Static tutorials, on the other hand, are the opposite in those respects. Consequently, MixT combines the advantages of both.

Another approach that uses the modelling history by [Chen et al., 2014] analyses the workflow to detect important regions of the model. These regions are then used as the highlights and focus regions of an interactive summary of the modelling process. Similar to MeshFlow, the system stores the model at discrete intervals, requiring significant amount of memory.

4.3 Implementation

Other groupware systems, as discussed in [Greenberg and Roseman, 1999], have dealt with the 'large model' problem in several ways. One common approach is screen sharing of single user applications: instead of sending the model, only the screen visuals are transmitted. Key limitations are that users have to take turns (simultaneous input does not really work as shown by [Nishino et al., 1999]), and that the model is not available at all sites for offline use. Another approach discussed by *Greenberg et. al.* transmits only user input, such as mouse movements, to keep the model synchronized, since as long as the input across applications remains synchronized, the models constructed at each site are the same. Such synchronization can be difficult in practice, and introduces the 'latecomer' problem. If a model has already been created ahead of time, either the entire model or the input stream up to that point have to be transmitted to bring the late entrant up to date. *CollabBlob*'s underlying data structure, the *BlobTree*, on the other hand has the advantage of a compact representation, even for a large model (see the airplane model in Figure 4.10d), so even sending the whole history does not involve a lot of data transfer.

4.3.1 Network Message Layers

To satisfy the aforementioned properties for efficient collaborative systems, *CollabBlob* maintains a true copy of the model(s) across all clients. For these cases *CollabBlob* is based on a parameterized approach that includes a protocol, which can be categorized into several layers, each of them dealing with separate parts of the required communication:

- *system* messages, described in Section 4.3.1

- *actions* described, described in Section 4.3.1
- *user interface* messages, described in Section 4.3.1

CollabBlob is a synthesis of techniques from different disciplines as the basis to overcome problems present in several existing distributed modelling environments. In this system, no node is a dedicated server, so the need for an election algorithm in case the server loses connection is not present. Every node connects to every other node and all messages are sent via multicast to all participating members. Each message apart from a system message contains its sending time stamp relative to the start time of the modelling session. These time stamps are mainly needed for synchronization, but they also directly provide one of the additional benefits of the message system, described in Section 4.7.1.

Message types differ in the way they are applied:

- System messages are executed right away when they are read by each host.
- Actions and user interface messages are buffered in between the rendered frames. The buffers are updated at the start of each frame to avoid unnecessary work between frames, and potentially not having the data changed while rendering is in progress. This reduces the computation workload, since the program polygonizes at maximum once every frame.

System Messages

CollabBlob uses Lamport timestamps [Lamport, 1978] to provide concurrency between all nodes, as described in Section 4.4. One main objective of the system messages is that all users use the same time base in the messages sent. These time stamps are in *coordinated universal time* (UTC) and every node in the system myst be synchronized to a local time server.

When a new node A connects to one of the nodes B currently in the modelling session, B sends the start time of the session to A. In case node A already has modelled something, the local model is reset and the remote one is loaded. The other main objective of the system messages are handling of all connected users. After A connects to B, B gathers the IP addresses of all its connected nodes and forwards them to A. A starts connections with all the nodes whose data it receives, and confirms to B when this is achieved. Then A receives the action history (see below) of the current modelling session from B to create the model and participate in the

session. If one single node loses connection to the system and reconnects, the same procedure applies. The reconnection is handled as if it is a new node connecting, discarding the old information on the reconnected node.

Actions

In *CollabBlob*, the term *action* is used for any network message that changes the current shared model. This means that after an action is received and applied, the actual model is changed. A user interface message, on the other hand, is used for immediate feedback and only shows an approximation of the future change.

Since actions modify the actual model, the following different types of *BlobTree* data objects are defined:

- *primitive objects* with their parameters (e.g. colour),
- *sketched objects* with the parameters (as above) plus the sample points,
- *operator objects* with optional parameters e.g. the *Ricci Blend Operator* and
- *transformations* (standard affine transformation or warps, bends and taper nodes).

Primitive objects and sketch objects are leaf nodes in the *BlobTree*, whereby transformations have one child node and operators usually have two child nodes. For primitives, sketched objects and operators, there is only a limited set of potential values (e.g. a sphere or a cube primitive, etc.), so the main information is given by setting the exact data objects using its explicit type information. If needed, a limited set of additional unique parameters (such as transformation values) is transmitted as well. Every action creating a new node in the *BlobTree* gets a unique time stamp, which is part of the message, and actions operating on the existing nodes (operators and transformations) take these IDs as parameters as well. This information can be seen as the minimal representation needed to describe an arbitrary *BlobTree* in the system: node type information, IDs and additional parameters.

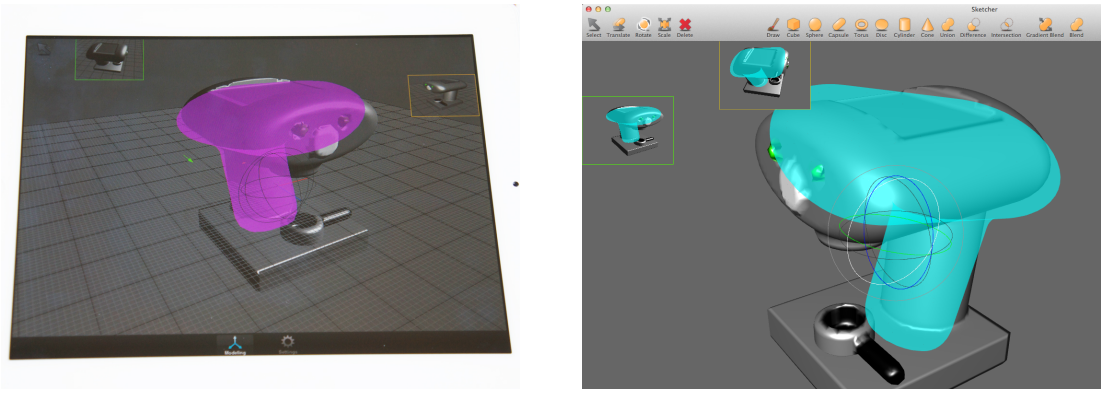
When a model is created, it can be thought of as a series of semantically different tasks:

- **add primitive** and set its parameters,
- **sketch object** based on a given control polygon,

- **add operator** combining several nodes of the tree that have parameters depending on the operator,
- **move**, **scale**, **rotate** and **delete** a *BlobTree* node and all its underlying children if present,
- **undo** and **redo** of any action

The actions defined above are independent of their actual implementation in a user interface. For example the delete action can be either triggered by a button click in an application having a CAD like interface or it can be triggered by the user directly using a gesture, in this case crossing out the object in a sketch interface (as in [Schmidt and Wyvill, 2005a]).

User Interface Messages



(a) iPad (b) OSX

Figure 4.2: An example modelling session between three users. Both, the iPad (left) and the desktop (right) application show the users, and the modification about to happen (initiated on the desktop).

A major advantage of *CollabBlob* is that the *BlobTree* data structures transmitted are small, enabling fine-grained and rapid updates of the scene. This approach also allows to transmit what the users are doing when they are not applying changes to the *BlobTree*. In order to achieve this immediate feedback of *what is going to happen* after the remote user finishes his current task, several messages to describe user input are incorporated into the system. *CollabBlob* uses these messages to update several distinct, non -*BlobTree*- related data:

- camera parameters to have information about everybody's point of view,
- cursor positions to show where remote users are pointing at in 3D space and
- immediate feedback showing the result of a geometric transformation

To avoid unnecessary immediate steps, the *BlobTree* itself is only changed and re-polygonized when the action for the final modification is sent.

Without these messages, changes to the model would just appear at every participant's computer when they are applied to the tree, without any previous feedback. An example of this behaviour would be an object being transported from one location to another. This feedback is needed to communicate changes between all participants in the modelling session. When a user adds a new shape to the scene using sketching, the control points of the sketched shape are transmitted as they are drawn, so the other users are informed at every stage of the drawing process. The information transmitted in the user interface messages does not modify the final tree, because the necessary transformation data is sent separately. This is done so it is possible to discard all the user interface messages when saving the final model actions, and still have all the necessary data to reconstruct the model.

4.4 Synchronization

CollabBlob's synchronization approach is based on *optimistic time stamp ordering*, as described by [Kung and Robinson, 1981]. The timestamps are transferred in relative time in microseconds since the session's start time, assuming that all the participating users have working clocks that are synchronized via their operating systems. Every action is assigned a time stamp by the originating host system and applied at every participant ordered by the time stamps. In case a *latecomer* message arrives and messages with a later timestamp have already been applied, the latter are rolled back, the latecomer message is inserted and all following ones are reapplied.

As defined in optimistic time stamp ordering, there is a chance of actions conflicting, which in this case can be:

- A node that is already a child node in the tree cannot be made a child node again, as it would have two parent nodes. If such an action occurs, it will be ignored and the originating user informed about it.

- A node that has been deleted cannot be altered. Similar to the above scenario, such a message will be ignored and reported too.

- In case different users modify the same part of the model, the messaging system decides if a potential collision has occurred. A global parameter to the modelling session is the minimum time between actions on the same node by different users. In case actions are too close, they are chosen on a first-come-first-served basis, and others are discarded. In *CollabBlob*'s implementation, a time frame of one second has proved to work well, since it can be assumed that an action will be applied at all the other nodes within this time and can be visually registered by all participants.

Every new node is assigned a unique ID using the standard UNIX uuid generator. This generator uses a combination of the local mac address and the timestamp to generate a 128bit wide ID that is considered unique [ISO, 1996]. An action creating a new node contains this ID and, as a result, it is easy to identify the same nodes across the network.

Messages in *CollabBlob* do not need to be acknowledged, but when they need to be ignored as a result of a conflict, a message is sent to make sure all other nodes ignore this action too. In these cases, changes are not applied to the model as requested, and users see the model automatically roll back the conflict.

It is expected that the number of conflicting actions can increase with the number of participants in one small modelling session. In such a situation it is very likely that people try to modify objects at the same time. A solution to this problem in a more sophisticated production system would be the capability to actively lock parts of the model, or implement a similar approach as *Branch-Explore-Merge*. In the latter case, users could branch parts of the model, and modify them without other people's interference. The branching results can be shared and demonstrated in the common workspace and voting mechanisms can be used to decide if a branch can be merged back for everybody to continue.

4.5 A Collaborative User Interface

The three types of user interface messages (camera parameters, cursor positions, intermediate transformations) are:

- Camera parameters are used to present the model from the point of view of the other users. These parameters are used to render the scene as seen by the other users into a texture. It is then used as the *interactive avatar* for the specific user, displayed as a screen aligned quad at the 3D position of the remote camera. If the remote user's camera is outside the current viewing frustum, the screen aligned quad is clipped to the frustum borders, so it is always present. If necessary, the *virtual camera* view can be enlarged by clicking on the avatar. This approach has also proven useful if multiple views of the same model are required.

- The 3D cursor positions of the users are visualized within the scene. If the user is not pointing at any object in the scene, the 3D position is at a constant distance along the view ray of the remote user. An arrow is used to visualize this 3D cursor, with the tip of the arrow being the position transmitted. Its orientation corresponds to the remote camera. In case the remote user is currently sketching a new shape, the transmitted 3D sketch control points are visualized, describing the control polygon of the part of the sketch already drawn. When the message to end the sketch action is received, the control polygon is removed from the screen, since it will soon be replaced by the actual sketched object.

- The intermediate transformation results are displayed using the same visuals that are used for transformations done by the local user. Depending on the type of transformations, certain widgets are displayed at the centre of the current *BlobTree* node. Widgets used locally display an active transformation mode depending on the chosen motion. Since the desired motion for incoming remote transformations is set by the remote user, only a shadow of the widget is displayed to illustrate that the current user has no control over the motion. A shadow of the node is also displayed when it is moved to convey the current position of the object to every participant. Otherwise, the object would simply be ported from one spot to another without actually illustrating who did it, and when the transformation was started.

Figure 4.2 illustrates the above mentioned features, shown for both the desktop and the mobile application. There are two additional users present in the modelling session. On the desktop, both the yellow and the green users look at the scene from their viewpoints, the mouse cursors hidden from the model in the main view. The desktop user interacts with the rotation widget (circles in grey), transforming the

rotation of the highlighted object. On the main display of the mobile device, the feedback of the translation of the main part of the coffeemaker (highlighted in pink) is transformed via the translation widget (original position in grey, the actual position shown in colour).

The main design mantra when the User Interface of *CollabBlob* was created was that it should be familiar to any user of CAD or other CSG-like modellers. Only modelling actions that are not yet known in standard modelling applications should use their already known sketch-based modelling method of input. This results in standard modelling behaviour, unless sketched shapes are involved. Normal primitives are placed by mouse click/finger tap, whereas any primitive or operator that is sketched is created by drawing the desired shape.

4.5.1 Transformation Gizmos

In general, the gizmos support modifying the transformation state of the selected sub-*BlobTree* and show interactive feedback to the user, with the option of restricting the transformation to the three main axes. Additionally, combinations of two of the axes, and a "free" transformation along all three is possible as well, in case it makes sense (ie. this is possible for the translation and scale, but not for rotation). On the other hand, the rotation widget also supports rotation within the current view plane, as defined through the camera orientation.

For every gizmo, two visual representations are used to visually highlight the change in the transformation:

- the current state (in colour)
- the origin state, when moving (in grey).

In addition, when the gizmo is static, immediate feedback about the potential transformation direction is highlighted, if the user would start the transformation based on the current cursor input. For example, when it is within a certain epsilon of the translation gizmo's origin, the whole gizmo is highlighted to indicate a movement along all three axes, whereby the highlighted region between two axes indicates the movement along both axes. If only one axis is highlighted, only this movement will be executed as soon as the movement is started.

In order for the gizmos to work on touchscreen devices, there is a certain epsilon region around the arrow tips or the handles of the other transformation gizmos, to

simplify usage. Since the touchscreen API on iOS devices already filters the input to provide an accurate "finger" position, the epsilon region size is the same as in the desktop case. It has to be mentioned that the pre-movement "hover" feedback found on the desktop is not shown on touchscreen devices due to a lack of reliable hovering capabilities on touchscreens.

The gizmos are also used to illustrate the interactions of the other participants. Depending on wether the gizmo represents a local or remote user, it is rendered in a different colour. While a local gizmo shows both the current state (in colours) and the origin state (in grey), the remote gizmo only shows the current state, and is rendered in grey to help differentiating.

Translation

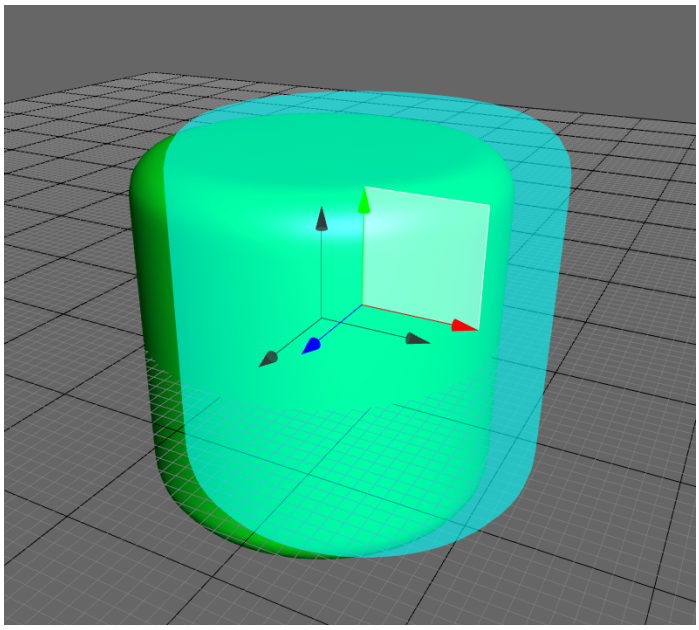


Figure 4.3: The translation gizmo used, providing interactive feedback.

The translation gizmo, as shown in Figure 4.3, is mainly built from three arrows for each main axis, with each of them colour coded as in any other well known modelling package (3ds Max, Maya, blender, etc.). By clicking the area surrounding the origin, the object is moved along all three axes, parallel to the view plane, whereby clicking on the arrow tips starts movement in on of the main directions. When hovering in the area between two axes, motion orthogonal to the third axis can be chosen. Independent from the object's rotation, the three axes are always displayed in world

orientation.

Scale

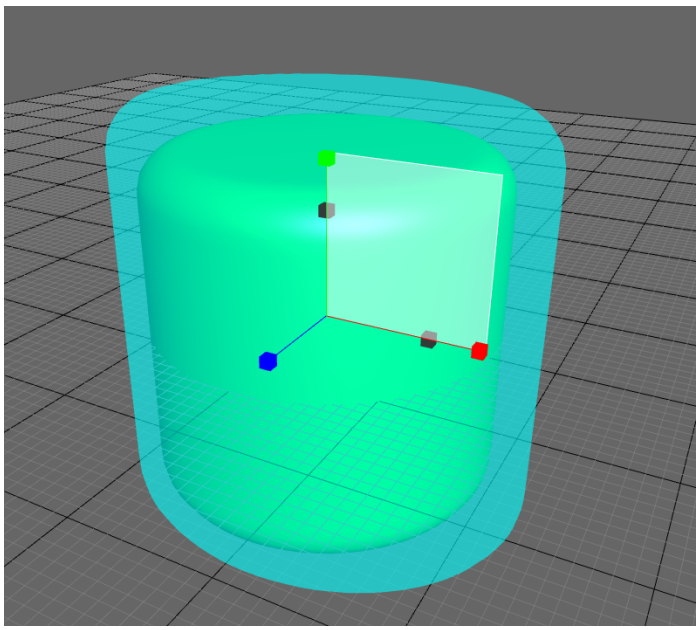


Figure 4.4: The scale gizmo used, providing interactive feedback.

Similar to the translation gizmo, the scale gizmo consists of three main axes, with handles at each end. To differentiate them from the arrows of the translation gizmo, the handles in this case are cubes to conform with some known modelling packages, such as Maya. The decision on which axes are transformed is done in the same way as in the case of the translation gizmo. In order to apply a uniform scale, the gizmo movement has to be initialized around the gizmo's origin.

Whereas the translation gizmo uses absolute movement, in the scale case, the amount of scale is based on the scale gizmo size change. This means that if an axis is elongated to twice the original length, the object's scale is doubled along this axis. In the opposite case, when the scale is done so that the handle is at half the original length, the applied scale factor is 0.5. Once the scale is applied to the object, the gizmo reverts back to its original measurements, whereby the object's size stays the same. The reason for this is that the gizmo is aways displayed at the same absolute size, independent of the actual scale of the object. In addition, as in Figure 4.4, the interactive feedback only shows the moving part of the gizmo, and the axes that do not change the object's scale do not render any feedback.

Rotation

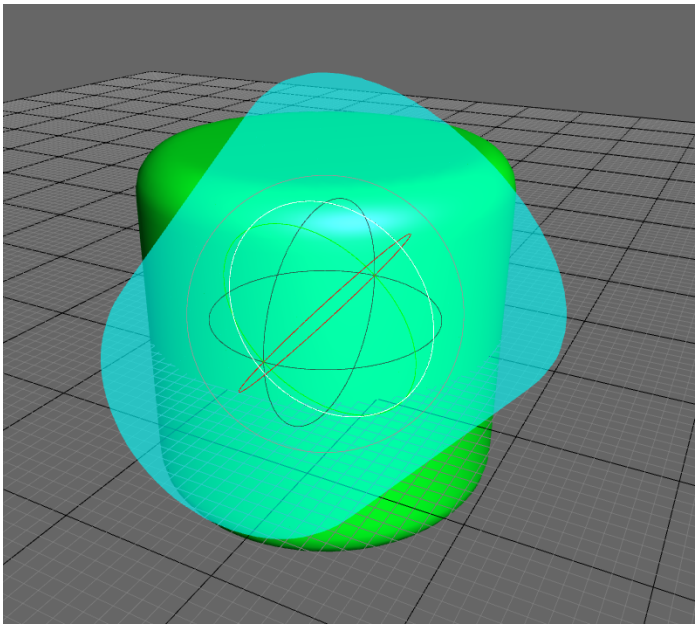


Figure 4.5: The rotation gizmo used, providing interactive feedback.

When controlling the rotation of an object through the gizmo, the rotation is applied in the same relative way as it is done with the translation gizmo. The rotation axis is determined via the three coloured circles, or in the case of a rotation within the view plane, the white outer circle. By moving the mouse cursor or the finger along the chosen circle, the rotation is applied to the model. If the mouse cursor is only moved from left to right, a half rotation around the given axis can be applied. For a full rotation, the full circle has to be "drawn" using the input methodology of choice. in case of a moving gizmo, the rotation axis is shown in white, with the remaining axes rendered in their specific colour based on the interactive motion feedback. This shows that the other two axes are transformed according to the selected motion. Once the user input has ended, the axes are again rendered in world space, not the object space.

4.6 Access Control

Building a complex model, such as a car, often involves creating several disjoint parts that might be built by different specialists. In some cases, it is desirable to have both of the models displayed together to see if they fit. For this reason, *CollabBlob* supports

several *BlobTrees*. The same unique identifiers as for tree nodes (see Section 4.4) are used to identify the trees in the system. In order to assign each action to the proper tree, these IDs are transmitted with each action. If no tree with the given ID is found in the local modelling session, a new one is created and gets assigned this ID. Each user chooses his current active tree and is allowed to switch at any time in the session. Any action the user takes can only apply to the current selected *BlobTree*, resulting in a lightweight access control system.

Assuming a working communication channel is in place, the designers and engineers can coordinate which *BlobTree* can be altered by whom. New *BlobTrees* can be added as needed and are displayed as half transparent until selected by the local user. This is done so that they don't obstruct the view of the current active *BlobTree* and to illustrate clearly which objects can be altered.

This lightweight access control system can be extended if needed by introducing formal access control based on users and user groups, similar to the systems described in Section 4.2. Every node in the *BlobTree* stores ownership information that can be used to restrict access to the specific node or subtree in the *BlobTree* to either a single user or a group.

Whereas the previous work described (e.g. [Han et al., 2003]) uses a central server managing access control, a similar mechanism could potentially cause problems if used in *CollabBlob*. If, for example, a user/group locks specific parts of the scene, and disconnects, the locked part will remain locked. Potentially, this problem can be solved by introducing timeouts to every lock, but this means that locks would have to be renewed regularly, resulting in potentially unnecessary communication overhead. Because of this problem *CollabBlob* does not use centralized locking and leaves a better locking mechanism for future work.

As mentioned above, a lightweight access control mechanism is implemented by splitting the whole scene into several smaller *BlobTrees*, which results in decreased visualization time, since a change requires repolygonization of only the changed *BlobTree*. If the same scene consisted of a single *BlobTree* with disjoint parts, a change in one disjoint part would require repolygonization all other disjoint parts. This specific problem has been solved by [Schmidt et al., 2005b], so a combination of both approaches can still result in fast visualization times. This forms another example of the advantage of maintaining a true copy of the *BlobTree*.

4.7 Results

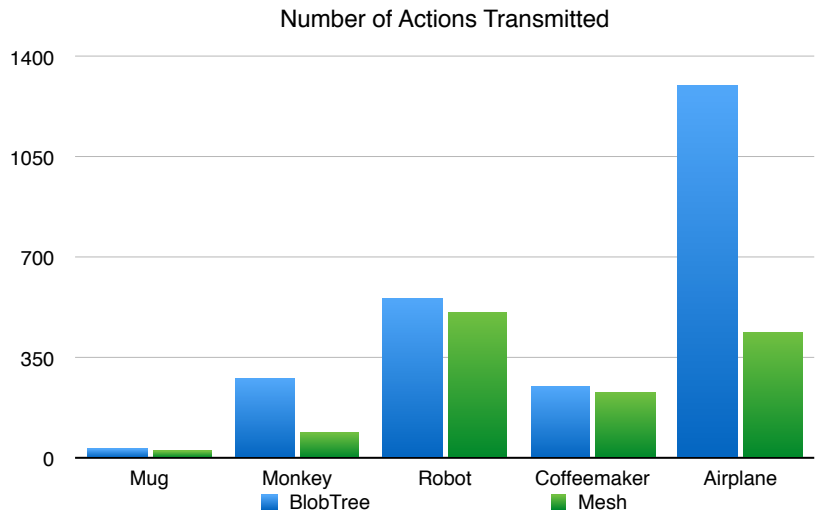


Figure 4.6: Comparison between the number of actions transmitted in the *BlobTree* case and the Mesh case.

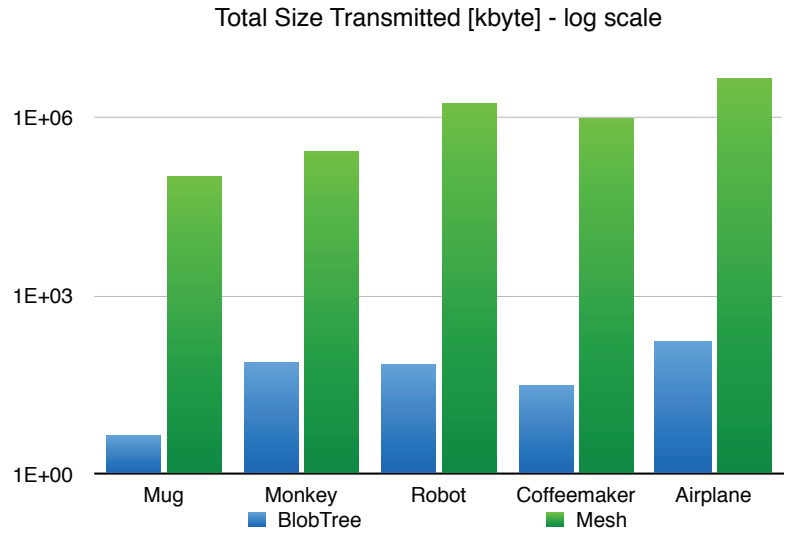


Figure 4.7: Comparison between the total size of memory transmitted in the *BlobTree* case and the Mesh case. The y axis is in log scale.

To provide quantitative data of *CollabBlob*, the modelling characteristics are compared against a mesh based synchronization method. The latter method sends the

mesh at every modelling step that requires the model to be changed to each participant. Several modelling sessions with results of different complexity (shown in Figure 4.10) are compared. This evaluation does not include user interface messages, which are assumed to be the same in both approaches. As a result, more actions are transmitted for the *BlobTree* case than for the mesh case, as shown in Figure 4.6.

In addition to the number of actions, the total size of data transferred (Figure 4.7), time spent transferring this data (latency) (Figure 4.8), and the average time to send a message updating the model (Figure 4.9) are measured. In the test an average speed 3G network, with 420 kbps uplink and 850 kbps downlink, is simulated.

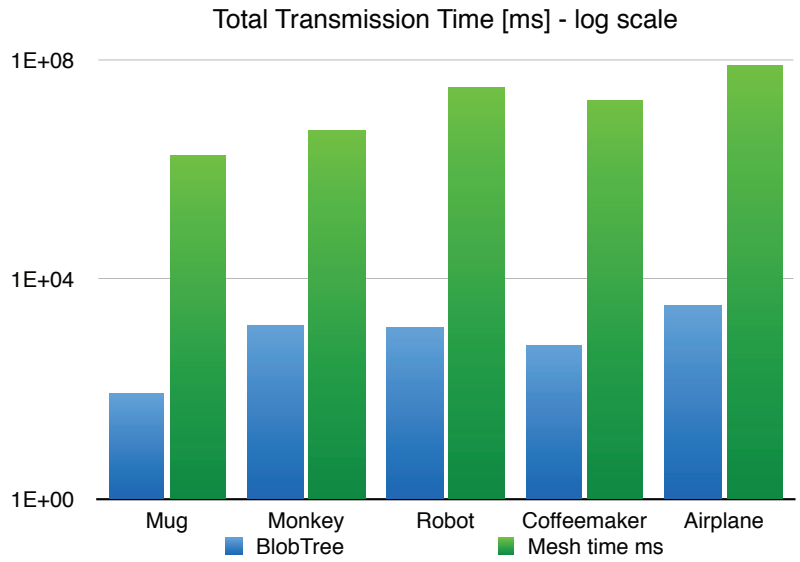


Figure 4.8: The total time spent transmitting the data. The y axis is in log scale.

For the mesh case, there are usually less messages sent, since the mesh will be regenerated at maximum once per frame, reflecting the changes of several *BlobTree* actions. Nevertheless, the mesh approach uses a significantly higher amount of data, resulting in longer transfer times between the participants. As a result, interactivity slows down significantly, since the average time for a mesh action lies in the 100 second range. The *BlobTree* approach, on the other hand, has an worst case action time of 5 milliseconds. The larger the model, the greater the size of the mesh, increasing the average transfer time. In the case of the *BlobTree*, the message size is independent of the size of the model, as it encodes only the changes in the tree. Sketched objects have a larger message sizes due to the variable number of control points (e.g. the monkey model, which has many sketch actions and fewer geometric primitives).

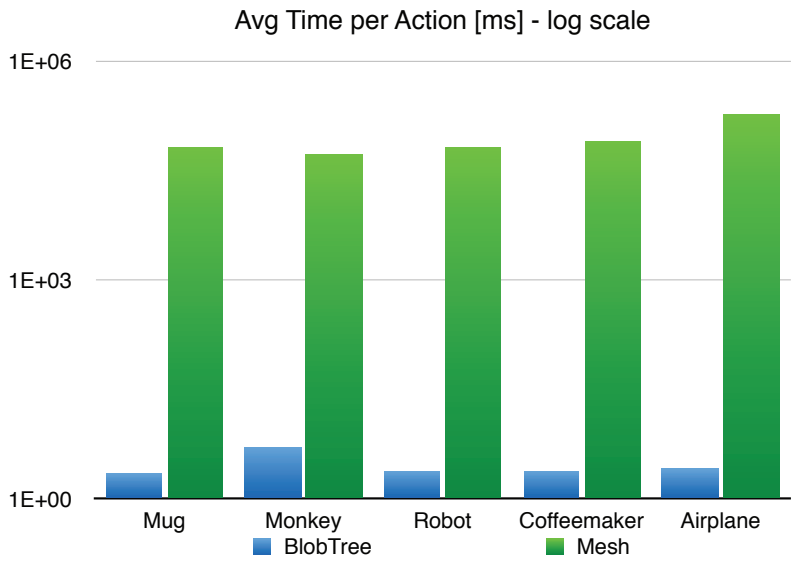


Figure 4.9: The average time spent transmitting a single action. The y axis is in log scale.

4.7.1 Construction History

There are several advantages of storing the whole construction history instead of storing only the final model. First of all, by saving the construction history of the different parts of the model on hand, the model can be recreated at each step. If a model is highly complex, or in case unnecessary parts were inserted, the actions building certain parts can be filtered out, to simplify the model. The full history is relatively small for *BlobTrees* and keeping it does not degrade the system.

Since *CollabBlob* also transmits user input and time stamps, it is possible to playback the whole construction of the model, either in real time, or similar to a video player, with changed speed. Several modelling communities teach modelling by using *video* tutorials that usually require considerable storage space and bandwidth. Compared to videos, *CollabBlob* needs significantly less storage, even if accompanied by an audio stream, commenting the construction history. If the producer of such a tutorial realizes that something undesired was done during the recording process, a video editing software is needed to alter the recording. If, in comparison, *CollabBlob* is used, the undesired messages can be removed using a text editor.

If errors or undesired changes in the final model are found, *CollabBlob* provides a simple way to determine the user responsible for that part of the model. Since every

(a) Coffeemaker (41 nodes)

(b) Monkey (64 nodes)

(c) Robot (119 nodes)

(d) Airplane (810 nodes)

Figure 4.10: The four models used for the performance comparisons.

message can be traced to its origin, all that needs to be done is to find the message causing the undesired model in the history and determine its sender.

Similar methods as *MeshFlow* [Denning et al., 2011] and *MixT* [Chi et al., 2012] can be based on the *BlobTree* construction history. The tutorials/construction sequences generated by such approaches are more advanced than a simple replay of the construction history. By using a history generated by *CollabBlob*, however, the storage requirements of these sequences can be significantly smaller than their mesh or video based versions. As a result, distributing the tutorials becomes more efficient.

4.8 Conclusion

CollabBlob is a system based on the *BlobTree* that allows collaborative sketch-based modelling across a network. The network traffic is minimized by using a hierarchical implicit modelling system. A sketch based metaphor is used for direct manipulation of a model, and a layered server-less messaging system that does not require locks for synchronization. This distributed system uses different layers of messages to distinguish between synchronization and setup (*system messages*), immediate user interface feedback (*Ui messages*) and messages that alter the model(s) under construction (*actions*).

The *CollabBlob* application was used to build the four models presented in Figure 4.10, and to illustrate the advantages of *CollabBlob*: reduced size of transmitted data between all users and optimistic time stamp ordering to avoid a lock-based synchronization approach. Future work will explore the relationship between model complexity and the use of the message system as described, as well as a detailed comparison with a mesh approach. No collaborative approach in the literature was found that only transfers the change in the mesh. Although this idea would reduce the bandwidth for communications, a large *BlobTree* can more efficiently encode details that would require far more data even in the incremental mesh case.

Actions can be recorded for training purposes and also for reviewing the steps that have been done to design a part of an object. To control access, a light weight system where disjoint parts of the model can be separated is integrated, and every person can only work on one tree, not several at the same time. This reduces the chance of people adding model information to the wrong parts during the session. Apart from the ability to allow for fine grained and rapid updates between all users in the current modelling session *CollabBlob* has several other advantages. It improves on the

most similar system from [Nishino et al., 1999], in that *CollabBlob* includes a wider variety of primitives and operators. Moreover, there is no need for a centralized server managing the scene and access rights and as a result does not have the problems related to this approach. This enables all users to simultaneously access a variety of alternative shape modifications and collaboratively choose the most appropriate result.

4.9 Future Work

The future work targets a more complex access control mechanism, similar to the one described by [Han et al., 2003], where the amount of details revealed for each participant can be controlled by roles defined in the system. Potentially, such a system can be built by extending the access control system presented to configure if messages per model section will be transmitted to other participants. In case the need arises to share models at a later stage, the history of the sub part can be manually sent to the collaborator.

Another topic for future work is improved handling of conflicting actions. *CollabBlob* does not allow conflicting actions within a certain time frame and as a result there will always be one ground truth of the model shared between all collaborators. Other approaches, such as [McGrath et al., 2012], have shown that often users like to apply their own set of modifications to the task on hand, without other people interfering. In the case of collaborative modelling this can happen when different solutions are desired by different participants. While *CollabBlob* provides methods to do this when only one incremental modification is done differently by several participants, it does not support the same for a series of changes. Being able to branch the model in the shared workspace will create additional challenges related to the visualization of the branched models. Additionally, this approach requires a form of merging the changes based on social protocols [Morris et al., 2004].

The approach to embodiment implemented using the $3D$ arrows as cursors, and the remote views will clutter the whole workspace once many users are involved in the modelling session. In addition, rendering many views on current mobile devices can have a performance impact resulting in a non responsive interface. Adding the option to turn off the remote views (or the widgets/gizmos) of other users will declutter such a situation, and potentially improve render times on mobile devices. Additionally, the $3D$ cursor might be replaced by the modelling mode currently selected by the

users, in order to extend the deictic pointing metaphor based on the research by [Wong and Gutwin, 2014].

CollabBlob uses coloured borders of the remote views to distinguish the remote users participating in the current modelling session. While in theory this allows a large number of users having unique colours, in practice this is limited by human vision. Consequently, future work should include an alternative way to distinguish between users.

Finally, future work includes a proper evaluation of the user interface. *CollabBlob*'s main objective is to demonstrate the advantages of using the compact description the *BlobTree* provides for large models over meshes. As a result the main focus of this work is the underlying transmission and synchronization system. The current user interface of *CollabBlob* is largely based on a non-collaborative *BlobTree* modelling system, and the models presented were built by people familiar with both *BlobTree* modelling and the application. In order to evaluate the user interface with its avatars and cursors, additional people need to use *CollabBlob* to finish given tasks. The users evaluating the system should be from two groups, familiar with *BlobTree* modelling and unfamiliar with Collaborative work, and vice versa. Each group will be introduced to the topic they are unfamiliar with, so that the objectives can be completed. Because *CollabBlob* runs on desktop and mobile devices, there will be tasks on each platform. The collection of the user data is already present, when the user-interface messages are stored within the scene. Consequently, different metrics can be extracted from the history, e.g. the number of conflicts or which type of actions often lead to conflicts. Another metric especially interesting for the modelling workflow is how often similar transformations (e.g. translations) are applied to the model in quick succession, since this can provide information on how easy it is to specify a desired operation. In addition to the metrics, direct feedback from the study participants will be important too.

Chapter 5

Angle-Based Filleting: Adding CSG-like control to *BlobTree* primitives

5.1 Introduction

5.1.1 Motivation

Filleting in the context of solid modelling describes the process of creating a controllable and smooth transition between surfaces. Often, the result of such a filleting operator (a fillet) is also called a bevel or a chamfer, however, fillet is the more general term for the resulting rounded edge. Figure 5.1 shows the different appearances of a cylinder top cap, depending on whether the surrounding edge is straight or filleted.

Many CAD applications contain this feature because there are several reasons why an artist or an engineer might want to model a fillet:

- **functional constraints**: An example for this would be the region where the bonnet of a car transitions into the fender.
- **aesthetic needs** of a designer: For example, the regions between the inlets and the main cartridge housing of a water tap.
- the **side-effects of a manufacturing process**: For example, the drill size of a CNC mill or a weld bead.

Figure 5.1: Interpolation between a straight edge along a cylinder top cap (right) and its filleted version (left).

It has been shown in the past that smooth transitions between surfaces can be created with little effort in the *BlobTree*. Additionally, transitions between surfaces in the *BlobTree* are smooth per default, so compared to problem of rounding an edge in CSG, the *BlobTree* lacks the control over straightening an already round corner while preserving the field continuity. This chapter discusses how filleting in the *BlobTree* context can be improved by giving the user control over the resulting surface.

5.1.2 Approaches to Filleting

Many different approaches to create these transition surfaces exist, often involving complex mathematical equations that insert surfaces into a CSG tree, such as the canal surfaces, described in detail by [Peternell and Pottmann, 1997], rolling-ball solutions (e.g. [Whited and Rossignac, 2009]), or surfaces described by sweeping spline-based cross sections [Elber, 2005]. These approaches have in common that additional surfaces are inserted into the CSG tree, replacing primitive boundaries created through the standard CSG operators. This is needed, as it is not possible to directly define such a surface using a "filleting operator" within the CSG tree due to the "binary" space classification every CSG primitive is based on. The *BlobTree* already provides limited capabilities to define smooth transitions between surfaces, especially of different primitives (known as *blending*), using the methods described by [Barthe et al., 2004], [Bernhardt et al., 2010] and[Gourmel et al., 2013], which all

use the implicit field to define such a surface, thereby fitting seamlessly into the *BlobTree* modelling paradigm. Little work has been done to introduce controllable filleting for surfaces defined within the *BlobTree*-primitives [Grasberger et al., 2010]. Whereas this approach based on modifying the difference calculation that the single primitives are based on tries to control the range between sharp edges and the standard *BlobTree* representation using interpolation, a scheme based on the filleting angle would be more desirable for the user, as this corresponds to the methodologies already available to CSG users.

5.1.3 Contributions

The main contribution of this chapter is *Angle-Based Filleting*, an extension to the *BlobTree* to support the specification of fillets using an opening angle. *Angle-Based Filleting* also gives a user the same tools to specify fillets that are already known from CAD programs. Compared to CSG approaches, *Angle-Based Filleting* provides a simpler mathematical formulation of the fillets. As a result of the generated field's C^2 continuity, the filleting of fillets is supported and will not cause the issues found in other solid modelling approaches [Middleditch and Sears, 1985]. The approach is based on the blending operators published in [Barthe et al., 2004] and [Gourmel et al., 2013] and, consequently, it seamlessly integrates with already existing *BlobTree* operations.

5.1.4 Outline

After giving an overview of related work in Section 5.2, the problems involved in filleting of primitive surfaces are outlined in Section 5.3. The user is guided through applying an extended implicit field calculation to the known skeleton primitives. Section 5.4 starts with the simplest case, where a single radius fillet is applied to one edge of a primitive. An approach called *Surface Fillet Curve*, where arbitrary edges can be drawn onto shapes to be used as the basis for filleting is demonstrated in Section 5.5 and a method for variable filleting along one edge in Section 5.6. Section 5.8 shows how filleting can be used to improve models without adding to their *BlobTree* complexity. *Angle-Based Filleting* is summarized in the Conclusion of this chapter, found in Section 5.9, and potential improvements are suggested in Section 5.10.

5.2 Related Work

According to [Middleditch and Sears, 1985] filletings of two surfaces have to satisfy three important properties:

1. tangency with the base surface to be blended
2. curves of tangency with the base surfaces which are a constant distant (sic) from the alternate base surface
3. cross section profiles defined and controlled by the 'shape' of the profile curve and the distance of the tangency curves from the alternate base surface.

For some surface configurations, the resulting polynomial filleting surface based on the above conditions can have an order of up to 23 [Middleditch and Sears, 1985]. This restricts the approach to be used in non-interactive modelling environments only. Mathematically simpler filleting operators have been developed as a result (see below), but the three aforementioned properties are still important.

The approach by [Elber, 2005], despite being fundamentally different, as it is based on defining the filleting surfaces' cross section using parametric curves, still tries to satisfy the above-mentioned conditions by allowing continuity up to C^2. This tangency is required to provide smooth shading along the surface boundaries between the surfaces to be filleted and the fillet itself. Given that this approach allows for any transition that can be described by the polynomial cross section, the constant distance requirement mentioned above is not satisfied.

Within the domain of implicit modelling, approaches based on functional representations ("F-Reps") attempt to solve the filleting problem (e.g. [Pasko et al., 1995] and [Adzhiev et al., 1999]). These approaches allow (controllable) blending between all objects and can control the influence region of blends [Pasko et al., 2005]. In the case of two F-Rep planes, the radius of the blend can be controlled, creating a fillet that matches the results of CAD approaches, as shown in Figure 5.2.

Both [Rossignac and Requicha, 1984] and [Hoffmann and Hopcroft, 1985] advocate blends/fillets with circular cross sections and provide methods to create them. These blends are very much related to canal surfaces [Peternell and Pottmann, 1997] and their surface boundaries. Canal surfaces are created by rolling a ball that remains in tangential contact with the adjacent surfaces. [Chen and Hoffmann, 1993] present an approach where the ball has constant radius, while [Whited and Rossignac, 2009]

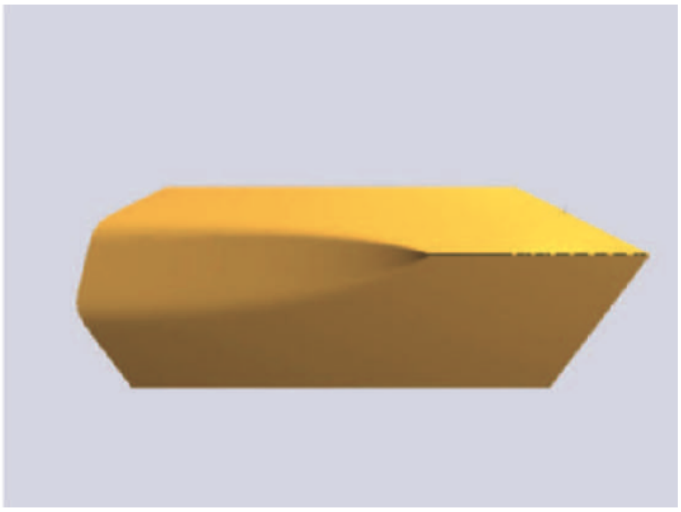


Figure 5.2: Bounded Blending showing how a blend can transition into a straight edge. (Image from [Pasko et al., 2005]).

incorporate varying radii for the ball used. Despite these surfaces being of lower complexity than [Middleditch and Sears, 1985], they can still have a high order. This complicates their evaluation in an interactive modelling environment or renderer, as there can be numerical stability issues and the interval arithmetic methods used to evaluate them are also costly.

In the case of *BlobTree* modelling, filleting can be split up into two domains:

- filleting between primitives (usually called "blending" in the *BlobTree* context)
- filleting between surfaces of a single primitive.

[Grasberger et al., 2010] proposed filleting the edges of *BlobTree* primitives by blending between a straight edge implementation of a *BlobTree* primitive and the standard *BlobTree* representation. This, however, creates two problems. First of all, the field of the straight edge primitive is discontinuous in the inside region of the primitives involved, as illustrated by Figure 5.3. Secondly the straight edge is also present in the interpolated versions (see Figure 5.4), and does not resemble an increase in the sphere/filleting radius when compared to a rolling ball solution in CSG. Ideally, the straight edge would stop immediately once the fillet has a radius $r > 0$.

Angle-Based Filleting is based on the controllable blend operators defined by [Barthe et al., 2004] and [Gourmel et al., 2013]. In [Barthe et al., 2004], blending is done based on the arc of an ellipse, which would actually allow for filleting based

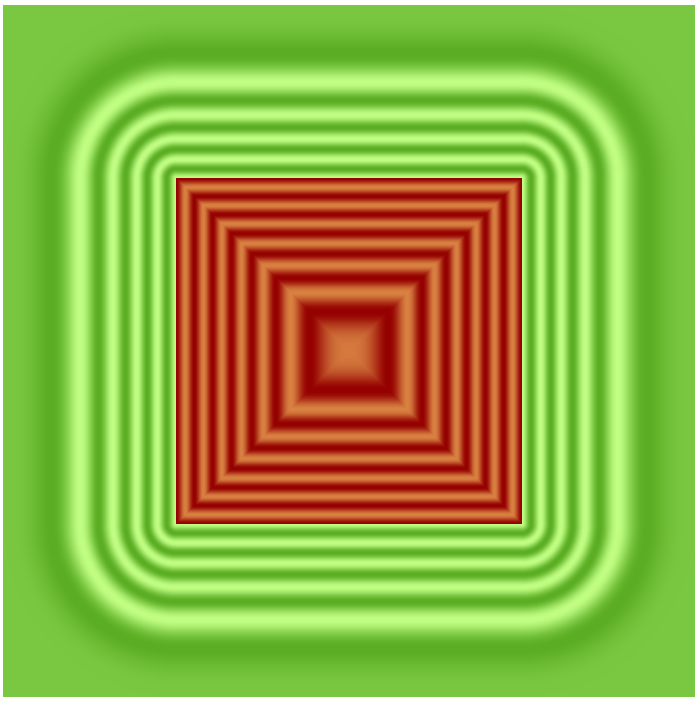


Figure 5.3: Cross-section of the discontinuous inside field (red) to calculate a straight edge (Image from [Grasberger et al., 2010]).

on a rolling ellipsoid. The problem with the ellipsoid-based blend approach, however, is that the so-called *clean union* has to be calculated using complex numbers. [Gourmel et al., 2013] restrict these operators to be based on a function close to the arc of a circle, which is symmetric and close to a rolling ball. This results in a clean union operator that is significantly less complex to compute.

For these reasons, *Angle-Based Filleting*'s method of filleting within primitives is based on the math described in [Gourmel et al., 2013], as it is the most powerful blending operator and C^2 continuous. In addition *Angle-Based Filleting* is used in conjunction with the Gradient Based Blend operator (see Section 2.6), so that the fields generated by fillets and blends are similar. A sophisticated implementation, as a result, can be transparent to the user whether a fillet is created between the surfaces of a *BlobTree* primitive (outlined in this chapter) or between the *BlobTree* primitives (shown in related work on blending in Section 2.4 and 2.6).

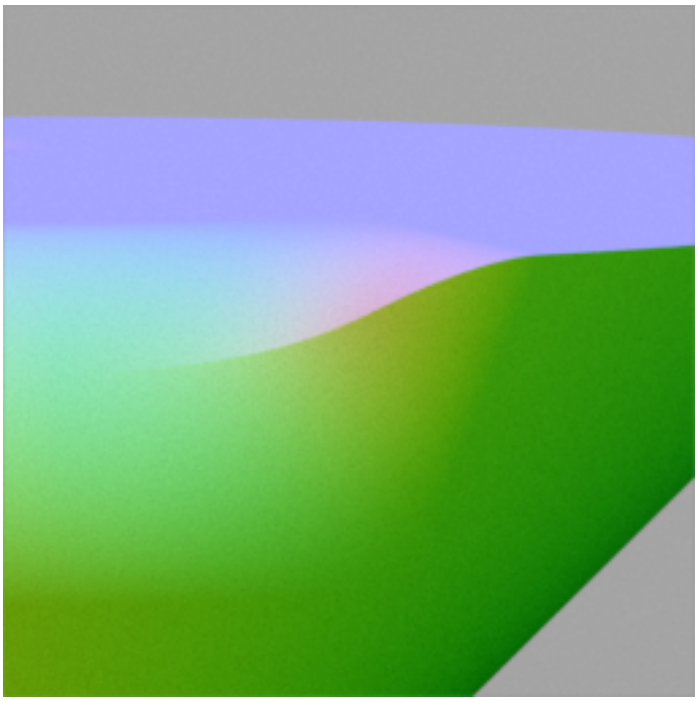


Figure 5.4: The straight edge shown during the interpolation (Image from [Grasberger et al., 2010]).

5.3 Mathematical Problems

All *BlobTree* primitives, due to their formulation as Skeletal Implicit primitives, are defined by a distance function to a given skeleton. For the point skeleton, the distance function to create the resulting sphere is uniform for the whole skeleton:

$$d(p_{(x,y,z)}) = \|o_{(x,y,z)} - p_{(x,y,z)}\|$$

In contrast, other primitives have a piece-wise distance function, such as the cube skeleton (other skeleton distance functions are stated in Appendix B), centered at $o_{(x,y,z)}$, which is defined ([de Groot, 2008]) by:

$$
\begin{aligned}
d(p_{(x,y,z)}) &= \sqrt{\uparrow(0, d_x - \|a_x\|)^2 + \uparrow(0, d_y - \|a_y\|)^2 + \uparrow(0, d_z - \|a_z\|)^2} \\
d_x &= \frac{a_x(o-p)}{\|a_x\|} \\
d_y &= \frac{a_y(o-p)}{\|a_y\|} \\
d_z &= \frac{a_z(o-p)}{\|a_z\|}
\end{aligned}
$$

The $\uparrow$ denotes the *max* operator. d_x, d_y and d_z are the distances between $p_{(x,y,z)}$ and the projection of o onto the three perpendicular arms a_x, a_y and a_z. They are normalized to the length of a_x, a_y and a_z. Assuming that a_x, a_y and a_z are actually axis aligned and a rotated cube is created using the affine transformation property of the primitive, this formulation can be simplified and split into distinct branches:

$$
d(p_{(x,y,z)}) = \begin{cases}
d_x & if\ d_x \geq 0 \wedge d_y < 0 \wedge d_z < 0 \\
d_y & if\ d_x < 0 \wedge d_y \geq 0 \wedge d_z < 0 \\
d_z & if\ d_x < 0 \wedge d_y < 0 \wedge d_z \geq 0 \\
\sqrt{d_x^2 + d_y^2} & if\ d_x \geq 0 \wedge d_y \geq 0 \wedge d_z < 0 \\
\sqrt{d_x^2 + d_z^2} & if\ d_x \geq 0 \wedge d_y < 0 \wedge d_z \geq 0 \\
\sqrt{d_y^2 + d_z^2} & if\ d_x < 0 \wedge d_y \geq 0 \wedge d_z \geq 0 \\
\sqrt{d_x^2 + d_y^2 + d_z^2} & if\ d_x \geq 0 \wedge d_y \geq 0 \wedge d_z \geq 0
\end{cases} \tag{5.1}
$$

$$
d_x = |p_x| - |a_x| \tag{5.2}
$$

$$
d_y = |p_y| - |a_y| \tag{5.3}
$$

$$
d_z = |p_z| - |a_z| \tag{5.4}
$$

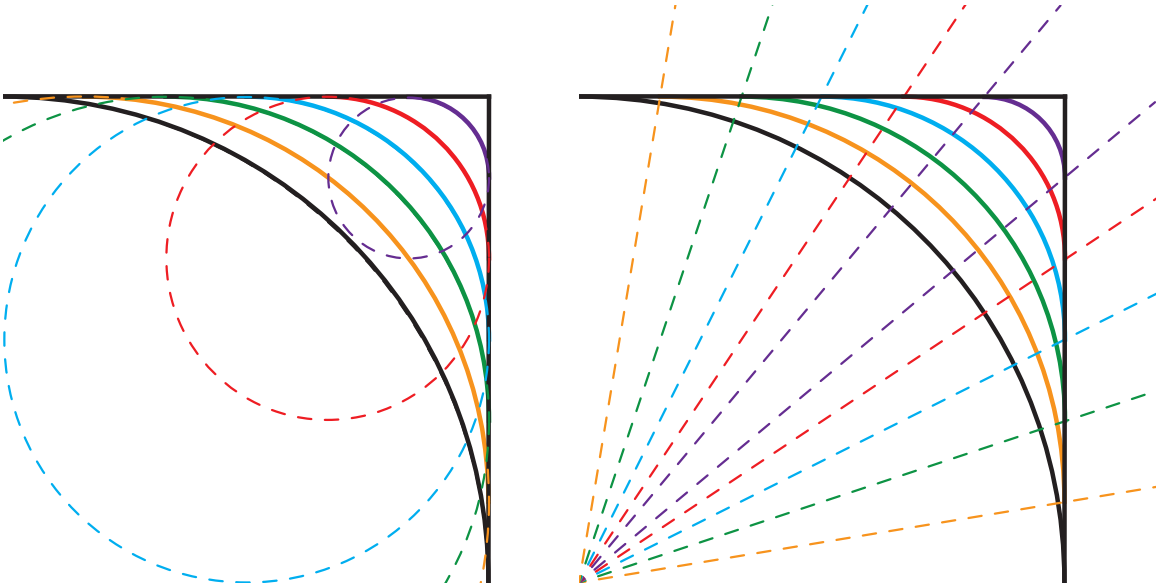
In these cases, a single coordinate is used for the distance calculation at the flat plane of a cube side and two coordinates are used at the edge (the edge is along the missing axis in the calculation, i.e. the field along z is calculated by $\sqrt{d_x^2 + d_y^2}$). The distance in a corner case is calculated with all three coordinates. The other skeletal primitives, potentially supporting filleting of edges, can have their distance functions separated into similar branches, such as the cylinder and the cone.

In the case of filleting two or more surfaces, the branch of the overall skeleton distance function defining the edge between the two (or three) planes adjacent to

the edge (or corner) has to be replaced. The sections below demonstrate, how this replacement can be done in the case of a single radius along a single edge (see Section 5.4), filleting along a *Surface Fillet Curve* (Section 5.5) and variable radius along one edge (Section 5.6).

5.4 Fixed Radius Filleting along one Edge

Previous CSG related work defines a fillet and its shape based on the radius of a rolling ball [Whited and Rossignac, 2009], which is used to create a canal surface. An implicit approach compatible with the *BlobTree* needs an implicit field of at least C^2 continuity. In addition, to create a modelling experience that gives the users familiar tools to specify fillets, *Angle-Based Filleting* also needs to to define a fillet using a rolling ball radius.



(a) Fillet defined using ball radius

(b) Fillet defined using opening angle

Figure 5.5: Comparison of a fillet defined using a rolling ball and the opening angles as defined in [Gourmel et al., 2013]. Radius, opening angle and fillet shape are colour coordinated.

[Gourmel et al., 2013] present a methodology to create a smooth blend between two fields of varying shape. This methodology defines a blend within the implicit space $\mathbb{I}^2$ (see [Barthe et al., 2003] and Section 2.4 for an in-depth discussion), by specifying

a so called opening angle within the first quadrant of $\mathbb{I}^2$ to define the smooth portion of the blend. The larger the opening angle θ, the smaller the smooth portion. The part of $\mathbb{I}^2$ that is within θ and $\frac{\pi}{2} - \theta$ defines the the smooth blend and approaches a circular curve.

As a result, it is possible to formulate how to transform the radius of the rolling ball into the opening angle. The mathematical formula defined by [Gourmel et al., 2013] can be used to calculate a new (close to circular) distance field, which is based on the opening angle θ as defined in [Barthe et al., 2004]:

$$Y_{p_0} = C_p tan(\theta) \tag{5.5}$$

where C_p defines the iso-distance and Y_{P_0} the y value of the intersection point between the two lines $x = C_p$, $y = tan(\theta)$. In this implicit space $\mathbb{I}^2$, the radius r_{fil} is then defined as

$$r_{fil} = C_p - Y_{P_0} \tag{5.6}$$

$$r_{fil} = C_p - C_p tan(\theta) \tag{5.7}$$

Given that θ needs to be expressed using the given filleting radius, the formula results in:

$$\theta = atan((r_{fil} - C_p)/C_p) \tag{5.8}$$

For any distance-based skeleton, the euclidean distance $d = \sqrt{d_1^2 + d_2^2}$ (see Equation 5.1) along a single edge can be replaced by $g(f_1, f_2)$ [Gourmel et al., 2013]:

$$g(f_1, f_2) \quad = \quad \begin{cases} max(f_1, f_2) & if\ f_1 \leq k_\theta(f_2)\ or\ f_2 \leq k_\theta(f_1) \\ \\ \{C : \bar{h}_C(f_1, f_2) = 1\} & otherwise \end{cases} \tag{5.9}$$

$\bar{h}_C$ is defined as:

$$\bar{h}_C(f_1, f_2) = \frac{\sqrt{(f_1 - k_\theta(C))^2 + (f_2 - k_\theta(C))^2}}{\bar{s}(\varphi)(C - k_\theta(C))} \tag{5.10}$$

$$where \tag{5.11}$$

$$k_\theta(f) = \frac{tan(\theta)}{2}\left(\frac{4}{1 + tan(\theta)}\lambda_\theta(f)\right)^2 \tag{5.12}$$

$$\lambda_\theta(f) = \begin{cases} f & if\ f \leq \frac{tan(\theta)}{2} \\ \frac{1-tan(\theta)}{4}\Phi(2\frac{2f-tan(\theta)}{1-tan(\theta)}) + \frac{tan(\theta)}{2} & otherwise \end{cases} \tag{5.13}$$

$$v(x) = exp(exp(exp(1) - exp(1)x) - 1) - 1 \tag{5.14}$$

In this case, the shape function $\bar{s}(\varphi)$ can be set to the identity, since only a "circular" fillet shape is desired (similar what has been done by [Vaillant et al., 2013]).

In these formulas Φ ensures C^∞ continuity of λ_θ. It can't be calculated directly, but has to be computed using binary search based on Φ^{-1} (see [Gourmel et al., 2013] for more details):

$$\Phi^{-1}(x) = x + \frac{1}{exp(1)}log\left(log\left(\frac{1}{v(x)} + 1\right) + 1\right) \tag{5.15}$$

Even though the original formula is used to apply a blend to two field-values f_1 and f_2, it can also be used to combine two distance values. There is a difference in how these formulas are used in the original paper compared to the usage within the filleting context. In the original paper, C corresponds to the iso-value, whereas in *Angle-Based Filleting*, the **iso-distance** is used instead. In order to accelerate the calculation of $g(f_1, f_2)$, the authors in [Gourmel et al., 2013] propose precomputing the results at regular intervals for the range of opening values $\theta \in [0, \frac{\pi}{4}]$. Any triplet of (f_1, f_2, θ) is then used to look up and interpolate the corresponding results using the method described in the original work.

Only the cylinder and the cone primitive have the capability to create fillets between two surfaces along one edge. Their unmodified distance calculations are stated in Appendix B. Both of these primitives have circular edges, due to their symmetric definition around the main axis. The fields in both cases are two dimensional fields that are rotated, and an edge in these fields is defined as a 2D corner, combining two distances, such as described above.

5.4.1 Cylinder Circular Edge

Figure 5.6 shows a quarter of the field of a cylinder's cross section. The skeleton itself is highlighted in faded green and the parts of the field that are only defined by the distance to the side or the top are faded too. This highlights the upper left corner section of the field that can be controlled through filleting.

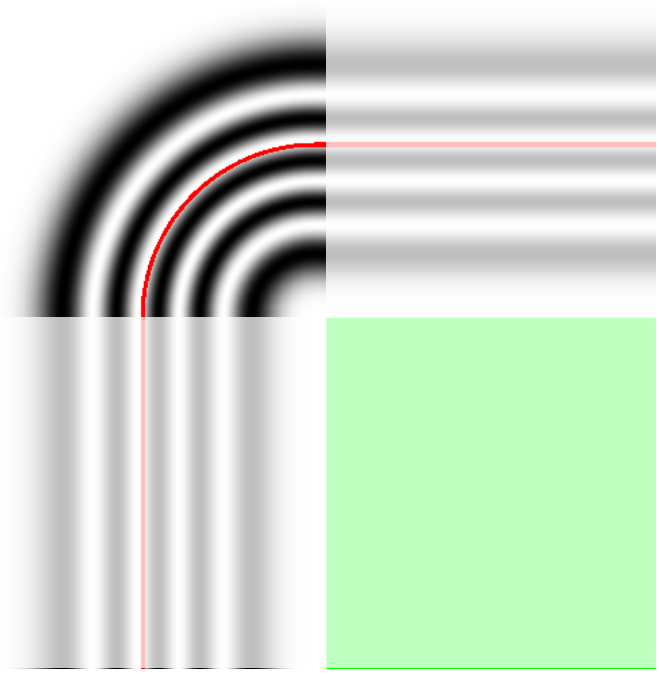


Figure 5.6: Quarter of cylinder's field cross-section. Everything apart from the edge case is faded, to highlight the area of interest. Red illustrates the cylinder surface.

For the 90° corner, the distances to the skeleton in the y axis and the xz plane, defined $d_{xz} = \sqrt{x^2 + z^2}$, can be used as the input for the above mentioned formulas, either the clean-union variant for a straight edge, or the blend variant with the desired opening angle/fillet radius as a parameter. This replaces the $\sqrt{y^2 + d_{xz}^2}$ branch of the skeletal distance formula, creating the following cylinder skeleton distance formula:

$$d(p_{(x,y,z)}) = \begin{cases} d_{xz} - r & if\ |y| < \frac{h}{2} \\ |y| - \frac{h}{2} & if\ |y| \geq \frac{h}{2}\ and\ d_{xz} < r \\ g(d_{xz} - r, |y| - \frac{h}{2}) & otherwise \end{cases} \tag{5.16}$$

5.4.2 Cone Circular Edge

Given the geometry of a cone skeleton, the edge regions and the tip don't always have 90° field arrangements, but the angle between the two fields depends on the two parameters of the cone skeleton:

- height h_{cone} and
- radius r_{cone}.

Figure 5.7 illustrates the field region of interest for the cone, in the same way as Figure 5.6 did for the cylinder. In this case, though to show the arrangement at the tip, the whole cross section is displayed, despite the symmetric field along the height/main axis.

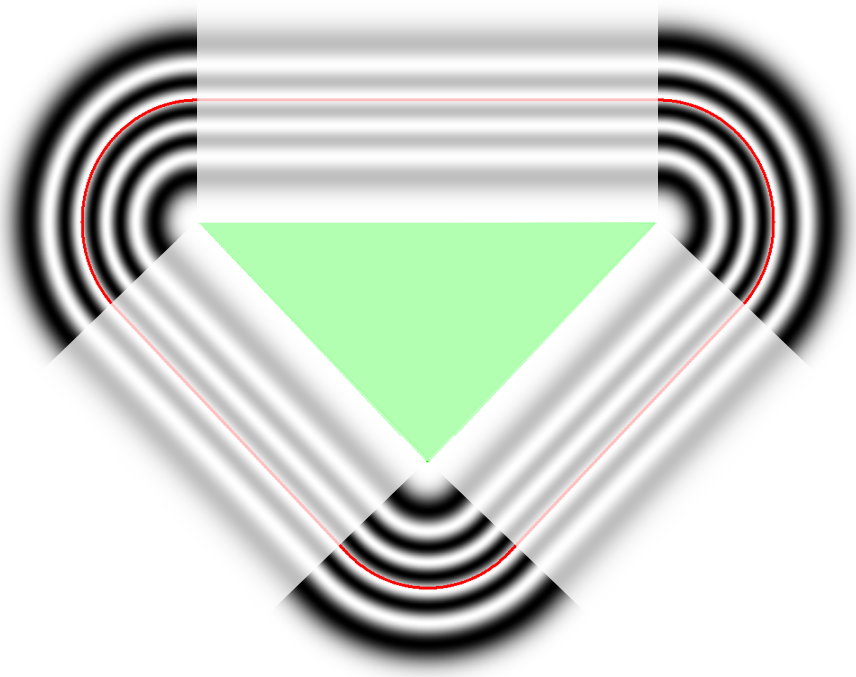


Figure 5.7: Cone field cross-section. Everything apart from the edge case is faded to highlight the area of interest. Red illustrates the surface.

Zooming in on the circular edge, the two mirrored corners in the two dimensional slice, Figure 5.8 shows that it is possible to define a local coordinate system going through the corner point of the skeleton. Assuming that the cone skeleton is defined so the skeleton's tip is the origin, the two axes, y and d_{xz} meet at their origin $o_c =$

(r_{cone}, h_{cone}). One axis of this coordinate system is y, the other one along the x value within the slice, which is d_{xz}.

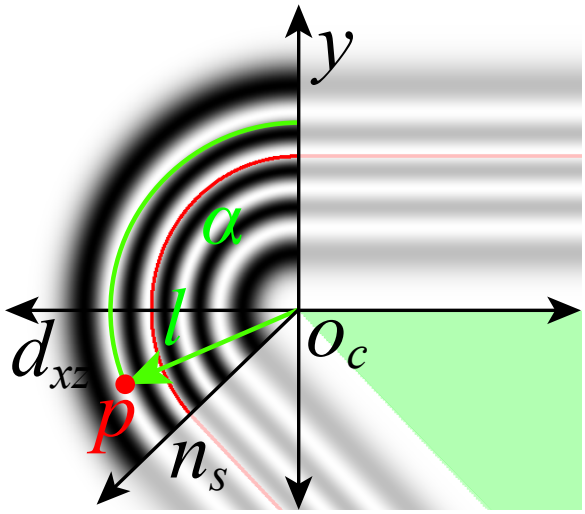


Figure 5.8: Transforming the euclidean coordinates for a point p into $\mathbb{I}^2$ at the cone circular edge, based on the polar coordinates $p = (l, \alpha)$.

The whole region of the corner is within y and n_s, where n_s is the normal of the cone skeleton side, calculated by

$$n_s = \frac{-h_{cone}}{r_{cone}} \tag{5.17}$$

Calculating the angle α_{n_s} between y and n_s provides the range of values that need to be mapped into the $90°\mathbb{I}^2$ region using polar coordinates for $p = (l, \alpha)$. (Note that y in this case is to be treated as the x coordinate in standard *atan* calculations, as the coordinate system is rotated by 90°).

$$\hat{n_s} = n_s - o_c \tag{5.18}$$

$$\alpha_{n_s} = atan(\frac{\hat{n_{sx}}}{\hat{n_{sy}}}) \tag{5.19}$$

Similarly the angle between y and p is:

$$\alpha = atan(\frac{p_{d_{xz}}}{p_y}) \tag{5.20}$$

$$l = \sqrt{p_{d_{xz}}^2 + p_y^2} \tag{5.21}$$

To map α into the orthogonal $\mathbb{I}^2$ space, it needs to be scaled:

$$\alpha_{\frac{\pi}{2}} = \alpha \frac{\pi}{2\alpha_{ns}} \tag{5.22}$$

resulting in adjusted polar coordinates $p_{\frac{\pi}{2}}$ for p.

$$p_{\frac{\pi}{2}} = (l, \alpha_{\frac{\pi}{2}}) \tag{5.23}$$

The polar representation of $p_{\frac{\pi}{2}}$ needs to be transformed back into euclidean coordinates:

$$p_{\frac{\pi}{2}} = (l \cos \alpha_{\frac{\pi}{2}}, l \sin \alpha_{\frac{\pi}{2}} \tag{5.24}$$

This results in the two input values $(p_{\frac{\pi}{2}x}, p_{\frac{\pi}{2}y})$ for $g(f_1, f_2)$ (Equation 5.9).

5.4.3 Cone Tip

Calculating the combined distance for the cone tip is very similar to the case of the circular edge. Figure 5.9 illustrates the local coordinate system(s) involved in transforming a point p within the cone tip region into $\mathbb{I}^2$. Note that there are two vectors orthogonal to the cone skeleton sides: $n_s = \frac{-h_{cone}}{r_{cone}}$ and $\grave{n}_s = \frac{h_{cone}}{r_{cone}}$.

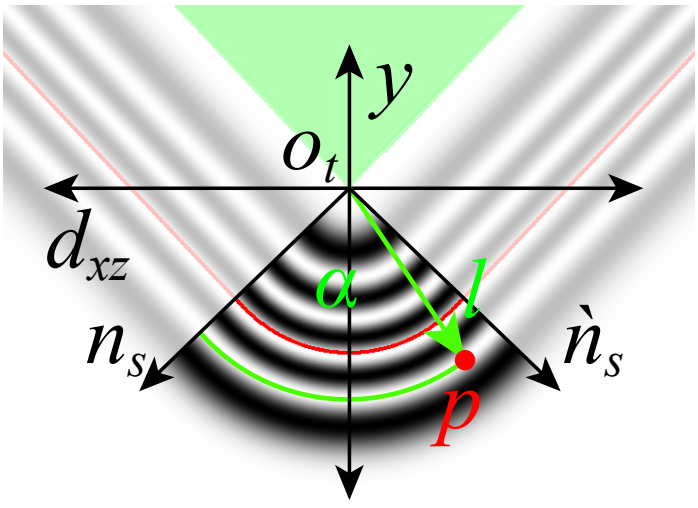


Figure 5.9: Transforming the euclidean coordinates for a point p into $\mathbb{I}^2$ at a cone tip, based on the polar coordinates $p = (l, \alpha)$.

In this case, $o_t = (0, 0)$. Thus, p is already defined in the local coordinate space once it is transformed into the two-dimensional space within the field's cross section.

While it would be possible to calculate the coordinates with the method of the circular corner, by using polar coordinates and projecting the angle α into the coordinate system spanned by n_s and $\grave{n}_s$, a computationally easier method is available. Assuming n_s and $\grave{n}_s$ are of length 1, all that is needed for $g(f_1, f_2)$ are the coordinates of p with regards to the two axes n_s and $\grave{n}_s$. These coordinates can be obtained by projecting p onto n_s and $\grave{n}_s$ respectively, creating p_{n_s}:

$$p_{n_s} \quad = \quad (p \cdot n_s, p \cdot \grave{n}_s)$$

In this formula, $a \cdot b$ denotes the vector dot product of two vectors a and b.

5.5 Creating a *Surface Fillet Curve*

A *Surface Fillet Curve* describes the creation of a warped surface, similar to what can be created with the WarpCurves approach. The main difference to WarpCurves is that a *Surface Fillet Curve* produces a fillet able edge along the curve, with a continuous field surrounding it. A sharp edge is produced, when the opening angle of the corresponding fillet is set to $\theta = 45°$. In the cases of smaller opening angles, a fillet analog to the fillets described above is produced. This means that for fillets smaller than the straight edge case, the control points of the WarpCurve only attract the surface, without the surface actually touching the control points. As a result the *Surface Fillet Curve* cannot create the same surface as the WarpCurve.

Another difference to WarpCurves is that the *Surface Fillet Curve*'s surface is not created by warping the underlying object, but instead a second *BlobTree* object is created based on the deformation which is blended with the underlying object to create the final surface. The object can be thought of as a sweep surface, where a "tear" shape is swept along the drawn line. This tear shape has the tip at the deformation curve, and the barycenter close to the original, undeformed, control curve. Section 5.5.1 describes how the field is calculated within the swept frame along the curve and Section 5.5.2 describes how this swept frame is calculated with C^2 continuity for any point in space surrounding the curve.

5.5.1 Calculating a *Surface Fillet Curve* Frame

A *Surface Fillet Curve* frame is built by transforming a $2D$ field (orthogonal to the drawn curve) into $\mathbb{I}^2$ (for an illustration of $\mathbb{I}^2$ see Figure 2.9). This transformation is similar to the cone primitive fillets described above; however, in this case it can be simplified, given that the area of the field to be filleted is always of the same size. The $2D$ coordinate system of a *Surface Fillet Curve* frame is built based on the displacement vector d between the original curve and the deformed curve, with the original curve being the origin within a frame and d the y coordinate. The x coordinate within this frame is orthogonal to the y coordinate, in $3D$ defined by the vector s (and the opposite vector s'), where $s \perp d$ and $s \perp t$ (t describes the tangent vector to the curve in $3D$). Both coordinate axes are sized relative to the deformation vector:

- y has the length of the deformation vector d; however, in $\mathbb{I}^2$ it is actually mapped to the vector along the 45° medial axis. Thus, its length corresponds to $\sqrt{2}$ in $\mathbb{I}^2$.
- $|x| = |d|$,

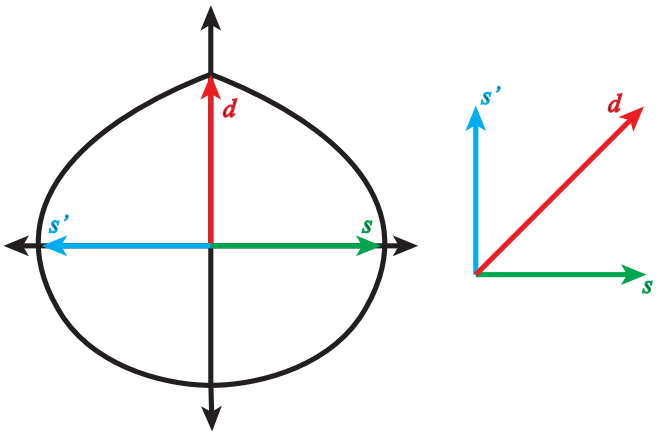


Figure 5.10: The coordinate system within a *Surface Fillet Curve* frame (left) and $\mathbb{I}^2$ (right).

Figure 5.10 illustrates how the vectors within a frame correspond *Surface Fillet Curve* to the coordinate system in $\mathbb{I}^2$. For any point p_{2D} within the coordinate frame, where $p_{2D_y} < 0$, the distance to calculate the field value is based on the euclidean distance to the displacement origin.

In order to calculate the distance within the coordinate frame, point p has to be transformed into p_{2D} (assuming $\hat{a}$ specifies the normalized vector of a):

$$p_{2D_x} = (p \cdot \hat{x})/|x| \tag{5.25}$$

$$p_{2D_y} = p_{y_{proj}} \tag{5.26}$$

$$where \tag{5.27}$$

$$p_{y_{proj}} = (p \cdot \hat{y})/|y| * \sqrt{2} \tag{5.28}$$

The resulting point p_{2D} is used to calculate the local distance value, needed for the field-value calculation. In this $2D$ coordinate system, the half-space, where $y \geq 0$, is used to create the fillet, whereas the opposite half-space ($y < 0$) creates a closed object. Since $\mathbb{I}^2$ and $g(p)$, as defined in Equation 5.9, only provide a definition for the first coordinate quadrant, a 90° opening, p_{2D} needs to be transformed into $p_{2D_{\frac{\pi}{2}}}$:

$$l = \sqrt{{p_{2D_x}}^2 + {p_{2D_y}}^2} \tag{5.29}$$

$$\alpha_{\frac{\pi}{2}} = atan(\frac{p_{2D_y}}{p_{2D_x}}/0.5) \tag{5.30}$$

$$p_{2D_{\frac{\pi}{2}}} = (l\ \cos\alpha_{\frac{\pi}{2}}, l\ \sin\alpha_{\frac{\pi}{2}}) \tag{5.31}$$

Assuming $p_{2D_{\frac{\pi}{2}}}$ as p_l, the resulting distance function is then:

$$d_{res} = \begin{cases} \sqrt{{p_{l_x}}^2 + {p_{l_y}}^2} & if\ p_{l_x} < 0\ \wedge\ p_{l_y} < 0 \\ g(p_l) & otherwise \end{cases} \tag{5.32}$$

Figure 5.11 shows the resulting field within the cross section of the *Surface Fillet Curve* primitive. Red shows the location of the surface (an epsilon is used to colour the pixels, so the line becomes thicker where the field is less dense), and it can be seen that the field on the bottom shows the compressed euclidean distance field without the tip. While this image shows the field in the case that a sharp edge is displayed, Figure 5.12 shows the field for the same frame through the same displacement curve, but with the opening angle set to fully open ($\theta = 0$). Again, since the lower half is compressed, the frame itself is asymmetric.

Figure 5.11: Field of a *Surface Fillet Curve* frame, shown with the straight edge tip.

Figure 5.12: Field of a *Surface Fillet Curve* frame, shown with the opening angle fully open.

5.5.2 Calculating the *Surface Fillet Curve* Object Field

In order to generate the primitive representing the *Surface Fillet Curve*, the single frame described in the previous section has to be swept along the displaced curve. Generating a C^2 continuous sweep of an implicit surface along a continuous curve creates several problems, mostly related to finding the closest point on the curve to a given point in $3D$ space. Luckily, alternative approaches exist, where the parameters to generate such a frame can be interpolated within a given $3D$ volume at the desired continuity. The thin-plate spline (or variational implicit surface) interpolation [Turk and O'Brien, 1999b] forms the basis for the *Surface Fillet Curve* object.

In order to create a coordinate frame for a specific point in space, the location of the displacement origin o, the displacement vector d and the normal to the displacement curve s are needed. The variational interpolation has to be set up, so that any point in space p surrounding the displaced curve returns the triplet (o, d, s) of the point on the displaced curve closest to p. Each vector contained in this triplet is defined in $3D$ and as a result the whole triplet interpolation is the result of nine piece-wise variational interpolations.

Additional control points for the variational interpolation similar to the off-curve constraints in [Sugihara et al., 2010] need to be specified. These additional control points need to have a minimum distance from each other so that the variational interpolation does not return incorrect values. Thus the minimum distance constraint according to [Sugihara et al., 2010] for new variational interpolation control points p_{new} applies to already added points $p_{control}$ (where $v_{control}$ specifies the corresponding value to be interpolated at $p_{control}$):

$$(|p_{new} - p_{control}|)/|v_{control}| > \frac{2}{3} \tag{5.33}$$

The off-curve constants are placed in the same way as outlined in the WarpCurves approach [Sugihara et al., 2010]: For every point along the curve, four off-curve constants are defined, as can be seen in Figure 5.13. The image shows the original control points along the curve in green, the displaced control points in orange, and the off-curve constants in blue. The second from the left off-curve constant below the curve is discarded due to it being too close to the second one. Contrary to the WarpCurve approach, where the weight at the off curve constants is 0 (to bound the field) the *Surface Fillet Curve* assigns the four off curve constants the same weights as the points along the displacement curve to create a region of constant values around the

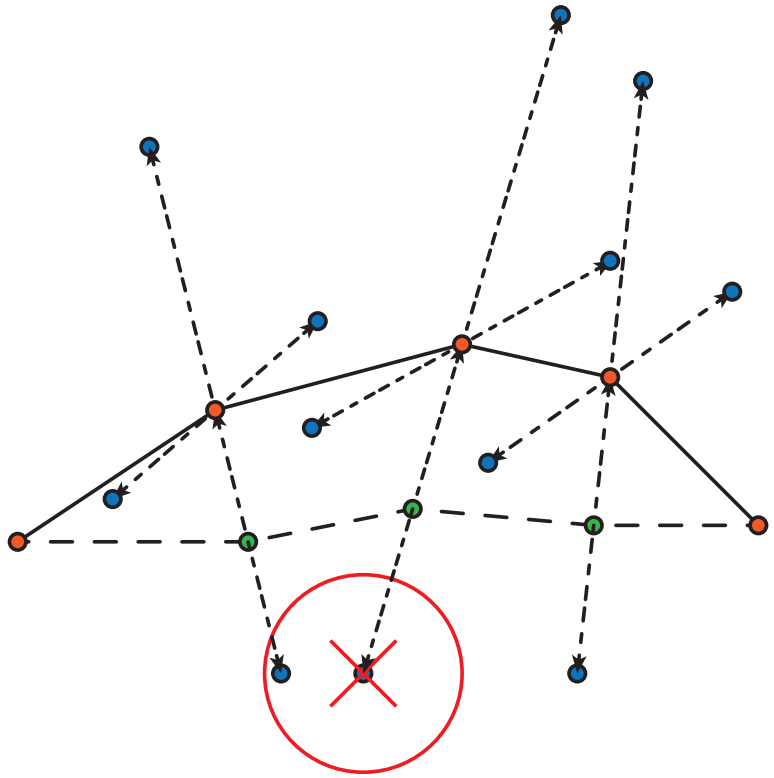


Figure 5.13: Placements of the off-curve constants.

curve.

The four off curve constants p_{new} to the control-points c_p, the displacement d and curve normal n to interpolate are:

$$p_{new} = \begin{cases} c_p \pm 2|d|\hat{d} \\ c_p \pm |d|\hat{n} \end{cases} \tag{5.34}$$

Generate the End Caps of the *Surface Fillet Curve* Primitive

In order to bound the field at the start and end of the *Surface Fillet Curve*, the variational interpolation has to be modified at the start and end of the curve. These regions are defined using planes built from the start/ end points and their respective plane normals (defined using the second and second last displaced control point locations). The constraint that the start and end point of the curve must not be displaced has to be enforced, in order to create a well formed implicit field. By having the zero displacement constraint, the *Surface Fillet Curve* starts and ends in a single point.

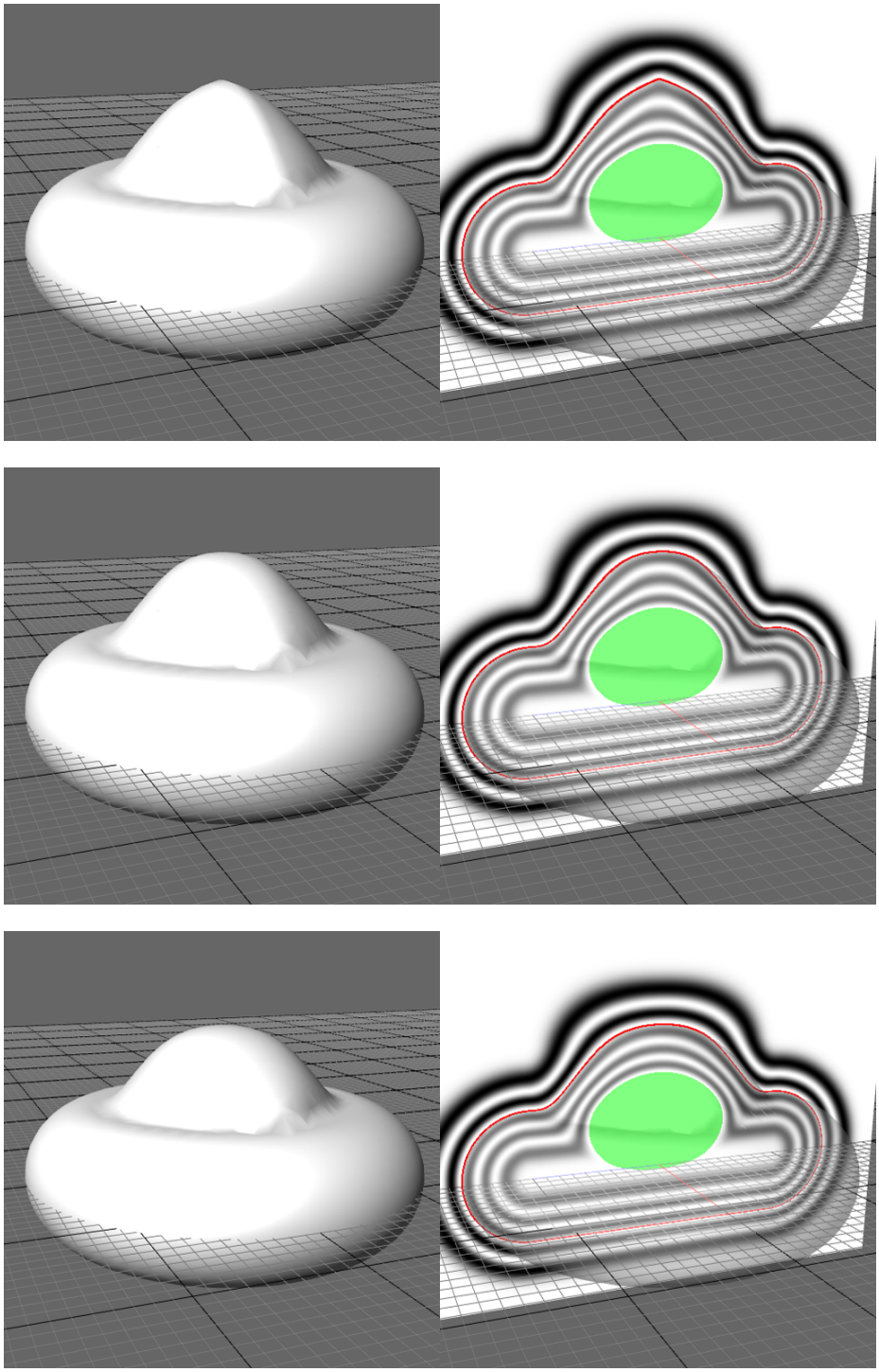


Figure 5.14: Comparison of different fillets along a *Surface Fillet Curve* and the corresponding slices through the field:
top row: opening angle of 45,
middle row: opening angle of 22.5 and
bottom row: opening angle of 0.

The field on the opposite sides of the start/end planes is calculated based on the point p projected onto the closest plane, p_{proj}. In addition, this field value is multiplied by the distance $\bar{p_{proj}P}$, modified by the Wyvill field-function (see Chapter 2).

5.5.3 Combining the *Surface Fillet Curve* Primitive with the Base Model

When the tear shaped *Surface Fillet Curve* primitive is blended with the underlying object it creates additional surface detail. Depending on the set opening angle when creating the *Surface Fillet Curve* frame, either a sharp edge or a round fillet is created. See Figure 5.14 for an example of different filletings along the *Surface Fillet Curve*.

5.6 Variable Radius Filleting along one Edge

Similar to the rolling ball solutions, such as [Whited and Rossignac, 2009], *Angle-Based Filleting* supports changing the radius of the "rolling ball" along the filleted edge. In *Angle-Based Filleting*, the variable fillet is done by changing the opening angle along the edge. The cylinder and the cone are the only primitives that have a single, circular edge. As a result, the filleting angle along the edge can be parametrized based on the angle within the xz plane. Theoretically, this parameterization would also be possible for the cone's tip, but it doesn't make sense to have a variable opening angle at only one point. As a result, the fillet at the cone tip can be considered constant, and only the circular edge requires a variable filleting radius.

In the canal surface context, [Xu et al., 2006] and [Peternell and Pottmann, 1997] discuss the maximum change $r(t)$ the radius can have. It depends on the formula defining the edge $m(t)$:

$$|m'(t)|^2 > r'(t)^2 \tag{5.35}$$

In this case, $m(t)$ describes the curve formed by the filleting origins around the main axis of a cylinder or cone. If the radius of the fillet is constant for the whole edge, $m(t)$ describes a circle surrounding the cylinder or cone base, parametrized by the rotation angle α around the center axis with radius r_{cone}:

$$m(\alpha) \quad = \quad (r_{cone} * cos(\alpha))^2 + (r_{cone} * sin(\alpha))^2 - r_{cone}^2 \tag{5.36}$$

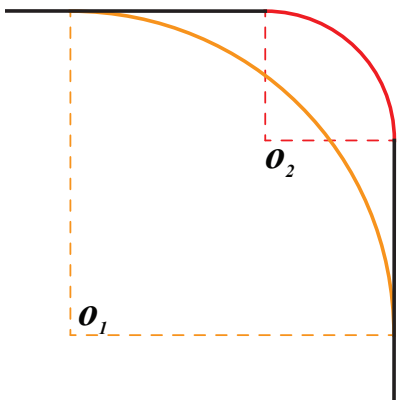


Figure 5.15: The different origins o_1 & o_2 of different filleting radii shown in $2D$.

However, as $m'(t)$ would evaluate to 0 for the constant angle case, using it to define the upper boundary of radius changes can't be done in this case.

When looking at this issue more closely, the curve defined as $m(t)$ in the canal surface case actually describes the so called spine of the canal surface. The ball center follows this curve when rolling so it keeps tangency with the two surfaces to be filleted. In this case, an opening angle of $\alpha = 0$ will place the curve center at the circular boundary of the skeleton, but, once $0 < \alpha < 45°$, the actual center of the sphere moves away from the skeleton. As a result, $m(t)$ ends up being more complex than the example in Equation 5.36. See Figure 5.15 for an outline of the different positions of the fillet origins o_1 and o_2 for different radii. For varying radii, $m(t)$ depends on the chosen radius control points, creating a different curve for any filleting radius combination. For this reason, an alternative way to parametrize a variable filleting radius is proposed below.

Given that the fillets are created using an implicit representation, it is important that the field is C^2 continuous. It can be guaranteed by choosing an appropriate interpolation method between the set opening angles. For example, interpolating between two opening angles along the circular edge around the given skeletons can be done using a combination of the first and the third basis function of cubic Hermite splines [Hearn and Baker, 2003]:

$$\begin{aligned} h_{00}(t) &= (1+2t)(1-t)^2 \\ h_{01}(t) &= t^2(3-2t) \end{aligned}$$

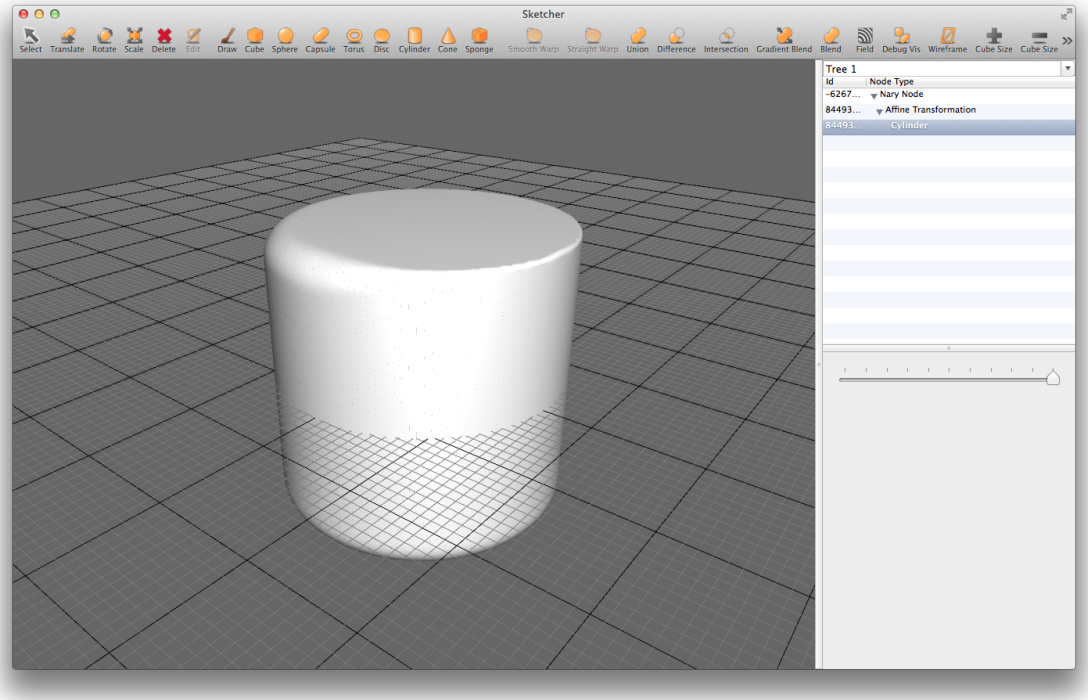


Figure 5.16: Cylinder showing a transition between a sharp edge and a filleted version with a larger opening angle/ radius.

Assuming that two opening angles α_1 and α_2 are defined along the circular edge (for example using the angle around y), h_{01} can be used to interpolate between α_1 and α_2, whereas h_{00} can be used for the reverse interpolation between α_2 and α_1. As long as the method is C^2 continuous, any other interpolation function could be used to calculate a variable number of set opening angles around the circle.

Figure 5.16 shows the results of different opening angles set along the cylinder edge circle. In the case of four or more angles set (the first and last angle are the same at $t = 0 = 1$), a Catmul-Rom spline interpolation [Hearn and Baker, 2003] is used to calculate the opening angle at any point around the circle. In Figure 5.17, the widget to change the opening angle is shown, with the angle at transformation start highlighted using the grey lines, and the new values in white.

5.7 Interactive User-Interface to Specify a Fillet

Filleting in itself is a slightly different than the three operations discussed in 4.5, because filleting changes a local property of an object, and the other transformations

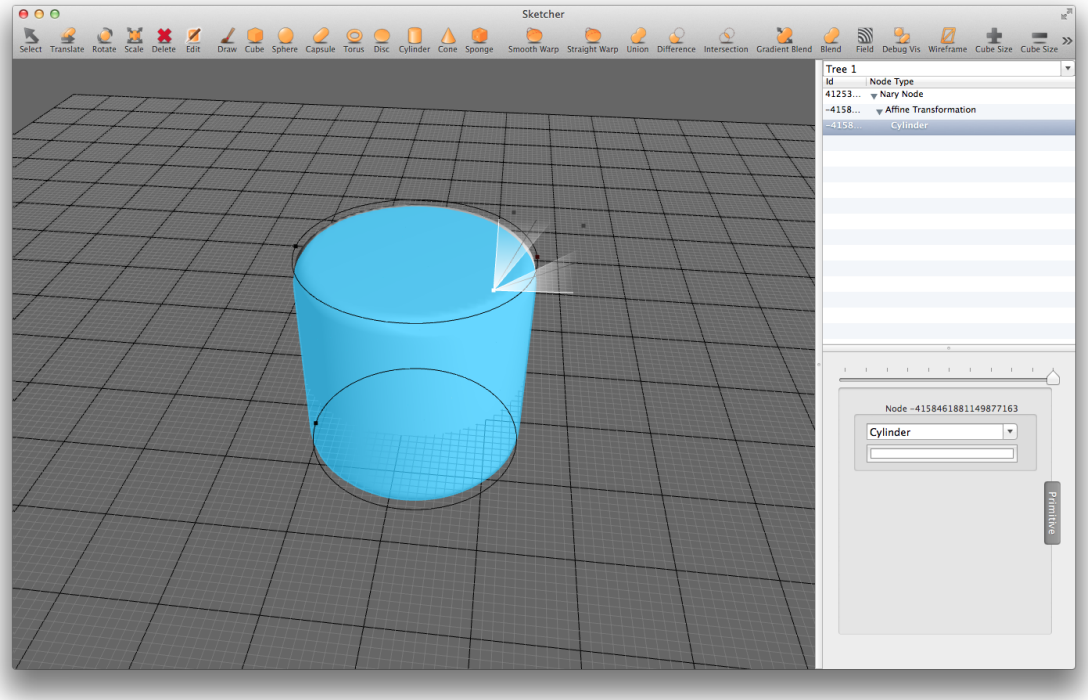


Figure 5.17: Cylinder showing the widget to control the opening angle along the edge.

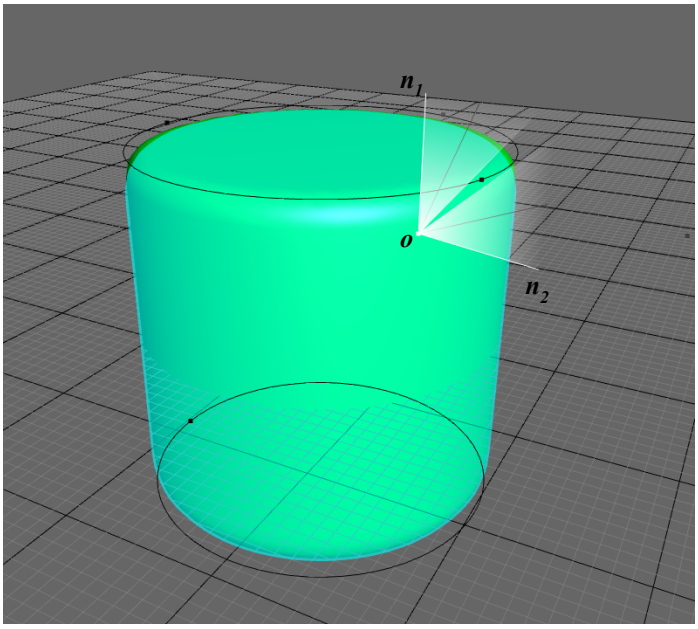


Figure 5.18: The filleting gizmo used, providing interactive feedback.

change a global object property. The main method of interaction, consisting of grabbing a gizmo handle and moving the mouse/finger into a specified direction, is the

same as the one used for the other gizmos. For this reason, the user interface to modify filleting is implemented in a similar fashion as the three standard transformation gizmos.

The filleting gizmo in itself is fairly simple, as it displays the local coordinate system along an edge based on the edge's curvature. It shows the origin point o (the location within the local coordinate frame along the edge, where the distance to the skeleton is 0) of the current position along the edge and the two surface normals n_1, n_2 going outwards from the origin point. In the example shown in Figure 5.18, the two surfaces filleted are orthogonal to each other, and the orientation of the gizmo is based on the angle of the gizmo's location around the cylinders main axis. The two lines in grey illustrate the current opening angle of the filleting, whereas the white section describes the new, modified opening angle. This is used to specify the new fillet at the gizmo location along the (circular) edge.

5.8 Example Models

The usage of the filleting techniques presented in this chapter is demonstrated on the example of two models:

- water tap/faucet,
- simplified model of car bonnet/hood.

Both of these models consist of shapes with variable fillet radii depending on their location, and often show smooth transitions between the primitives they are built from.

5.8.1 Tap Model

Figure 5.19 shows the water tap, with certain parts of the model highlighted to demonstrate different filleting (and blending) features, as discussed above. Highlight one shows the varying radius on the upper part of the tap handle, and proper blending of the tap handle and the vertical "head". In addition, the lower parts of the handle and the "head" demonstrate the straight edge.

Another straight edge between faces of the model is shown in highlight two, where the pipe inlet to the tap and the section connecting both pipes meet. On the back,

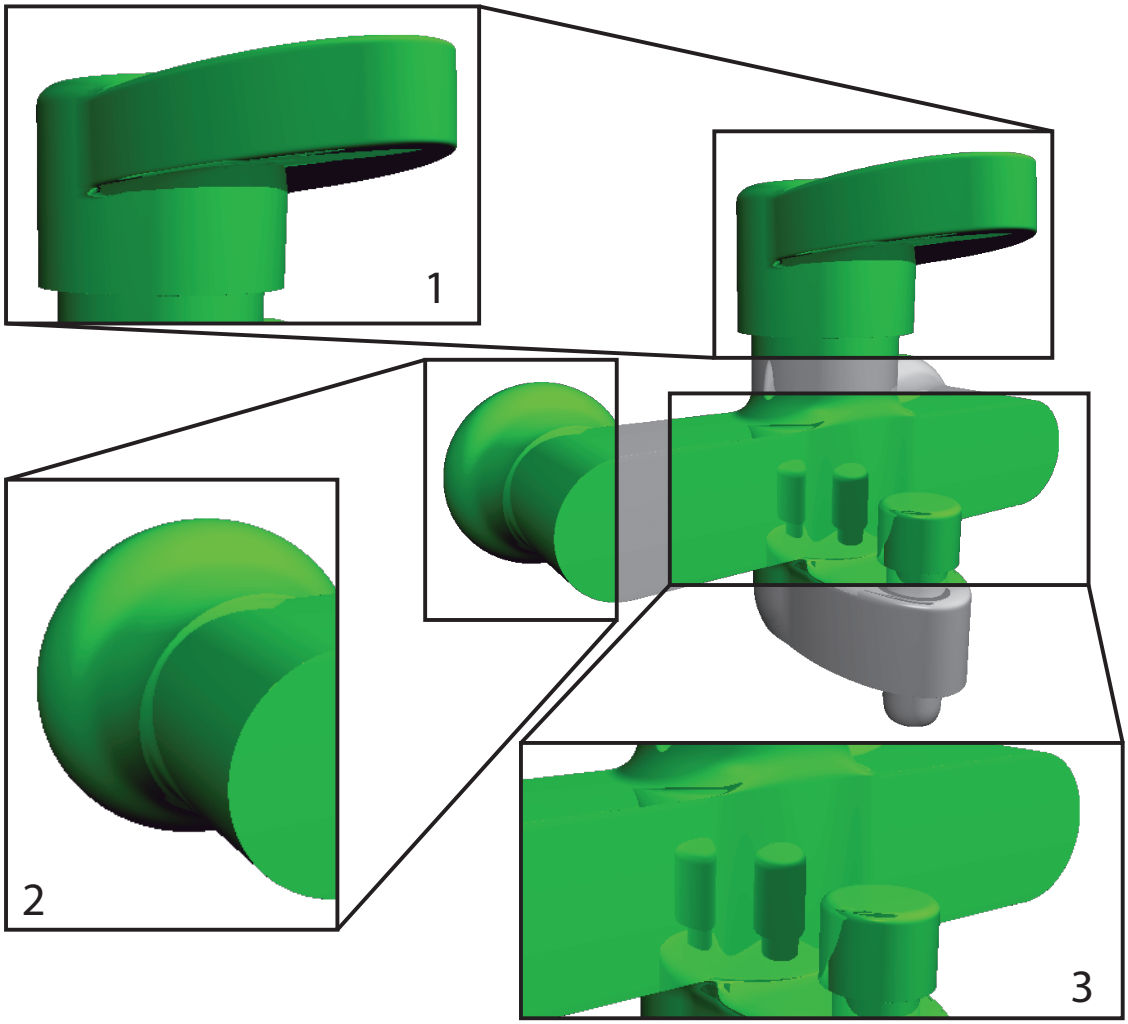


Figure 5.19: The water tap, with three sections of the image showing distinct fillets implemented.

the pipe section built from a cylinder (with the straight edge in the front) smoothly blends into the inlet base, modelled using an additional cone primitive.

In highlight three, the model shows how the straight edge of the stretched cube connecting the two inlet cylinder primitives smoothly blends into the shape of the central cylinder. There is a nice transition between the straight edge and a smooth corner, without having discontinuities in the underlying field because the function used to create the straight edge is C^2 continuous.

The model itself is built similar to the tap model found in [Grasberger et al., 2010], outlining comparable features. However, in the older approach, blends involving straight edges needed special care, since otherwise, a discontinuous field would be

created. On the contrary, *Angle-Based Filleting* creates cleaner looking models by having more control over the fillets, while being easier to build.

5.8.2 Car Bonnet Model

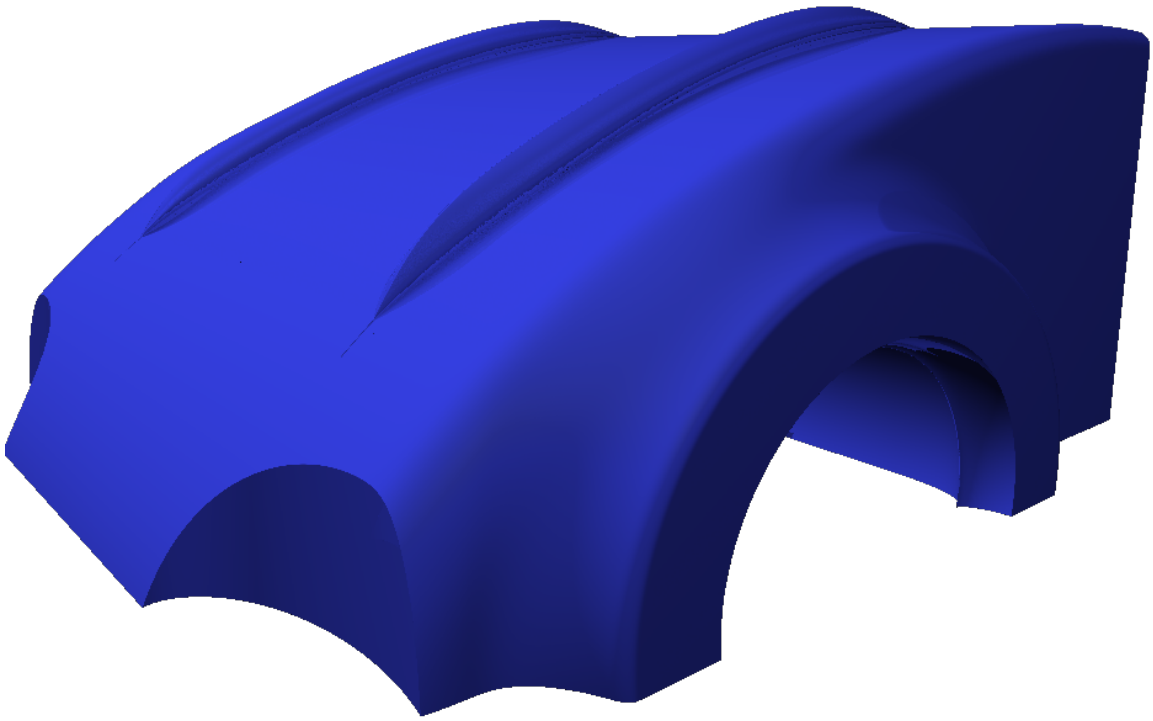


Figure 5.20: The bonnet, demonstrating the use of straight-edge warping.

The main features of the bonnet shown in Figure 5.20 are the two filleted curve accents on the bonnet itself, applied using the *Surface Fillet Curve* approach. These *Surface Fillet Curve* primitives are blended with the base shape of the bonnet using the Gradient Based Blend [Gourmel et al., 2013], so that the detailed features are not lost in the blending process, and the blend itself can be controlled.

In addition, the transition between the bonnet and the fenders is modelled using a variable fillet. Since the base shape of the bonnet with the adjoining fenders is a non-uniformly scaled cylinder, the implementation of the primitive enabled the variable radius fillet shown. The fender flares consist of additional cylinders, with

smaller filleting radii that are blended to the basic shape. Overall, this model shows that previous blend operators [Gourmel et al., 2013], *Angle-Based Filleting* and the *Surface Fillet Curve* approach can be used within one *BlobTree* and generate a C^2 continuous field.

5.9 Conclusion

In this Chapter, *Angle-Based Filleting* is proposed to extend the filleting capabilities of the *BlobTree*. The resulting field created by *Angle-Based Filleting* is of better continuity than fields created by previous approaches. With *Angle-Based Filleting*, even fillets resulting in sharp edges will not produce a discontinuous implicit field when used in a blending situation. Moreover, by allowing the user to specify the fillet based on a filleting radius (represented via an opening angle, analog to the Gradient Based Blend), the user can directly control the resulting shape. This extends the set of possible shapes that can be created using the *BlobTree* without adding additional shapes to the existing model to create certain fillets. Since the filleting operator has the same mathematical foundation as the Gradient Based Blend operator, the implicit fields of blends and fillets have similar properties. Having a more homogeneous field ensures that the visual distinction between blends and fillets is minimal, which means that user interfaces where the distinction between blends or fillets is made transparent to the user can be created.

The second contribution of this work, the *Surface Fillet Curve*, allows a user to draw unique contours onto a *BlobTree* model to create additional surface details. Since these surface details are based on the same mathematical foundation as the fillets, they will not destroy the continuous implicit field when blended with the model.

5.10 Future Work

The *Surface Fillet Curve* approach can be improved by supporting displacements that are directed towards the inside of the underlying primitive. Potentially, this could be realized by altering the blend function used to combine the *Surface Fillet Curve* primitive and the underlying object. Depending on the angle between the displacement vector and the base surface's normal, either normal blending or a variant of subtractive blending can be used to combine the base surface with the *Surface Fillet Curve*. In addition, better control on the slope of the *Surface Fillet Curve*

would help a designer to extend the potential shapes that can be created with this approach. Having methods to manually specify some of the off-curve constants could achieve this. The main challenge with the extended *Surface Fillet Curve* approach is to make sure the field continuity is preserved.

Angle-Based Filleting in its current form cannot properly fillet a corner where three edges meet and and create fillets in a controlled manner with preserved continuity. While this problem hasn't been solved yet, the following chapter explains how a potential solution to this problem may be found.

5.10.1 Filleting a Corner

While a corner in general can be defined by n edges originating at the same location, in the case of skeletal implicit surfaces, the only common primitive having a corner is the cube, where three orthogonal edges meet at every corner . Non-orthogonal edges at a cube primitive only appear once the cube primitive is sheared. This can be done by adding a shear transformation on top of the cube primitive in the *BlobTree*, leaving the primitive itself with orthogonal edges.

In many current modelling applications, there is no ground truth how the resulting surface of a fillet of three (or more) edges should look like. Often, it depends on the filleting order of the edges meeting at the corner. In other cases, the shape is dependent on the radius used, ie. if two edges have the same radius, the corner is defined as the transition between the two matching edges along the fillet radius of the third. In the case of the *BlobTree*, the main concern is to create a smooth continuous field. For this reason, the definition of the corner case is directly dependent on creating the continuous field. One way to implement this would be to have a common "corner radius/angle" and every edge interpolates between its opening angles and the corner angle. However, this results in a corner where all of the edges potentially need to interpolate the radius along their length. If the user wants to have three edges of fixed, but different radii, this might not be the desired solution. As a result, there should be a continuous interpolation between the three different opening angles to create the corner case.

A different approach could combine the fillet distances of the three neighbouring edges. The $[0, 1]$ range of an edge can be defined so it starts and ends at the region where the distance is influenced by another edge. As a result, the region of a corner is limited either by a parameter t along the edge of value 0, or value 1. In order

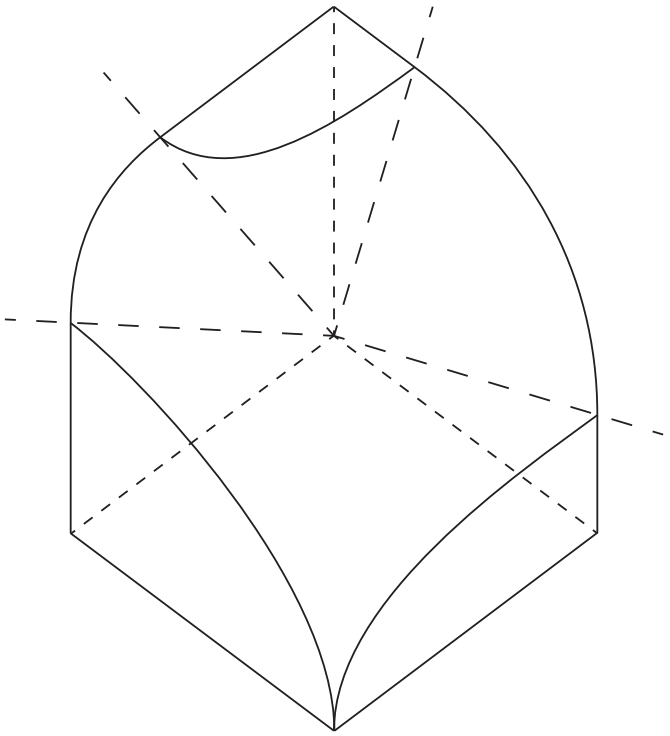


Figure 5.21: Illustration how a corner merging three different radii could be constructed.

to calculate the single distance of a point within the corner region, it is projected onto the start/end of each edge involved. Since the cube sides are axis aligned, only coordinate pairs of a point $p = (x, y, z)$ need to be taken into account. To calculate the points local to the edge fillet frames, the edge origin $o_e = (x_0, y_0, z_0)$ needs to be subtracted from p, resulting in $p_e = p - o_e$. The result needs to be projected onto the three edges: $p_x = (p_{e_y}, p_{e_z})$, $p_y = (p_{e_x}, p_{e_z})$, $p_z = (p_{e_y}, p_{e_z})$.

Based on these three points and their respective parameters t at the the corner boundaries, the base distances for each frame can be calculated. These three distances can be combined using the $max(d_x, d_y, d_z)$ operator, which, however, doesn't create a continuous field, and thus is not usable within the *BlobTree* context. Figure 5.21 shows a schematic of how the continuous corner shape can look like, including the three edge coordinate frames at the corner boundaries. The desired shape smoothly increases the opening angles along a circular arc, so that the closest opening angles of the three distinct edges are connected by an imaginary line. It line is also continuous with the opening angle interpolation along each edge, in order to produce a smooth field at any point within the corner.

Chapter 6

Improvements in *BlobTree* Sketch-based Modelling

Sketch-based *BlobTree* modelling is often used to rapidly prototype models. By drawing the outlines of the desired shape(s) and being able to combine the resulting objects using *BlobTree* operators, it is possible to create a complex model within a very short timeframe. Applications, such as Teddy [Igarashi et al., 2007], introduced sketch-based modelling to mesh approaches, and applications based on the *BlobTree* were introduced by [Schmidt and Wyvill, 2005a]. This implicit approach forms the basis for the research presented in this chapter.

6.1 Introduction

6.1.1 Motivation

Sketch-based modelling provides the user with tools to generate models based on a drawn shape's outline. This outline generates a $2D$ implicit field, which forms the basis of several transformations into $3D$:

- sweep
- surface of revolution and
- inflation.

The challenge in these approaches, as with any implicit modelling approach, is to create objects that have a C^2 continuous field. It is equally important that the

process to generate the $2D$ field is fast and that the resulting implicit field is well formed without any significant field compressions.

6.1.2 Implicit Sketch-based Modelling

Some implicit sketch-based modelling approaches, such as [Alexe et al., 2005] and [Alexe et al., 2007], use variants of a distance calculation followed by a medial-axis calculation to extract the skeleton of the drawn shape. In order to convert the skeleton into a $3D$ object with a continuous field, a convolution surface is created. [Schmidt and Wyvill, 2005a] describe the problem of using the distance-field directly to create $3D$ implicit objects. As a result, a variational implicit interpolation approach using sample points on and around the drawn shape to generate the $2D$ field is proposed.

6.1.3 Contributions

The work presented in this chapter is an improvement of the variational approach to sketching introduced in [Schmidt and Wyvill, 2005a]. It shows how the placement of the points the variational implicit field is based on can be optimized, so that fewer points are used to generate an implicit field of equal accuracy. The interpolated implicit field-values are improved and the number of variational interpolation coefficients is reduced.

6.1.4 Outline

Section 6.2 outlines how the state of the art approach (see Chapter 2) can be improved and Section 6.3 shows how to find non intersecting exterior control points. Section 6.4 describes how to reduce the number of interior control points and Section 6.5 defines which weights to choose for these interior control points to improve the resulting implicit field. Results are shown in Section 6.6 and the chapter is concluded in Section 6.7.

6.2 Problem Statement

The variational interpolation-based approach by [Turk and O'Brien, 1999b] uses a set of $2D$ points p_i combined with a weight v_i to generate the necessary matrices

to calculate a weight for an arbitrary input point. These weights v_i need to be set to the iso-value for each p_i, so that the variational interpolation returns the correct field-value for any point in the $2D$ plane defined by the sketched shape.

The number of terms of the variational interpolation function is directly related to the number of input points:

$$f(p) = \sum_{i \in N} a_i (\|p - p_i\|)^2 ln(\|p - p_i\|) + P(p) \tag{6.1}$$

$$where$$

$$P(p) = c_1 p_x + c_2 p_y + c_3 \tag{6.2}$$

For any input point added, an additional coefficient a_i needs to be calculated and used in the evaluation of the polynomial above. Reducing the number of input points also reduces the number of a_i's, resulting in faster field-value calculations. This enables direct calculation of field-values, while the previous approach by [Schmidt and Wyvill, 2005a] needs to precompute and interpolate them.

The original approach by [Schmidt and Wyvill, 2005a] uses several iso contours of the sketched shape to generate an implicit field. In this approach, the density of the implicit values corresponds to the distance values of the shape's skeleton. One iso contour is placed outside the sketched shape, whereas the other one is placed inside. However, finding these iso contours is computationally expensive and thus a distance transform based on sampling the field at a regular grid is used [Jain, 1989].

The iso contour outside the shape can be calculated using an offset curve, created by displacing every sample point along the polygon normal. Using the same approach for an interior curve, however, can cause some sampling issues inside the object. Figure 6.1 shows how interior sample points can overlap and be too close to others, which potentially can result in undesired interpolated values. An easy approach would be to discard sample points that are too close, but this still creates more sample points than needed. Additionally, when an interior offset curve is used for the variational interpolation, the resulting field might not reach a field-value of $f = 1$ due to the interpolation function. As a result of the field not using the full range ($[0, 1]$), the calculations in a blend or intersection situation might not produce the desired shape. For these reasons, this chapter describes an improved approach to finding interior control points for the variational interpolation in Section 6.4 and the corresponding weights in Section 6.5.

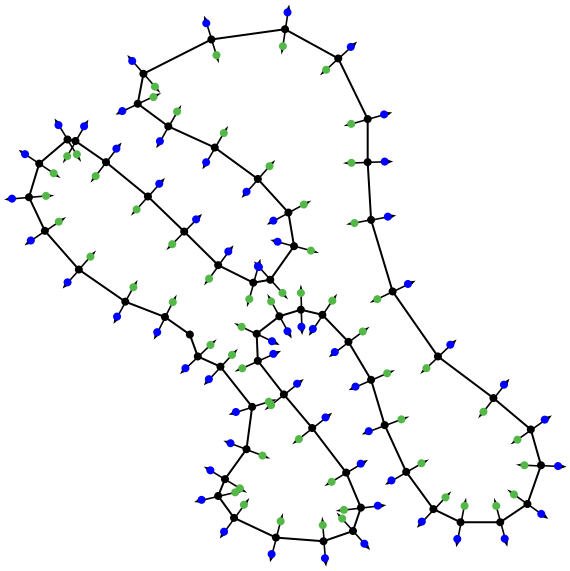


Figure 6.1: Exterior (blue) and interior (green) sample points, displaced by the control polygon normals of equal length.

6.3 Finding the Exterior Control Points

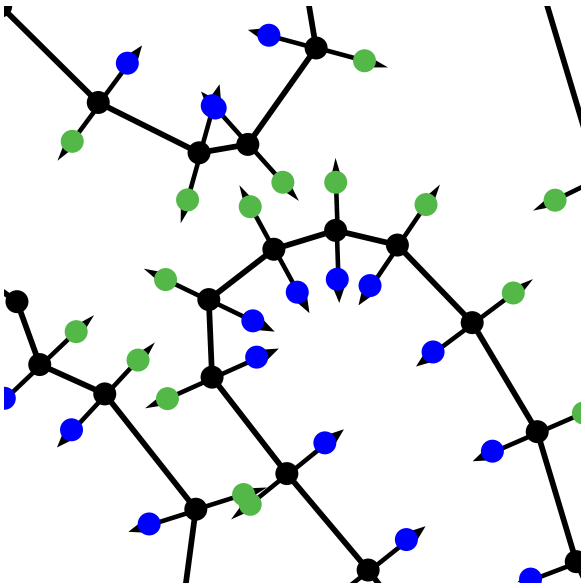


Figure 6.2: Exterior samples (blue) that are too close (top of polygon) and properly spaced samples (bottom of polygon).

In order to support base shapes of very high concave detail (e.g. sketching the

tentacles of an octopus, or branches of a tree), it is necessary to compute the maximum distance the sample points can be displaced outwards without producing self intersections. To calculate this maximum distance, the minimum distance d_{min} between all pairs of vertices of the original polygon can be computed. Assuming that neighbouring vertices cannot be part of two notches at once, the direct neighbourhood of each vertex can be ignored in the distance computation. Although this algorithm is $O(n^2)$, the number of points is relatively small and there is no loss in interactivity. The outside offset curve is now produced by displacing each control point along its normal (Figure 6.1):

$$p_{i_o} = p_i + n_i * \frac{d_{min}}{2} \tag{6.3}$$

The displacement is half of d_{min} to avoid a self intersecting polygon formed by the displaced points. Adding a minimum distance between the offset points avoids interpolation issues when control points are too close. Moreover, it reduces the number of points used in the interpolation algorithm. Figure 6.2 shows an example of exterior control points being too close.

One criterion for *BlobTree* primitives is that they are bounded, meaning the implicit field reaches 0 at a given distance (in most cases $d = 1$) from the skeleton. To satisfy this criterion, additional control points are placed on the border of the sketch area that have distances $d > 1$. These distances will then be bounded by the field function $g(d)$.

6.4 Finding the Interior Control Points

Depending on the shape of the polygon formed by the sampled input sketch, different strategies need to be used to find interior control points used for the variational interpolation. If the input polygon is convex, it is sufficient in most cases, to calculate the centroid point p_C of the polygon and use it as the only interior control point, where p_C is defined as:

$$p_C = \frac{1}{N} \sum_{i=1}^{N} p_i$$

It is not guaranteed that the centre point of a concave polygon lies within the polygon boundaries, so it is not a candidate for a sample point. Furthermore, if only an offset curve is used for sampling the interior of the polygon, it is not guaranteed that the resulting field contains field-values of $f = 1$. Consequently, concave polygons need an

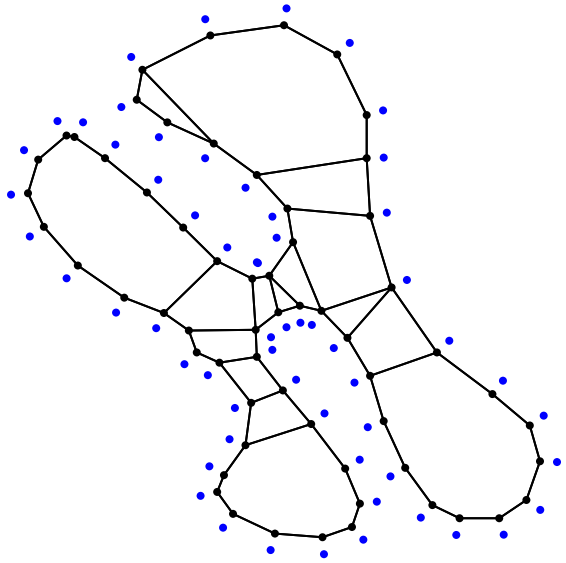


Figure 6.3: Control Polygon and Convex Decomposition.

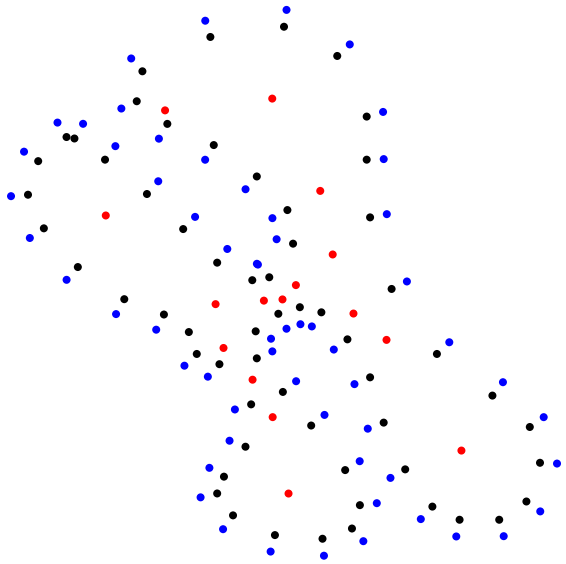


Figure 6.4: The final sample points used to build the thin-plate spline. The original points are black, and the outside displaced ones are blue and the inside sample points are red.

additional step to find the interior samples.

The medial axis [Blum, 1967] of a polygon describes a set of points, that can be interpreted as the skeleton of the control polygon. For standard Skeletal Implicit

Primitives, the distance on the skeleton is $d = 0$, resulting in a field-value $f(d) = 1$. It is not easy to directly find points along the medial axis of a concave polygon. The medial axis can be approximated by creating the polygon's *Approximate Convex Decomposition* as described by [Lien and Amato, 2006], which can be seen in Figure 6.3. For each of the convex sub-polygons of the Approximate Convex Decomposition, the

Figure 6.5: The implicit field created using a thin plate spline multiplied by a cosine function.

centroid can be used to approximate the medial axis of the shape.

6.5 Finding the Interior Control Point Weights

The standard Skeletal Implicit Primitives yield a field value $f(d) = 1$ on the skeleton and drop to zero at the limit of their influence range (commonly at a distance of $d = 1$ to the skeleton). This means that the straight-forward approach to assigning weights to the interior control point samples is to assign $v_i = 1$ for each of the convex decomposition polygon centroids.

Given that not every centroid will have the same distance to the polygon boundary, this will effectively produce an implicit field where the variation in field-value does not directly correspond to the variation in the distance to the control polygon. If a centroid is close to the polygon boundary (see the upper left convex polygon of the

upper right notch in Figure 6.3), the field will be compressed. On the other hand, if the centroid has a large distance to the polygon, the field will change less than the change in distances.

For this reason a better solution is to calculate the minimum distance of each centroid to the polygon boundary. The centroid with the largest minimum distance d_{max} corresponds to the "on skeleton" control point, whereas all the other centroids have their distances normalized by d_{max} to bring them into the $[0, 0.5]$ range (0.5 since the interior only describes half the implicit field):

$$d_{i_{corr}} = \frac{d_i}{d_{max}} * 0.5 \tag{6.4}$$

This ensures that there is at least one sample point produces the field-value $f = 1$. Figure 6.4 shows all sample points: on the control polygon in black, outside in blue and inside in red, and Figure 6.5 shows the resulting 2D field.

6.6 Results

Figure 6.6: Monkey model.

Every sketched shape benefits from the approach presented in this chapter since the implicit field can be generated using less variational sample points compared to the approach presented in [Schmidt and Wyvill, 2005a]. Because the interior control points are created using the convex decomposition, concave sketched polygons will gain a more correct interior field.

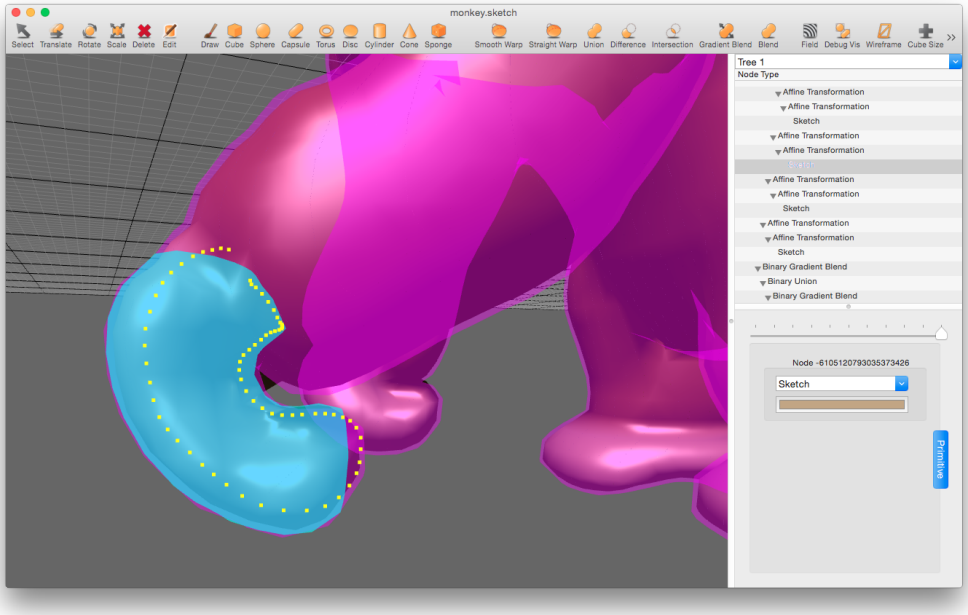
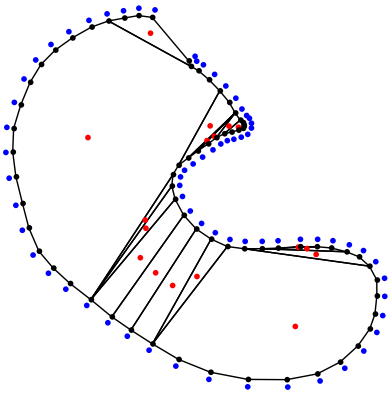


Figure 6.7: The convex decomposition of the hand, including the sample control points (black), interior control points (red) and exterior control points (blue).

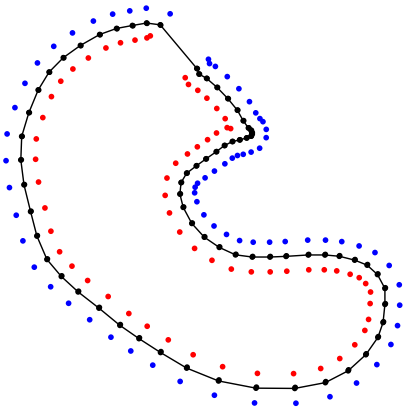
The monkey model in Chapter 4 (also see Figure 6.6), is partially built from concave shapes. For example, the main body and the hands as well as the feet of the monkey are concave shapes. Even though these shapes are drawn so that the single centroid would be sufficient as the only interior control point, the field of the sketched shapes still benefits from the convex decomposition and the additional interior control points.

Figure 6.8a shows the world space control points of the front hand in context of the whole model. These sample points form the control polygon shown in Figure 6.7. The convex decomposition of this polygon results in several sub-polygons and their centroids. Some of these centroids are fairly close to the control polygon itself, but due to the weighting algorithm described in Section 6.5, a proper implicit field is generated inside and outside the polygon.



(a) Centroid control points

(b) The field formed by the centroid control points.



(c) Offset control points

(d) The field formed by offsetting the control points.

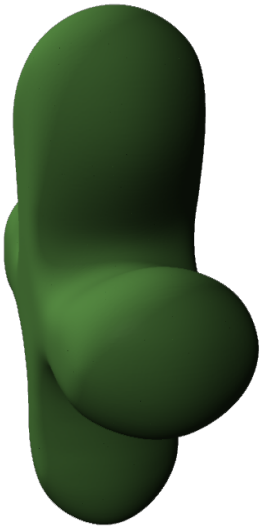
Figure 6.8: The sampled control points for the hand shape.

The field created by the control points is shown in Figure 6.8b. It can be seen that the centroid with the maximum distance to the polygon is mapped to the largest field value, and the others create field-values corresponding to the weighted distance values. In comparison, the field shown in Figure 6.8d is built using control points

(a) Centroid-based field

(b) The resulting inflated model.

(c) Offset-based field



(d) The resulting inflated model.

Figure 6.9: Comparison of the cross-like shape created using offset curves or this centroid based method. See the difference in the shape on the right.

calculated by using offset curves. The interior, while at first looking more uniform, has regions where the change in field-values does not correspond to the change in distance from them drawn shape. Compared to the presented sketching approach,

there is also a difference in the field outside the shape.

Another example of the difference between the centroid based approach and the offset curve based approach is demonstrated in Figure 6.9. A cross-like shape is drawn and used to create an inflated $3D$ object. The inflation is implemented according to [Schmidt et al., 2005a] and is based on the sketched field multiplied by a modified distance (using the Wyvill function) to the object's center. When Figure 6.9b is compared to Figure 6.9d, the interior region, having constant value of one, can be seen on the object's surface. Additionally, the difference in the field density results in a different shape.

6.7 Conclusion

The main contribution of this chapter is an alternative approach to calculating sample points for sketched shapes based on variational interpolation. Control points outside the drawn shape are calculated by displacing the shape along the normals. The length of the displacement is dependent on the shape itself to avoid a self intersecting polygon if the exterior control points are connected. In addition, reducing the number of interior control points is a goal of this approach as well. Convex polygons can be built using the polygon's centroid as the only control point.

Concave polygons, on the other hand, can't rely on the centroid being inside the polygon. For this reason, the proposed approach makes use of a convex decomposition algorithm to split the concave polygon into multiple convex polygons. The centroids of each convex sub-polygon are used as a variational interpolation sample that is weighted based on the maximum distance a centroid has to the polygon boundary.

This approach results in an implicit field, where the field-value variation corresponds to the variation in distance and doesn't create any regions where the field is compressed. Lastly, the number of variational interpolation sample points is reduced, resulting in a faster calculation of field-values.

One downside of the algorithm is the need to calculate d_{min} using an $O(n^2)$ algorithm in addition to the run time of the Approximate Convex Decomposition, which is stated as $O(nr)$, where n is the number of vertices and r the number of notches. Once the Approximate Convex Decomposition is created, the centroids have to be calculated and their distances to the sketch have to be determined. The larger the sketched shape, the longer it will take to create the resulting object, which, as a result, impacts the modelling experience negatively. Additionally, Figure 6.8a shows, that

d_{min} is very little, due to the neighbouring vertices being close. A more sophisticated approach, left for future work, could attempt to reduce the number of exterior control points, by choosing a larger value for d_{min}, and removing parts of the resulting polygon that self-intersect.

Chapter 7

Conclusion

Throughout this thesis several distinct improvements to *BlobTree* modelling are presented:

- Traversing the *BlobTree* can be *accelerated* when done linearly (Chapter 3).
- The *BlobTree* can be used for *collaborative* modelling on mobile devices (Chapter 4).
- The modelling capabilities are *extended* by *Angle-Based Filleting*, *Surface Fillet Curve* (Chapter 5) and improved sketching (Chapter 6).

7.1 Accelerated *BlobTree* Modelling

The accelerated *BlobTree* traversal method shows that a modified *BlobTree* traversal algorithm can improve visualization times by at least one order of magnitude. These visualization time improvements were achieved on GPUs and CPUs (on desktop and mobile variants). In fact, the original motivation of the accelerated traversal was to utilize a GPU for the field-value calculations needed in visualizing the *BlobTree*. Only later it turned out that other processing devices also benefited from the change in the traversal approach. For any hardware, the algorithm raises the *BlobTree* complexity that can still be visualized at interactive frame-rates. While this means for desktop hardware that very large *BlobTrees* can be edited while still having a responsive application, for mobile devices, this enables interactive modelling in the first place.

To sum up, changing a *BlobTree* traversal method to use the bottom-up approach presented solves two distinct issues. On desktop devices, it allows a *BlobTree* imple-

mentation to make use of the available GPU processing power and it enables modelling on lower power mobile CPUs as well.

While the approach presented to *BlobTree* traversal shows large improvements in the time a field-value calculation takes, one might argue that it is still not fast enough. Polygonization benefits most from this approach and the performance use case demonstrated closely follows a naive polygonization method. Ray-tracing, on the other hand, requires a lot more field-value calculations per image. The number of rays is at least 2073600 for a a HD-resolution image of 1920×1080 pixels (one ray per pixel). Unfortunately, the *BlobTree* has to be evaluated multiple times along a ray until the surface location can be determined. Assuming a very optimistic average of 10 field-value calculations per ray, and the synthetic test scene with 1024 leaf nodes accelerated using the kD-Tree, only the field value calculation part would take 2 seconds. This neither includes the time needed to transfer the data to the GPU, nor time spent shading and there are no secondary shadow rays casted in this case. Another problem with current GPUs is the code complexity possible in OpenCL. During implementing the algorithms as OpenCL kernels, one reason for doing some of the optimizations related to warp curves, was that the compiler could not create a running binary, if an algorithm was implemented in the most straight-forward fashion. This limitation is likely to disappear within the next couple of GPU generations, but it shows that the hardware is not yet as versatile as CPUs in some respect.

7.2 Collaborative *BlobTree* Modelling

CollabBlob shows that the small memory footprint of *BlobTree* models can be used to transmit changes in the model between several users. A protocol can be derived from a *BlobTree* definition that allows many users to work on the same model at the same time. Every user is allowed to modify the model and immediate feedback can be given to each participant because of the fast transmission of *BlobTree* model changes.

In the case that the participants try to apply conflicting *BlobTree* changes, the proposed action system includes an algorithm to determine and solve conflicts. Since every action is time-stamped and provides methods to undo and redo them, actions can be sorted and latecomer messages can be properly inserted.. Conflicting actions can be handled during this process as well.

Due to *CollabBlob*'s user interface the number of people who can realistically collaborate in a scene is limited. As already mentioned, every participant in the scene

is represented using a miniature of the current view point placed in the 3D space, or, if outside the current view frustum, along the window border. Additionally, each participant is represented by their own unique color. When the number of users in the current modelling session increases, the number of remote views will start to clutter the scene, up to the point where, especially on small screens, more screen real-estate will be occupied by these views, than the actual model. Moreover, the more users are participating, the harder it becomes to distinguish between the automatically assigned colours. Overall, *CollabBlob* lacks the capabilities to deal with a large number of users in the scene, and it would, additionally, benefit from something similar to the Branch-explore-merge [McGrath et al., 2012] approach.

7.3 Extended *BlobTree* Modelling

7.3.1 Filleting

Fillets can increase a user's control on the resulting shape. When fillets can be added to the model by altering a skeleton's distance function the complexity of a *BlobTree* stays the same, while altering the appearance of the resulting model. *Angle-Based Filleting*, the algorithm to introduce C^2 continuous fillets into Skeletal Implicit Primitives is based on a controllable C^2 blend operator. It is possible, that an application implementing *Angle-Based Filleting* has an interface, that does not have the distinction between fillet and blend. Instead the user can place smooth transitions between surfaces, and the underlying application decides whether a blend or a fillet has to be created. The added mathematical complexity of the *BlobTree* skeleton increases visualization time. On the other hand, if the same surface shapes were to be generated by adding and subtracting several *BlobTree* primitives, visualization times would be increased even more.

The *Surface Fillet Curve* approach presented provides additional tools to add surface details to an existing model. It is based on the same mathematical foundation as *Angle-Based Filleting*, but it creates additional *BlobTree* nodes. By drawing, displacing and filleting a curve, that is blended with the underlying *BlobTree*, novel shapes can be created in the *BlobTree* context.

The main disadvantage of the filleting approach presented is the lack of C^2 continuous handling of the corner case. While filleting along edges can be done in CSG using additional surfaces, blending between several fillets can still lead to problems,

especially when automatically generated. Once this is solved within the *BlobTree* framework, and the shape of the corner result can be controlled using a small set of parameters, there will be a large advantage over solutions with CSG. *Surface Fillet Curve*, on the other hand, lacks the capability to work with deformations directed into the original surface, which is possible and widely used in the WarpCurves approach. Additionally it could benefit from better control over the slope of the added shape, either by controlling the outline of the blend region, e.g. through a drawn outline or blend parameters, or by controlling the surface normals. Only then *Surface Fillet Curve* can provide a similar set of modelling operators than WarpCurves, which were the original inspiration.

7.3.2 Sketching

In general, sketch-based modelling within the *BlobTree* domain is based on the generation of an implicit field from a sketched shape using variational interpolation. This work shows how the variational interpolation can be improved by an alternative method to generate the control points used in the variational interpolation. For convex sketched polygons, the number of control points can be reduced significantly. Concave-sketched polygons can be split into convex sub-polygons, which, in the next step, can be used to generate the interior control points for the variational interpolation.

The implicit field generated by this sketch-based approach is generated without any significant undesired field compressions. This improves the usage of these implicit sketched shapes later on in additional modelling situations.

While this improvement to sketching reduces the number of control points needed to create the variational interpolation, only work has been done on optimizing the sketch interior. In fact, a similar approach using a convex decomposition and the centroids could be applied to the outside region as well, if it were interpreted as a rectangular polygon uses the sketched shape as a hole. The convex decomposition algorithm used for the interior can be used in this case as well. As a result, the $O(n^2)$ step of calculating the minimum distance between two sketch control points could be removed from the algorithm. In theory, the resulting field should show similar improvements over offset curves, as already demonstrated for the interior.

Bibliography

[Adzhiev et al., 1999] Adzhiev, V., Cartwright, R., Fausett, E., Ossipov, A., Pasko, A. A., and Savchenko, V. (1999). HyperFun project: a framework for collaborative multidimensional F-rep modeling. *Proceedings of Eurographics & ACM SIGGRAPH Workshop "Implicit Surfaces'99"*, pages 59–69.

[Akkouche and Galin, 2001] Akkouche, S. and Galin, E. (2001). Adaptive Implicit Surface Polygonization Using Marching Triangles. *Computer Graphics Forum*, 20(2):67–80.

[Alexe et al., 2005] Alexe, A., Barthe, L., Cani, M.-P., and Gaildrat, V. (2005). A Sketch-Based Modelling system using Convolution Surfaces. *Pacific Graphics*.

[Alexe et al., 2007] Alexe, A., Barthe, L., Cani, M.-P., and Gaildrat, V. (2007). Shape modeling by sketching using convolution surfaces. *ACM SIGGRAPH 2007 courses*, page 39.

[AMD, 2011] AMD (2011). *OpenCL Programming Guide*. Advanced Micro Devices, Inc., 1.3f edition.

[Bae et al., 2008] Bae, S.-H., Balakrishnan, R., and Singh, K. (2008). ILoveSketch: as-natural-as-possible sketching system for creating 3d curve models. In *UIST '08: Proceedings of the 21st annual ACM symposium on User interface software and technology*. ACM Request Permissions.

[Barbier and Galin, 2004] Barbier, A. and Galin, E. (2004). Fast Distance Computation Between a Point and Cylinders, Cones, Line-Swept Spheres and Cone-Spheres. *Journal of Graphics, GPU, and Game Tools*, 9(2):11–19.

[Barthe et al., 2002] Barthe, L., Dodgson, N. A., Sabin, M. A., Wyvill, B., and Gaildrat, V. (2002). Different Applications of Two-Dimensional Potential Fields for Volume Modeling. Technical Report 541, University of Cambridge.

[Barthe et al., 2003] Barthe, L., Dodgson, N. A., Sabin, M. A., Wyvill, B., and Gaildrat, V. (2003). Two-dimensional potential fields for advanced implicit modeling operators. *Computer Graphics Forum*, 22(1):23–33.

[Barthe et al., 2001] Barthe, L., Gaildrat, V., and Caubet, R. (2001). Extrusion of 1D Implicit Profiles: Theory and First Application. *International Journal of Shape Modeling*.

[Barthe et al., 2004] Barthe, L., Wyvill, B., and de Groot, E. (2004). Controllable binary csg operators for soft objects. *International Journal of Shape Modeling*.

[Benthin, 2006] Benthin, C. (2006). *Realtime ray tracing on current CPU architectures*. PhD thesis, Saarländische Universitäts- und Landesbibliothek, Postfach 151141, 66041 Saarbrücken.

[Bernhardt et al., 2010] Bernhardt, A., Barthe, L., Cani, M.-P., and Wyvill, B. (2010). Implicit Blending Revisited. *Computer Graphics Forum*, 29(2):367–375.

[Bloomenthal, 1988] Bloomenthal, J. (1988). Polygonization of Implicit Surfaces. *Computer Aided Geometric Design*, 5(4):341–355.

[Bloomenthal, 1994] Bloomenthal, J. (1994). An implicit surface polygonizer. In Heckbert, P. S., editor, *Graphics Gems IV*, pages 324–349. Academic Press Professional, Inc., San Diego, CA, USA.

[Bloomenthal, 1997] Bloomenthal, J. (1997). *Introduction to Implicit surfaces*. Morgan Kaufmann.

[Blum, 1967] Blum, H. (1967). A Transformation for Extracting New Descriptors of Shape. In Wathen-Dunn, W., editor, *Models for the perception of speech and visual form*, pages 362–380, Cambridge. Air Force Cambridge Research Laboratories (U.S.). Data Sciences Laboratory, MIT Press.

[Bryant, 1986] Bryant, R. E. (1986). Graph-Based Algorithms for Boolean Function Manipulation. *Computers, IEEE Transactions on*, C-35(8):677–691.

[Bryant, 1995] Bryant, R. E. (1995). Binary decision diagrams and beyond: enabling technologies for formal verification. In *Computer-Aided Design, 1995. ICCAD-95. Digest of Technical Papers., 1995 IEEE/ACM International Conference on*, pages 236–243.

[Bunnell, 2005] Bunnell, M. (2005). Dynamic Ambient Occlusion and Indirect Lighting. In Pharr, M. and Fernando, R., editors, *GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation*. Addison-Wesley Professional.

[Cani, 1993] Cani, M.-P. (1993). An implicit formulation for precise contact modeling between flexible solids. *SIGGRAPH '93: Proceedings of the 20th annual conference on Computer graphics and interactive techniques*.

[Chen et al., 2014] Chen, H.-T., Grossman, T., Wei, L.-Y., Schmidt, R. M., Hartmann, B. o. r., Fitzmaurice, G., and Agrawala, M. (2014). History Assisted View Authoring for 3D Models. In *Proceedings of the 32Nd Annual ACM Conference on Human Factors in Computing Systems*, pages 2027–2036, New York, NY, USA. ACM.

[Chen and Hoffmann, 1993] Chen, X. and Hoffmann, C. (1993). Trimming and closure of constrained surfaces. *cs.purdue.edu*.

[Chi et al., 2012] Chi, P.-Y., Ahn, S., Ren, A., Hartmann, B., Dontcheva, M., and Li, W. (2012). MixT: automatic generation of step-by-step mixed media tutorials. In Konstan, J. A., Chi, E. H., and Höök, K., editors, *CHI Extended Abstracts*, pages 1499–1504. ACM.

[Chu et al., 2006] Chu, C.-H., Chang, C.-J., and Cheng, H.-C. (2006). Empirical Studies on Inter-Organizational Collaborative Product Development. *Transactions of the ASME. Journal of Computing and Information Science in Engineering*, Volume 6(Issue 2):Pages 179–187.

[Chu et al., 2009] Chu, C.-H., Wu, P.-H., and Hsu, Y.-C. (2009). Multi-agent collaborative 3D design with geometric model at different levels of detail. *Robotics and Computer-Integrated Manufacturing*, 25(2):334–347.

[Darema et al., 1988] Darema, F., George, D. A., Norton, V. A., and Pfister, G. F. (1988). A single-program-multiple-data computational model for EPEX/FORTRAN. *Parallel Computing*, 7(1):11–24.

[de Groot, 2008] de Groot, E. (2008). *BlobTree Modelling*. PhD thesis, The University of Calgary, University of Calgary.

[de Groot and Wyvill, 2005] de Groot, E. and Wyvill, B. (2005). Rayskip: faster ray tracing of implicit surface animations. *GRAPHITE '05: Proceedings of the 3rd international conference on Computer graphics and interactive techniques in Australasia and South East Asia*.

[Denning et al., 2011] Denning, J. D., Kerr, W. B., and Pellacini, F. (2011). MeshFlow: Interactive Visualization of Mesh Construction Sequences. *ACM Trans. Graph.*, 30(4):66:1–66:8.

[Elber, 2005] Elber, G. (2005). Generalized filleting and blending operations toward functional and decorative applications. *Graphical Models*, 67(3):189–203.

[Elber and Cohen, 1997] Elber, G. and Cohen, E. (1997). Filleting and rounding using trimmed tensor product surfaces. *Proceedings of the fourth ACM symposium on Solid modeling and applications*, pages 206–216.

[Fatahalian and Houston, 2008] Fatahalian, K. and Houston, M. (2008). A Closer Look at GPUs. *Communications of the ACM*, 51(10).

[Fox et al., 2001] Fox, M., Galbraith, C., and Wyvill, B. (2001). Efficient Implementation of the BlobTree for Rendering Purposes. In *Shape Modeling and Applications, SMI 2001 International Conference on.*, pages 306–314. IEEE.

[Fuchs et al., 1980] Fuchs, H., Kedem, Z. M., and Naylor, B. F. (1980). On Visible Surface Generation by A Priori Tree Structures. *SIGGRAPH '80 Proceedings of the 7th annual conference on Computer graphics and interactive techniques*, 14:124–133.

[Gourmel et al., 2013] Gourmel, O., Barthe, L., Cani, M.-P., Wyvill, B., Bernhardt, A., and Grasberger, H. (2013). A Gradient-Based Implicit Blend. *Transactions on Graphics (SIGGRAPH 2013)*, 32(2).

[Gourmel et al., 2010] Gourmel, O., Pajot, A., Paulin, M., Barthe, L., and Poulin, P. (2010). Fitted BVH for Fast Raytracing of Metaballs. *Computer Graphics Forum*, 29(2):281–288.

[Grasberger, 2009] Grasberger, H. (2009). CSB: Combining traditional CSG with Blobs . Master's thesis, Vienna University of Technology, Vienna.

[Grasberger et al., 2010] Grasberger, H., Weidlich, A., Wilkie, A., and Wyvill, B. (2010). Precise Construction and Control of Implicit Fillets in the BlobTree. *Shape Modeling and Applications, International Conference on*, 0:151–162.

[Greenberg and Roseman, 1999] Greenberg, S. and Roseman, M. (1999). Groupware Toolkits for Synchronous Work. In Beaudouin-Lafon, M., editor, *Computer-Supported Cooperative Work (Trends in Software 7)*, pages 135–168. John Wiley & Sons Ltd.

[Hable and Rossignac, 2005] Hable, J. and Rossignac, J. (2005). Blister: GPU-based rendering of Boolean combinations of free-form triangulated shapes. In *ACM SIGGRAPH 2005 Papers*, pages 1024–1031, New York, NY, USA. ACM.

[Hable and Rossignac, 2007] Hable, J. and Rossignac, J. (2007). CST: Constructive Solid Trimming for Rendering BReps and CSG. *IEEE Transactions on Visualization and Computer Graphics*, 13(5):1004–1014.

[Han et al., 2003] Han, J. H., Kim, T., Cera, C., and Regli, W. (2003). Multiresolution modeling in collaborative design. *Computer and Information SciencesISCIS 2003*, pages 397–404.

[Hanniel and Haller, 2011] Hanniel, I. and Haller, K. (2011). Direct Rendering of Solid CAD Models on the GPU. In *2011 12th International Conference on Computer-Aided Design and Computer Graphics (CAD/Graphics)*, pages 25–32. IEEE.

[Hearn and Baker, 2003] Hearn, D. D. and Baker, M. P. (2003). *Computer Graphics with OpenGL*. Prentice Hall Professional Technical Reference, 3 edition.

[Hoffmann and Hopcroft, 1985] Hoffmann, C. and Hopcroft, J. (1985). The potential method for blending surfaces and corners. Technical report.

[Igarashi et al., 2007] Igarashi, T., Matsuoka, S., and Tanaka, H. (2007). Teddy: a sketching interface for 3D freeform design. In *ACM SIGGRAPH 2007 courses*, New York, NY, USA. ACM.

[Igarashi et al., 2005] Igarashi, T., Moscovich, T., and Hughes, J. F. (2005). As-rigid-as-possible shape manipulation. In *ACM SIGGRAPH 2005 Papers*, pages 1134–1141, New York, NY, USA. ACM.

[ISO, 1996] ISO (1996). Information technology – Open Systems Interconnection – Remote Procedure Call (RPC). *Internatioal Organization of Standardization*, ISO/IEC 11578.

[Jain, 1989] Jain, A. K. (1989). *Fundamentals of Digital Image Processing*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA.

[Jansen, 1991] Jansen, F. W. (1991). Depth-order point classification techniques for CSG display algorithms. *ACM Trans. Graph.*, 10(1):40–70.

[Kalra and Barr, 1989] Kalra, D. and Barr, A. (1989). Guaranteed ray intersections with implicit surfaces. *SIGGRAPH '89: Proceedings of the 16th annual conference on Computer graphics and interactive techniques*.

[Kim et al., 2006] Kim, T., Cera, C. D., Regli, W. C., Choo, H., and Han, J. (2006). Multi-Level modeling and access control for data sharing in collaborative design. *Adv. Eng. Inform.*, 20(1):47–57.

[Knoll et al., 2009] Knoll, A., Hijazi, Y., Kensler, A., Schott, M., Hansen, C., and Hagen, H. (2009). Fast Ray Tracing of Arbitrary Implicit Surfaces with Interval and Affine Arithmetic. *Computer Graphics Forum*, 28(1):26–40.

[Kung and Robinson, 1981] Kung, H. T. and Robinson, J. T. (1981). On Optimistic Methods for Concurrency Control. *ACM Transactions on Database Systems*, 6(2):213–226.

[Lamport, 1978] Lamport, L. (1978). Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21:558–565.

[Lien and Amato, 2006] Lien, J.-M. and Amato, N. M. (2006). Approximate convex decomposition of polygons. *Computational Geometry*, 35(1-2):100–123.

[Mäntylä, 1987] Mäntylä, M. (1987). *An introduction to solid modeling*. Computer Science Press, Inc., New York, NY, USA.

[Marion and Jomier, 2012] Marion, C. and Jomier, J. (2012). Real-time collaborative scientific WebGL visualization with WebSocket. In *Proceedings of the 17th International Conference on 3D Web Technology*, pages 47–50, New York, NY, USA. ACM.

[McCormack and Sherstyuk, 1998] McCormack, J. and Sherstyuk, A. (1998). Creating and Rendering Convolution Surfaces. *Computer Graphics Forum*, 17(2):113–120.

[McGrath et al., 2012] McGrath, W., Bowman, B., McCallum, D., Hincapié-Ramos, J. D., Elmqvist, N., and Irani, P. (2012). Branch-explore-merge: Facilitating Real-time Revision Control in Collaborative Visual Exploration. In *Proceedings of the 2012 ACM International Conference on Interactive Tabletops and Surfaces*, pages 235–244, New York, NY, USA. ACM.

[Middleditch and Sears, 1985] Middleditch, A. and Sears, K. (1985). Blend surfaces for set theoretic volume modelling systems. *SIGGRAPH '85: Proceedings of the 12th annual conference on Computer graphics and interactive techniques*.

[Mishra et al., 1997] Mishra, P., Varshney, A., and Kaufman, A. (1997). CollabCAD: A Toolkit for Integrated Synchronous and Asynchronous Sharing of CAD Applications. In *Proceedings TeamCAD: GVU/NIST Workshop on Collaborative Design, Atlanta, GA, USA*. State University of New York at Stony Brook.

[Mitchell, 1990] Mitchell, D. P. (1990). Robust ray intersection with interval arithmetic. In *Proceedings on Graphics interface'90*, pages 68–74, Halifax, Nova Scotia. AT&T Bell Laboratories Murray Hill, NJ 07974.

[Morris et al., 2004] Morris, M. R., Ryall, K., Shen, C., Forlines, C., and Vernier, F. (2004). Beyond "Social Protocols": Multi-user Coordination Policies for Co-located Groupware. In *Proceedings of the 2004 ACM Conference on Computer Supported Cooperative Work*, pages 262–265, New York, NY, USA. ACM.

[Mouton et al., 2011] Mouton, C., Sons, K., and Grimstead, I. (2011). Collaborative visualization: current systems and future trends. In *Proceedings of the 16th International Conference on 3D Web Technology*, pages 101–110, New York, NY, USA. ACM.

[Munshi, 2011] Munshi, A. (2011). The OpenCL Specification. Khronos.

[Nishino et al., 1999] Nishino, H., Utsumiya, K., Korida, K., Sakamoto, A., and Yoshida, K. (1999). A method for sharing interactive deformations in collaborative 3D modeling. In *Proceedings of the ACM symposium on Virtual reality software and technology*, pages 116–123, New York, NY, USA. ACM.

[O'Brien and Turk, 2002] O'Brien, J. F. and Turk, G. (2002). Modelling with Implicit Surfaces that Interpolate. *ACM Transactions on Graphics*, 21(4).

[Parisi, 2012] Parisi, T. (2012). *WebGL: Up and Running*. O'Reilly Media, Inc., 1st edition.

[Pasko et al., 1995] Pasko, A. A., Adzhiev, V., Sourin, A., and Savchenko, V. (1995). Function Representation in Geometric Modeling: Concepts, Implementation and Applications. *The Visual Computer*, 11(8):429–446.

[Pasko et al., 2005] Pasko, G. I., Pasko, A. A., and Kunli, T. L. (2005). Bounded blending for function-based shape modeling. *IEEE Computer Graphics and Applications*, 25(2):36–45.

[Peternell and Pottmann, 1997] Peternell, M. and Pottmann, H. (1997). Computing rational parametrizations of canal surfaces. *Journal of Symbolic Computation*, 23:255–266.

[Pharr and Mark, 2012] Pharr, M. and Mark, W. R. (2012). ispc: A SPMD Compiler for High-Performance CPU Programming. In *Innovative Parallel Computing Conference*, pages 1–13.

[Pinelle et al., 2003] Pinelle, D., Gutwin, C., and Greenberg, S. (2003). Task analysis for groupware usability evaluation: Modeling shared-workspace tasks with the mechanics of collaboration. *ACM Trans. Comput.-Hum. Interact.*, 10:281–311.

[Popov et al., 2007] Popov, S., Günther, J., Seidel, H.-P., and Slusallek, P. (2007). Stackless KD-Tree Traversal for High Performance GPU Ray Tracing. *Computer Graphics Forum*, 26(3):415–424.

[Ramani et al., 2003] Ramani, K., Agrawal, A., Babu, M., and Hoffmann, C. (2003). CADDAC: Multi-Client Collaborative Shape Design System with Server-based Geometry Kernel. *Journal of Computing and Information Science in Engineering*, 3(2):170–173.

[Reiner et al., 2011] Reiner, T., Mückl, G., and Dachsbacher, C. (2011). Interactive modeling of implicit surfaces using a direct visualization approach with signed distance functions. *Computers and Graphics*, 35(3):596–603.

[Ricci, 1973] Ricci, A. (1973). A constructive geometry for computer graphics. *The Computer Journal*, 16(2):157–160.

[Romeiro et al., 2006] Romeiro, F., Velho, L., and De Figueiredo, L. (2006). Hardware-assisted Rendering of CSG Models. In *2011 12th International Conference on Computer-Aided Design and Computer Graphics (CAD/Graphics)*, pages 139–146. IEEE.

[Rossignac, 1999] Rossignac, J. (1999). Blist: A Boolean List Formulation of CSG Trees. Technical Report GIT-GVU-99-04.

[Rossignac, 2012] Rossignac, J. (2012). Ordered Boolean List (OBL): Reducing the Footprint for Evaluating Boolean Expressions. *IEEE Transactions on Visualization and Computer Graphics*, 17(9):1337–1351.

[Rossignac and Requicha, 1984] Rossignac, J. and Requicha, A. (1984). Constant-Radius Blending in Solid Modeling. *ASME Computers In Mechanical Engineering (CIME)*, 3:65–73.

[Rubin and Whitted, 1980] Rubin, S. M. and Whitted, T. (1980). A 3-dimensional representation for fast rendering of complex scenes. In *Proceedings of the 7th annual conference on Computer graphics and interactive techniques*, pages 110–116, New York, NY, USA. ACM.

[Sabin, 1968] Sabin, M. (1968). The Use of Potential Surfaces for Numerical Geometry. *British Aircraft Corporation, Dynamics and Mathematical Services, Weybridge Division, UK. VTO/MS/153.*

[Schmidt, 2006] Schmidt, R. (2006). *Interactive Modeling with Implicit Surfaces*. PhD thesis, University of Calgary, University of Calgary.

[Schmidt and Wyvill, 2005a] Schmidt, R. and Wyvill, B. (2005a). Generalized sweep templates for implicit modeling. In *Proceedings of the 3rd international conference on Computer graphics and interactive techniques in Australasia and South East Asia*, pages 187–196, New York, NY, USA. ACM.

[Schmidt and Wyvill, 2005b] Schmidt, R. and Wyvill, B. (2005b). Implicit Sweep Surfaces. Technical Report Tech. Rep. 2005-778-09, University of Calgary.

[Schmidt et al., 2005a] Schmidt, R., Wyvill, B., Costa-Sousa, M., and Jorge, J. A. (2005a). ShapeShop: Sketch-Based Solid Modeling with the BlobTree. In *Proc. 2nd Eurographics Workshop on Sketch-based Interfaces and Modeling*, pages 53–62. Eurographics, Eurographics.

[Schmidt et al., 2005b] Schmidt, R., Wyvill, B., and Galin, E. (2005b). Interactive implicit modeling with hierarchical spatial caching. *SMI '05: Proceedings of the International Conference on Shape Modeling and Applications 2005*, pages 104–113.

[Sederberg and Parry, 1986] Sederberg, T. W. and Parry, S. R. (1986). Free-form deformation of solid geometric models. *SIGGRAPH Comput. Graph.*, 20(4):151–160.

[Shapiro, 1994] Shapiro, V. (1994). Real Functions for Representation of Rigid Solids. *Computer Aided Geometric Design*, 11(2).

[Shirazian et al., 2012] Shirazian, P., Wyvill, B., and Duprat, J.-L. (2012). Polygonization of implicit surfaces on Multi-Core Architectures with SIMD instructions. In Childs, H. and Kuhlen, T., editors, *Eurographics Symposium on Parallel Graphics and Visualization (EGPGV)*, pages 89–98.

[Shirley and Marschner, 2009] Shirley, P. and Marschner, S. (2009). *Fundamentals of Computer Graphics*. A. K. Peters, Ltd., Natick, MA, USA.

[Smits, 2005] Smits, B. (2005). Efficiency issues for ray tracing. In *ACM SIGGRAPH 2005 Courses*, New York, NY, USA. ACM.

[Snyder, 1992] Snyder, J. (1992). Interval Analysis for Computer Graphics. *SIGGRAPH '92: Proceedings of the 19th annual conference on Computer graphics and interactive techniques*, pages 121–130.

[Sugihara et al., 2008] Sugihara, M., de Groot, E., Wyvill, B., and Schmidt, R. (2008). A Sketch-Based Method to Control Deformation in a Skeletal Implicit Surface Modeler. In *Proceedings of the 5th Eurographics Workshop on Sketch-Based Interfaces and Modeling*, pages 65–72.

[Sugihara et al., 2010] Sugihara, M., Wyvill, B., and Schmidt, R. (2010). WarpCurves: A tool for explicit manipulation of implicit surfaces. *Computers and Graphics*, 34(3).

[Tobiasz et al., 2009] Tobiasz, M., Isenberg, P., and Carpendale, S. (2009). Lark: Coordinating Co-located Collaboration with Information Visualization. *IEEE Transactions on Visualization and Computer Graphics*, 15(6):1065–1072.

[Turk and O'Brien, 1999a] Turk, G. and O'Brien, J. F. (1999a). Shape transformation using variational implicit functions. In *Proceedings of the 26th annual conference on Computer graphics and interactive techniques*, pages 335–342, New York, NY, USA. ACM Press/Addison-Wesley Publishing Co.

[Turk and O'Brien, 1999b] Turk, G. and O'Brien, J. F. (1999b). Variational Implicit Surfaces. Technical report, Georgia Institute of Technology.

[Vaillant et al., 2013] Vaillant, R., Barthe, L. i. c., Guennebaud, G. e. l., Cani, M.-P., Rohmer, D., Wyvill, B., Gourmel, O., and Paulin, M. (2013). Implicit skinning: real-time skin deformation with contact modeling. *ACM Trans. Graph.*, 32(4):125:1–125:12.

[van Overveld and Wyvill, 2004] van Overveld, K. and Wyvill, B. (2004). Shrinkwrap: An efficient adaptive algorithm for triangulating an iso-surface. *The Visual Computer*, 20(6):362–379.

[W3C, 2013] W3C (2013). Websockets API specification. W3C.

[Wald, 2004] Wald, I. (2004). *Realtime Ray Tracing and Interactive Global Illumination*. PhD thesis, Computer Graphics Group, Saarland University.

[Wald and Havran, 2006] Wald, I. and Havran, V. (2006). On building fast kd-trees for ray tracing, and on doing that in O(N log N). In *Proceedings of the 2006 IEEE Symposium on Interactive Ray Tracing*, pages 61–69. SCI Institute, University of Utah, Salt Lake City, UT.

[Whited and Rossignac, 2009] Whited, B. and Rossignac, J. (2009). Relative blending. *Computer-Aided Design*, 41(6):456–462.

[Wong and Gutwin, 2014] Wong, N. and Gutwin, C. (2014). Support for Deictic Pointing in CVEs: Still Fragmented After All These Years'. In *Proceedings of the 17th ACM Conference on Computer Supported Cooperative Work & Social Computing*, pages 1377–1387, New York, NY, USA. ACM.

[Wyk, 1991] Wyk, C. J. V. (1991). *Data Structures and C Programs, 2nd Ed.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2nd edition.

[Wyvill et al., 1999] Wyvill, B., Guy, A., and Galin, E. (1999). Extending the CSG tree. Warping, blending and Boolean operations in an implicit surface modeling system. *Computer Graphics Forum*, 18(2):149–158.

[Wyvill et al., 1986] Wyvill, G., McPheeters, C., and Wyvill, B. (1986). Data structure for soft objects. *The Visual Computer*, 2(4):227–234.

[Xu et al., 2006] Xu, Z., Feng, R., and Sun, J. (2006). Analytic and algebraic properties of canal surfaces. *Journal of Computational and Applied Mathematics*, 195(1-2):220–228.

[Yagel et al., 1994] Yagel, R., Cohen, D., and Kaufman, A. (1994). Normal Estimation in 3D Discrete Space.

Appendix A

Performance Evaluation of Traversal Algorithm

While the performance graphs for the comparison between Top-Down tree traversals (with and without a Bounding-Volume-Hierarchy) and the Bottom-Up tree traversal(with and without a KD-Tree built with the Median-Split strategy) are shown in Chapter 3, the underlying values are given below. The tables for time are stated in μs, and show the average time per field-value evaluation (ie traversal and distance/blend calculations). Due to size constraints on the page, the columns are marked with several abbreviations describing the test case for which the numbers are stated. The meaning of these abbreviations is explained in Table A.1.

TD	Top-Down Traversal
BU	Bottom-Up Traversal
B	Balanced Tree
LH	Left-Heavy Tree
RH	Right-Heavy Tree
BVH	Bounding-Volume-Hierarchy Acceleration
KDM	KD-Tree Median Split Acceleration

Table A.1: Legend of the abbreviations used in the following performance and memory related tables

	TD B	TD LH	TD RH	TD B BVH	TD LH BVH	TD RH BVH
2	0.13	0.063	0.061	0.137	0.072	0.071
4	0.264	0.14	0.14	0.241	0.172	0.169
8	0.471	0.294	0.302	0.331	0.334	0.325
16	0.931	0.708	0.806	0.398	0.667	0.674
32	1.802	1.88	2.003	0.526	1.436	1.489
64	3.593	4.964	5.11	0.683	2.973	3.228
128	7.976	9.328	9.897	0.854	5.913	6.073
256	17.733	18.947	20.319	1.011	11.74	11.752
512	33.05	40.443	41.67	1.136	23.608	24.305
1024	94.537	77.634	82.688	1.281	48.858	47.823
2048	139.31	157.192	172.426	1.426	95.7	95.876

Table A.2: The performance numbers for all Top-Down traversal variants, applied to the 3 distinct artificial tree cases.

A.1 Average Traversal Time

While the graphs in chapter 3 are only printed until the number of leaf nodes reached 1024, the performance numbers here also include the case for 2048. The reason, why the last case wasn't plotted before is, that the last case is not entirely comparable to the rest. The iterative Top-Down approach needs to store the traversal state and

	BU B	BU LH	BU RH	BU KDM B	BU KDM LH	BU KDM RH
2	0.068	0.046	0.049	0.066	0.05	0.051
4	0.115	0.07	0.073	0.071	0.06	0.068
8	0.189	0.099	0.12	0.095	0.059	0.066
16	0.331	0.161	0.221	0.098	0.069	0.081
32	0.621	0.276	0.409	0.1	0.08	0.093
64	1.256	0.509	0.81	0.13	0.083	0.093
128	2.418	0.947	1.759	0.141	0.107	0.127
256	4.83	1.824	3.236	0.141	0.117	0.135
512	9.534	3.589	6.445	0.19	0.136	0.19
1024	19.651	7.093	12.739	0.285	0.175	0.21
2048	39.877	14.085	25.271	0.728	0.382	0.457

Table A.3: The performance numbers for all Bottom-Up traversal variants, applied to the 3 distinct artificial tree cases.

the temporary memory for the result stack. The problem is that for large *BlobTrees* the storage requirement is bigger than the maximum buffer size that can be allocated using OpenCL on the device.

As a result, the implementation of Top-Down traversal reverts to a multi-pass approach, which reuses the temporary memory. This requires additional kernel-runs to be enqueued on the device, potentially altering the performance behaviour. Table A.2 shows, that there is a clear trend for the average field-value evaluation time. However, since only kernel time on the GPU is measured for this case, the additional time spent enqueuing the kernel multiple times for each pass, can influence these numbers as well.

The Bottom-Up approach on the other hand does not require a multi-pass approach for 2048 leaf nodes. For 2048 nodes, ie. twice the tree size, Bottom-Up traversal takes twice as long as for 1024, highlighting the trend of execution times already shown in Chapter 3.

A.2 Traversal Memory Usage

In order to properly analyze the aforementioned performance numbers, their memory usage has to be taken into account, since it directly affects the performance. The more temporary memory is needed, the longer the tree-traversal takes. This is especially true for the Top-Down traversal approach. It can also be seen that it doesn't matter that much for the Top-Down traversal compared to the Bottom-Up case, because the Top-Down traversal accesses the memory in a less predictable pattern The Bottom-Up

	TD B	TD LH	TD RH	TD B BVH	TD LH BVH	TD RH BVH
2	3	2	2	3	2	2
4	4	4	4	4	4	4
8	5	8	8	5	8	8
16	6	16	16	6	16	16
32	7	32	32	7	32	32
64	8	64	64	8	64	64
128	9	128	128	9	128	128
256	10	256	256	10	256	256
512	11	512	512	11	512	512
1024	12	1024	1024	12	1024	1024
2048	13	2048	2048	13	2048	2048

Table A.4: The memory usage for the Top-Down traversal variants, applied to the 3 distinct artificial tree cases.

can be reduced to two distinct memory access patterns:

- linear (including one direction change at the array end)
- accessing the same few memory blocks multiple times (the smaller the number of memory blocks, the better the performance)

This can be seen in Table A.5. The "BU B" case requires the medium memory size and the locations are read multiple times, resulting in the worst performance. The left-heavy tree variant needs constant memory, independent of the tree size. It uses the same two locations of memory repeatedly, and as a result they are likely to stay in the cache, resulting in the best performance. The right-heavy case, on the other hand, fills the whole array front to back and then reads the array back to front, resulting in performance that lies in between the two other cases. The performance of the right-heavy tree, however, depends on the underlying GPU. Running it on an older GPU from a different vendor (AMD Radeon HD 5870), this case is the slowest, possibly because the hardware has a worse performing predictor for memory reads.

	BU B	BU LH	BU RH	BU KDM B	BU KDM LH	BU KDM RH
2	3	2	2	3	2	2
4	4	2	4	3	2	3
8	5	2	8	3	2	3
16	6	2	16	3	2	4
32	7	2	32	3	2	4
64	8	2	64	4	2	4
128	9	2	128	4	2	5
256	10	2	256	4	2	5
512	11	2	512	4	2	5
1024	12	2	1024	5	2	7
2048	13	2	2048	6	2	11

Table A.5: The memory usage for the Bottom-Up traversal variants, applied to the 3 distinct artificial tree cases.

Appendix B

Skeleton Distance Functions

Below is a list of the skeleton distance functions for the most common *BlobTree* skeletons.

B.1 Point Skeleton

For a point skeleton S_{point} the distance at p is defined as:

$$D_{S_{point}}(p) = \|p - q\|$$

B.2 Line Skeleton

For a line skeleton S_{line}, defined using the points q_0 and q_1, the distance is:

$$D_{S_{line}}(p) = \begin{cases} t \leq 0 & \rightarrow \|p - q_0\| \\ 0 < t < 1 & \rightarrow \|(q_0 - p) - t\frac{q_0 - q_1}{\|q_0 - q_1\|}\| \\ 1 \leq t & \rightarrow \|p - q_1\| \end{cases}$$

The distance t between q_0 and the projection of p onto the line is normalized to the length of the line:

$$t = \frac{(q_0 - p)(q_0 - q_1)}{\|q_0 - q_1\|}$$

B.3 Cube Skeleton

For a cube skeleton S_{cube}, centred at p with three perpendicular arms a_x, a_y and a_z, the distance is defined as:

$$\begin{aligned} D_{S_{cube}}(p) &= \sqrt{\uparrow(0, d_x - \|a_x\|)^2 + \uparrow(0, d_y - \|a_y\|)^2 + \uparrow(0, d_z - \|a_z\|)^2} \\ d_x &= \frac{a_x(p-q)}{\|a_x\|} \\ d_y &= \frac{a_y(p-q)}{\|a_y\|} \\ d_z &= \frac{a_z(p-q)}{\|a_z\|} \end{aligned}$$

This formula uses the *max* operator, represented by the $\uparrow$ symbol. The normalized distances d_x, d_y and d_z are calculated between q and the projection of p onto a_x, a_y and a_z.

B.4 Circle Skeleton

For a circle skeleton S_{circle}, with radius r, centred at q and having the normal n, the distance function is defined as:

$$D_{S_{circle}}(p) = \sqrt{(\sqrt{\|p-q\| - ((p-q)n)^2} - r)^2 + ((p-q)n)^2}$$

B.5 Disc Skeleton

For a disc skeleton S_{disc} with radius r, centred at q and having the normal n the distance is ($\uparrow$ is the *max* operator):

$$D_{S_{disc}}(p) = \sqrt{\uparrow(0, \sqrt{\|p-q\| - ((p-q)n)^2} - r)^2 + ((p-q)n)^2}$$

B.6 Cylinder Skeleton

For a cylinder skeleton $S_{cylinder}$, with radius r, centred at q having the normal n and height h, the distance is:

$$D_{S_{cylinder}}(p) = \sqrt{\uparrow (0, \sqrt{\|p-q\| - ((p-q)n)^2} - r)^2 + \uparrow (0, (p-q)n - \frac{h}{2})^2}$$

This formula uses the *max* operator, represented by the $\uparrow$ symbol.

B.7 Cone Skeleton

For a cone skeleton S_{cone}, with the tip at point q, having normal n, height h and radius r at the base, the distance is:

$$\begin{aligned} D_{S_{cone}}(p) &= \begin{cases} s \leq 0 & \rightarrow \|p-q\| \\ 0 < s < 1 & \rightarrow \|st + q - p\| \\ 1 \leq s & \rightarrow \|p-q-s\| \end{cases} \\ s &= \frac{(p-q)t}{\|t\|^2} \\ t &= hn + \frac{ru}{\|u\|} \\ u &= (p-q) - n((p-q)n) \end{aligned}$$

Appendix C

Blobtree Operations

Below is a list of the formulation of the most basic blend operators used in the *BlobTree*.

C.1 Union

The union recreates the corresponding CSG-or operator. It uses the maximum $\uparrow$ operator to return the maximum value of the sub-*BlobTrees* found at p.

$$f_R(p) = \uparrow_{n \in N} f_n(p)$$

This operator causes discontinuities in the resulting potential field.

C.2 Intersection

The intersection recreates the corresponding CSG-and operator. It uses the minimum $\downarrow$ operator to return the minimum value of the sub-*BlobTrees* found at the p.

$$f_R(p) = \downarrow_{n \in N} f_n(p)$$

The $\downarrow$ operator causes discontinuities in the resulting potential field.

C.3 Difference

To recreate the CSG difference operator the potential fields of the *BlobTrees* that are subtracted need to be inverted. This means subtracting the field value from 1. The equation

$$f_R(p) =\downarrow \left(\{f_B(p)\} \bigcup \{1 - f_n(p) | n \in N\}\right)$$

describes the difference of several *BlobTrees* N from the *BlobTree* B. The operator results in a discontinuous field.

C.4 Summation Blend

The summation blend is defined as

$$f_R(p) = \sum_{n \in N} f_n(p)$$

and produces a continuous potential field when the sub-*BlobTrees* N are continuous.

C.5 Ricci Blend

The Ricci blend [Ricci, 1973] extends the summation blend by an additional parameter r to control the shape of the blend. Depending on the values of r it creates a blend operation or a union operation, and variants in between. The Ricci blend for *BlobTrees* N is defined as:

$$f_R(p) = \sqrt[r]{\sum_{n \in N} f_n(p)^r}$$

Appendix D

Contributions of Other Researchers

Parts of this work are the results of contributions with various other researchers. While my supervisor Brian Wyvill has been involved in every single topic through discussions about potential solutions to the stated problems and by suggesting textual improvements, the involvement of other researchers were more specific. This chapter outlines the roles other researchers had during the course of the projects.

D.1 Efficient Data-Parallel Tree-Traversal for *Blob-Trees*

The work described in Chapter 3 is the result of collaboration with:

- **Jean-Luc Duprat** (University of Victoria) was involved in many discussions regarding the algorithm and pushed me to find a different approach than the original, stack-based top-down traversal. He also helped with the text and wrote the Mathematica script to generate the performance graphs from the benchmark application output.

- **Paul Lalonde** (University of Victoria) helped understanding hardware specifics that are important when porting a general purpose algorithm to SPMD hardware. He also helped with the text of the paper based on the thesis chapter.

- **Jaroslaw Rossignac** (Georgia Tech) has published several papers regarding CSG traversal and helped to refocus the paper with regards to his previous work on accelerating CSG. In addition, he helped writing the mathematical formula-

tions of the *BlobTree* traversal and improved the sections about reordering the *BlobTree* to improve performance.

D.2 *CollabBlob*: A Data-Efficient Collaborative Modelling Method using Websockets and the BlobTree for Over-the Air Networks

The following authors collaborated with me on the work presented in Chapter 4:

- **Pourya Shirazian** (University of Victoria) was involved in the implementation of the first prototype, that only included actions, but not yet the whole user interface related communication. He also helped with a first draft of the related work section and did research on the publications related to this project.
- **Saul Greenberg** (University of Calgary) helped improve the overall user experience of the final applications by suggesting the extension of the communication between participants to involve the immediate feedback system. He also provided the basic text about how collaboration between users should be and which communication channels can be used.

D.3 *Angle-Based Filleting*: Adding CSG-like control to *BlobTree* primitives

The work presented in Chapter 5 involved several discussions with **Loïc Barthe** (Université Paul Sabatier of Toulouse). He suggested several mathematical approaches to solving the corner case and in general helped with applying the GBB formulas to this use case.

D.4 Improvements in *BlobTree* Sketch-based Modelling

No significant collaborations with another researcher happened on this topic. **Jyh-Ming Lien** (George Mason University) provided the source code of his Approximate Convex Decomposition algorithm.