Seminar Report:
Massively Parallel Rendering of Complex Closed-Form Implicit Surfaces

Fabian Friederichs
Mat. Nr.: 3242333

March 18, 2021

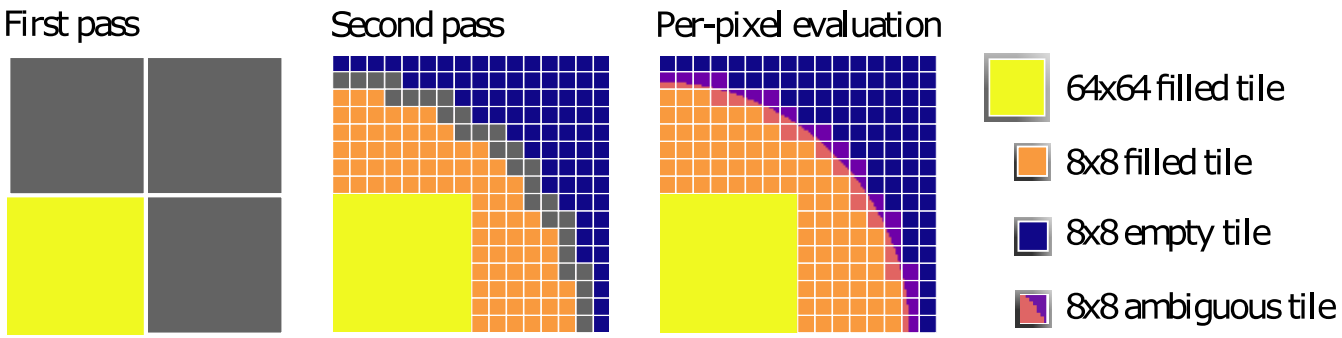


Figure 1: Three-pass process for efficiently rendering an implicit function. Figure adapted from (Keeter, 2020, Fig. 2)

Abstract

This seminar report discusses the recent work of Keeter (2020), a novel technique for efficiently rendering closed-form implicit surfaces by harnessing computational capabilities of modern GPU devices. We briefly review the necessary foundations of implicit surface representations and interval arithmetic and then discuss existing methods followed by details of the new algorithm. Finally, we analyze results presented in the paper and elaborate on the method's benefits and limitations.

1 Introduction

Implicit surface representations are an important tool when working with geometry in computer graphics, computer-aided design, robotics (May et al., 2014) and many other fields. Some operations on these surfaces are much easier and more efficient than using continuous (parametric surfaces) or discrete (polygonal meshes) boundary representations.

Implicit surfaces can be trivially combined to more complex shapes using basic set operations i.e. union, intersection and complement which all reduce down to simple min or max operations and sign flips.

Rendering implicit surfaces directly is still inefficient and most graphics hardware today is highly optimized for processing triangle meshes. Therefore, implicit surfaces are often used as an intermediate representation for geometry processing; the final visualization step often involves converting them to a polygonal mesh approximation first and then rendering them using the common pipelines like the standard rasterization procedure. This can lead to discretization artifacts and inherently limits the faithfulness of the visualization due to the limited resolution of the mesh. Other techniques like the ones related to ray marching also suffer from limited accuracy due to necessary tradeoffs for computational efficiency.

This seminar report discusses the approach of Keeter, which produces pixel perfect results while being computationally efficient by combining a hierarchical subdivision scheme with the vast compute capabilities of modern GPU devices.

In Section 2 we will review the basics on implicit surface representations, followed by an introduction to interval arithmetic in Section 3. Section 4 then reviews existing methods for rendering implicit surfaces which leads to the discussion of the new technique as described by Keeter (2020) in Section 5. The presented results are analyzed in Section 6. In Section 7 we then further discuss benefits and limitations of the technique. The report is concluded with Section 9 and some directions for further research in

Section 8.

2 Implicit Surface Representations

2.1 Implicit surfaces

Implicit surfaces are defined as level sets of some implicit function f:

$$f : \mathbb{R}^3 \to \mathbb{R}$$

An *isosurface* is then defined as the level set $\mathcal{S}_{|c}$ whose points are mapped by f to a constant *isovalue* c:

$$\mathcal{S}_{|c} = \left\{ \mathbf{p} \in \mathbb{R}^3 \mid f(\mathbf{p}) = c \right\} \tag{1}$$

The canonical implicit surface $\mathcal{S}_{|0}$ is defined to be the isosurface of f with isovalue 0. In the remainder of this report, the term *implicit surface* refers to the canonical implicit surfaces with isovalue 0.

By convention, points in $\mathbb{R}^3$ that are mapped to values < 0 are considered inside the surface, points mapped to values > 0 outside and points mapped to exactly 0 lie on the surface.

2.2 Constructive solid geometry

Two volumes enclosed by implicit surfaces can be combined to more complex shapes by union, intersection and complement operations. Union and intersection are as trivial as min and max operations and the complement is easily retrieved by flipping the sign of the implicit function.

Let V_1 and V_2 be two volumes, enclosed by two arbitrary implicit surfaces $\mathcal{S}_1$ and $\mathcal{S}_2$ respectively, which in turn are defined by the corresponding implicit functions f_1 and f_2. Then the following holds:

$$V_1 \cup V_2 = \left\{ \mathbf{p} \in \mathbb{R}^3 \mid \min\left(f_1(\mathbf{p}), f_2(\mathbf{p})\right) \leq 0 \right\} \tag{2}$$

$$V_1 \cap V_2 = \left\{ \mathbf{p} \in \mathbb{R}^3 \mid \max\left(f_1(\mathbf{p}), f_2(\mathbf{p})\right) \leq 0 \right\} \tag{3}$$

$$\mathbb{R}^3 \setminus V_1 = \left\{ \mathbf{p} \in \mathbb{R}^3 \mid -f_1(\mathbf{p}) \leq 0 \right\} \tag{4}$$

Applying these recursively is commonly called *constructive solid geometry*, short *CSG* (Wyvill et al., 1999) and results in a CSG tree structure with set operations as inner nodes and primitives like spheres, cylinders etc. at the leaves. CSG modelling is very useful for example in computer-aided design environments because it allows for arbitrary and, most importantly, exact additive and subtractive sculpting of objects.

2.3 Calculating normals

Implicit surfaces have the nice property that surface normals can be calculated trivially by normalizing the gradient of the function evaluated at the respective surface point:

$$\mathbf{n}(\mathbf{p}) = \frac{\nabla f(\mathbf{p})}{\|\nabla f(\mathbf{p})\|} \tag{5}$$

2.4 Transforming implicit surfaces

To transform an volume defined by an implicit function f with a transformation $T : \mathbb{R}^3 \to \mathbb{R}^3$, the original function is simply composed with the inverse transformation T^{-1}, resulting in a new function g of the transformed points $\mathbf{u} = T(\mathbf{p})$:

$$\begin{aligned} g(T(\mathbf{p})) &= f(\mathbf{p}) \\ g\left(T\left(T^{-1}(\mathbf{u})\right)\right) &= f\left(T^{-1}(\mathbf{u})\right) \\ g(\mathbf{u}) &= f\left(T^{-1}(\mathbf{u})\right) \end{aligned} \tag{6}$$

2.5 Signed distance fields

If the gradient of the implicit function satisfies a special case of the Eikonal equation (Equation (7)), the function is called a *signed distance function* or *-field*.

$$\|\nabla f(\mathbf{p})\| = 1 \tag{7}$$

A function that solves Equation (7) evaluates at every point in space to the shortest distance to the surface. This also allows us to easily find the closest point on a surface and thus makes tasks like collision detection extremely simple. Given a SDF and an arbitrary point $\mathbf{p} \in \mathbb{R}^3$ we can find the closest point $\mathbf{u}$ on the surface as:

$$\mathbf{u} = \mathbf{p} - f(\mathbf{p}) \cdot \nabla f(\mathbf{p}) \tag{8}$$

2.6 Closed-form vs. sampled representations

Implicit surface representations divide into two classes: exact, closed-form expressions and sampled approximations.

2.6.1 Closed-form expressions

Closed-form expressions are usually evaluated by translating the mathematical expressions into code, which produces results accurate up to machine precision. Combining primitive shapes like spheres, cylinders and so on into more complex ones is simple as explained in Section 2.2, but finding expressions for arbitrary or more complex, organic shapes is cumbersome and unintuitive. When the expressions get complex, they can also become costly to evaluate, which is problematic in terms of runtime when using, for example, ray marching to render these shapes directly, due to the high number of function evaluations necessary.

2.6.2 Sampled approximations

Sampled approximations, in contrast, model the implicit function through a finite set of samples and local reconstruction functions. Naive variants store signed distance values in a uniform grid, spanning the volume of interest. A simple reconstruction function could be trilinear interpolation of

neighboring distance values in case of a 3D grid. The finite resolution limits the accuracy of the reconstructed surfaces.

Memory consumption imposes another practical problem, which is partially solved by space partitioning data structures like octrees. The octree datastructure has the advantage that large inside or outside regions which don't contain the surface can be stored at a much coarser resolution. Close to the surface the resolution typically increases up to a predefined maximum which then limits the achievable accuracy.

Adaptively sampled distance fields (Frisken et al., 2000) reduce the necessary number of cells drastically by checking if the locally reconstructed surface exceeds some user defined error threshold. Only if that is the case, the cell is subdivided further.

Despite these shortcomings, sampled approximations of signed distance fields are widely used in geometry acquisition (Curless and Levoy, 1996; Newcombe et al., 2011), robot mapping (May et al., 2014; Oleynikova et al., 2016) and other fields, because they are intuitive and easily, locally editable which is especially important in those scenarios.

2.7 Rendering

In general there are two classical ways of rendering implicit surfaces. One is tracing rays from the camera and searching intersections, i.e. zero crossings, between the ray and the surface. The other first converts the zero-level set of the implicit function into an explicit, approximate boundary representation (e.g. triangle meshes) and then renders it using the usual rasterization pilelines.

The work of Keeter deals with rendering implicit surfaces in real-time, specifically the kind represented as closed-form expressions. After discussing another prerequisite in the next section we will give an overview of existing rendering methods in Section 4.

3 Interval Arithmetic

This section introduces some foundations of interval arithmetic and is based on (Moore et al., 2009).

Interval arithmetic extends computations on real numbers to computations on closed intervals (Moore et al., 2009, pp. 7). Instead of computing a single, scalar result which may include unknown (e.g. roundoff) error, upper and lower bounds are tracked throughout the entire calculations. These bounds guarantee that the result lies in a certain interval. The tighter the bounds, the more accurate the result can be estimated.

A closed interval is defined as the set of numbers enclosed between two bounds, including the bounds:

$$[a, b] = \{x \in \mathbb{R} \mid a \leq x \leq b\} \tag{9}$$

In interval arithmetic real numbers are now extended to *intervals* consisting of 2-tuples representing upper and lower bounds of the respective interval. The left (lower) and right (upper) endpoints of an interval are denoted by $\underline{X}$ and $\overline{X}$. An interval is then defined as:

$$X = \left[\underline{X}, \overline{X}\right] \tag{10}$$

If X and Y are two intervals and $\underline{X} = \underline{Y}$ and $\overline{X} = \overline{Y}$, X and Y are *equal*.

Intervals for which $\underline{X} = \overline{X}$ are called *degenerate* and contain only a single, real number.

3.1 Basic Operations on Intervals

3.1.1 Set Operations

If $\overline{Y} < \underline{X}$ or $\overline{X} < \underline{Y}$ the *intersection* of the two intervals is the empty set: $X \cap Y = \emptyset$.

If the intersection is non-empty, it also is an interval and defined as:

$$X \cap Y = \{z \mid z \in X \text{ and } z \in Y\} = \left[\max\{\underline{X}, \underline{Y}\}, \min\{\overline{X}, \overline{Y}\}\right] \tag{11}$$

Likewise we can define the *union* in the non empty intersection case which again also gives another interval:

$$X \cup Y = \{z \mid z \in X \text{ or } z \in Y\} = \left[\min\{\underline{X}, \underline{Y}\}, \max\{\overline{X}, \overline{Y}\}\right] \tag{12}$$

In case of an empty intersection the union is not an interval anymore, because the new set consists of two disjunct subsets. To get an interval again we include the gap between the two subsets. The resulting interval is then called the *interval hull* of X and Y:

$$X \underline{\cup} Y = \{z \mid z \in X \text{ or } z \in Y\} = \left[\min\{\underline{X}, \underline{Y}\}, \max\{\overline{X}, \overline{Y}\}\right] \tag{13}$$

Hence, interval hulls of two intervals, by definition, always contain the union of the intervals:

$$X \cup Y \subseteq X \underline{\cup} Y$$

3.1.2 Overestimation

Using interval hulls and other operations in interval arithmetic often leads to *overestimation* of the bounds, which means that the possible outcomes of an operation lie in a subset which is narrower than estimated using the interval variant of the operation.

When we have two intervals which both contain the result and the intersection is non-empty, we know that the value lies inside the intersection (Moore et al., 2009, p. 8). In practice the intersection is often narrower than the two intersected intervals and therefore leads to tighter bounds and to a better estimation of the result of interest.

3.1.3 Arithmetic Operators

The set-theoretic definition of the basic arithmetic operators *sum*, *difference*, *product* and *quotient* is the following, where $\odot$ denotes any of the above operators:

$$X \odot Y = \{x \odot y \mid x \in X, y \in Y\} \tag{14}$$

Computable expressions are derived in (Moore et al., 2009, pp. 11) by applying the operators to the inequalities which define the respective operand intervals. The results are summarized below:

$$-X = \left[-\overline{X}, -\underline{X}\right] \tag{15}$$

$$X + Y = \left[\underline{X} + \underline{Y}, \overline{X} + \overline{Y}\right] \tag{16}$$

$$X - Y = \left[\underline{X} - \overline{Y}, \overline{X} - \underline{Y}\right] \tag{17}$$

$$X \cdot Y = \left[\min\{\underline{XX}, \underline{X}\overline{Y}, \overline{X}\underline{Y}, \overline{XY}\}, \max\{\underline{XX}, \underline{X}\overline{Y}, \overline{X}\underline{Y}, \overline{XY}\}\right] \tag{18}$$

$$\frac{X}{Y} = X \cdot \left(\frac{1}{Y}\right) \text{ with } \frac{1}{Y} = \left[\frac{1}{\overline{Y}}, \frac{1}{\underline{Y}}\right] \tag{19}$$

Note that Equation (19) is only defined if Y does not contain 0. In practice, if Y does contain 0 one splits the quotient operator into several cases and introduces positive and negative infinity bounds, which still leads to valid intervals as the result is still contained. These cases are then further processed independently of one another.

3.1.4 Interval Vectors

For working with vector valued expressions Moore et al. (2009, p. 14) show how to modify the above.

An *n-dimensional interval vector* is denoted as an n-tuple of intervals $\mathbf{X} = (X_1, \ldots, X_n)$. A real vector $\mathbf{x} = (x_1, \ldots, x_n)$ is contained in $\mathbf{X}$ if $x_i \in X_i$ for $i = 1, \ldots, n$.

The intersection of two interval vectors is defined as $\mathbf{X} \cap \mathbf{Y} = (X_1 \cap Y_1, \ldots, X_n \cap Y_n)$. If one of the component intersections is empty, i.e. $X_i \cap Y_i = \emptyset$, the whole intersection is empty: $\mathbf{X} \cap \mathbf{Y} = \emptyset$.

Furthermore, $\mathbf{X}$ is a subset of $\mathbf{Y}$, i.e. $\mathbf{X} \subseteq \mathbf{Y}$, if $X_i \subseteq Y_i$ for $i = 1, \ldots, n$.

The operators defined above are usually applied at a component-wise basis to the vectors. Further, operations like the inner product and matrix-vector products can be defined (Moore et al., 2009, pp. 14).

For more details on algebraic properties of interval operations please refer to (Moore et al., 2009, pp. 31).

3.1.5 Interval Extensions of Functions

Let $f(x)$ be a function $f: \mathbb{R} \to \mathbb{R}$. In interval arithmetics we are interested in the set of all possible outputs of f given some input interval X:

$$f(X) = \{f(x) \mid x \in X\} \tag{20}$$

Or in the vector valued case:

$$\mathbf{f}(\mathbf{X}) = \{\mathbf{f}(x_1, \ldots, x_m) \mid x_1 \in X_1, \ldots, x_m \in X_m\} \tag{21}$$

For monotonic functions this set can be calculated in a straightforward manner by directly applying the function to the interval boundaries:

$$f(X) = \begin{cases} \left[f\left(\underline{X}\right), f\left(\overline{X}\right)\right], & \text{if } f \text{ is monot. inc.} \\ \left[f\left(\overline{X}\right), f\left(\underline{X}\right)\right], & \text{if } f \text{ is monot. dec.} \end{cases} \tag{22}$$

Note that upper and lower bounds swap places when the function is monotonically decreasing.

For some non-monotonic functions the resulting interval is easy to compute. Usually this involves handling several different cases for monotonic sections of the function. For example the function $f(x) = x^2,\ x \in \mathbb{R}$ gives the following set when applied to an interval (Moore et al., 2009, p. 38, Eq. (5.4)):

$$f(X) = \left\{x^2 \mid x \in X\right\}$$

By considering the cases where the input interval X is all positive, all negative or contains 0 the resulting interval evaluates exactly as (Moore et al., 2009, p. 38, Eq. (5.5)):

$$f(X) = \begin{cases} \left[\underline{X}^2, \overline{X}^2\right], & \text{if } 0 \leq \underline{X} \leq \overline{X} \\ \left[\overline{X}^2, \underline{X}^2\right], & \text{if } \underline{X} \leq \overline{X} \leq 0 \\ \left[0, \max\left\{\underline{X}^2, \overline{X}^2\right\}\right], & \text{if } \underline{X} \leq 0 \leq \overline{X} \end{cases} \tag{23}$$

This is different from applying Equation (18) (Moore et al., 2009, p. 38). For example, we have:

$$[-1, 1]^2 = [0, 1], \text{ whereas } [-1, 1] \cdot [-1, 1] = [-1, 1]$$

The exact solution provides tighter bounds than the interval product, but the interval product is still valid; it contains $[0, 1]$. This is a concrete example of overestimation caused by the interval product ignoring the *interval dependency* between the two operands.

In case of periodic functions like $f: x \in \mathbb{R} \to \sin(x)$ a valid result interval is always $\min_{x \in [c, c+T], c, T \in \mathbb{R}} f(x)$, e.g. $[-1, 1]$ for the sine function. Tighter bounds can again be acquired by dividing the function into monotonic sections and handling each case separately.

Another way to generalize functions to interval-valued ones is to replace occurrences of the arguments in the expression of the function (called *formula* in (Moore et al.,

2009)) with the respective input intervals and then applying the elementary operators and functions as shown above. Functions generalized this way are called *natural interval extensions* (Moore et al., 2009, p. 47).

Let $F = F(X_1, \ldots, X_n)$ be an interval extension. F is called *inclusion isotonic* if (Moore et al., 2009, p. 46, Def. 5.4.):

$$Y_i \subseteq X_i \text{ for } i = 1, \ldots, n \implies F(Y_1, \ldots, Y_n) \subseteq F(X1, \ldots, X_n) \tag{24}$$

An important case are rational functions which all are inclusion isotonic (Moore et al., 2009, p. 47, Lemma 5.1.).

The fundamental theorem of interval analysis then states (Moore et al., 2009, p. 47, Theorem 5.1.): If F is an inclusion isotonic interval extension of f, then

$$f(X_1, \ldots, X_n) \subseteq F(X_1, \ldots, X_n) \tag{25}$$

In other words, an inclusion isotonic interval extension always contains the image of the input interval through the original function f; which is exactly the quantity we're interested in (Equation (20)).

For arbitrary functions there are no interval extensions in general. Also, interval extensions are never unique (Moore et al., 2009, pp. 43). The fundamental theorem implies that any interval extension which contains the result is feasible, i.e. there are no restrictions on how loose the bounds can be. In practice one uses a combination of exact solutions for elementary functions and combines them using natural interval extensions according to the definitions of the basic arithmetic operators (Section 3.1.3) to keep the bounds as tight as possible.

When implementing interval arithmetic operations we usually work with floating point representations of the real numbers, hence the precision we're working with is always finite. Because of this it is important to round correctly: Results for lower bounds must always be rounded downwards, upper bounds must be rounded upwards to guarantee that the whole result interval is actually captured. Rounding modes are specified in the floating point standard ("IEEE Standard for Floating-Point Arithmetic" 2019) but not necessarily available in all contexts e.g. some GPGPU frameworks or high-level shading languages. For rendering purposes though, the errors occuring are not too severe for practical visualization purposes (Keeter, 2020).

4 Related Work

4.1 Classical Methods

The methods discussed in this subsection are know a long time and they are frequently used due to their simplicity. There are two general approaches: Direct rendering methods based on raytracing, i.e. identifing zero-crossings along camera rays, and indirect methods which first convert the implicit representation to a boundary representation which better suits the established rasterization based rendering pipelines.

4.1.1 Direct Rendering by tracing Rays

Raytracing methods work by searching for zero-crossings along viewing rays originating from the camera. For simple implicit functions, for instance simple algebraic functions, it is possible to analytically solve for these points which assures the most accurate solution (Hanrahan, 1983). There is a multitude of works dealing with this problem for specific classes of functions but a single, general approach does not exist.

When considering arbitrary closed-form implicit functions, the functions are too complicated and do not exhibit closed-form expressions for the zero-crossings in general. In this case numerical methods are the tool of choice. Classical root finding procedures like newton iterations can be employed for this purpose which lead to accurate results, but they may be too computationally expensive for real-time interactive applications. Thus, the rest of this section concentrates on existing fast but approximative solutions.

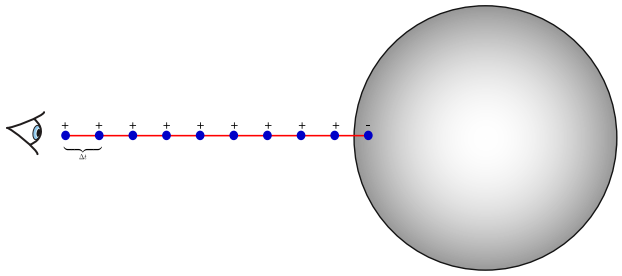


Figure 2: Naive raymarching uses a constant step size Δt to sample the rays.

The classical ray-marching method is a very simple but effective approximation. Instead of directly solving for zero-crossings along the rays, the ray is sampled inside the viewing volume of interest. In its most naive variant the sampling is done uniformly. The ray is *marched* from its origin into the scene by fixed step sizes as depicted in Figure 2. At the sample points the implicit function is evaluated. When a change in sign is observed we know that there is at least one zero crossing in the interval spanned by the last step. At this point one has several choices to compute the final result. The choice made here directly represents a trade-off between accuracy and speed:

- Return the midpoint of the last interval
- Linearly interpolate using the function values at the endpoints of the last interval (gives sensible results if

the function is approximately linear in the interval)

- Apply an iterative root finding method like the bisection method or newton's method

Another parameter is the step size; if it's too large, thin sections of the shape might be missed entirely, if it's too small, the process becomes very slow.

More advanced methods building up on this scheme try to achieve adaptive step sizes to quickly skip large empty regions of space and increase resolution close to the surface.

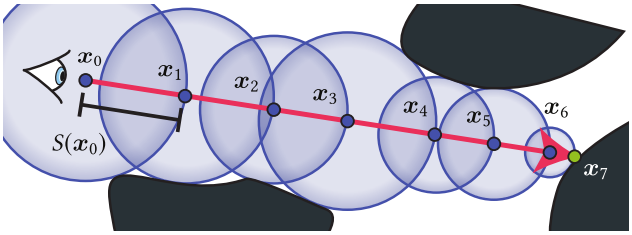


Figure 3: Sphere tracing calculates step sizes adaptively based on the shortest distance to the surface. This information is obtained trivially by evaluating the signed distance function at the sample points. (Seyb et al., 2019, Fig. 3)

In case of signed distance functions (Section 2.5) a very simple and intuitive adaptive scheme is *sphere tracing* or *sphere marching* (Hart, 1996). The idea is that, at all points in space we know the shortest distance to the surface, which is an upper bound to the size of a step in any direction we can take, without crossing the 0-isosurface (Figure 3). If the ray travels exactly perpendicular towards the surface, taking this upper bound as the step size allows us to compute the accurate result in a single step. A downside of this scheme is that it becomes very inefficient if many rays happen to be close but almost parallel to the surface. Due to the vicinity to the surface, the step size is very small and many iterations are necessary to converge. Convergence is reached when the next step size falls below some user defined threshold, meaning that we are "close enough" to the surface.

More general functions f which don't closely resemble the signed distance function have to be handled with additional care. If $\|\nabla f\| > 1$ (Seyb et al., 2019) sphere tracing will not converge because it overshoots the 0-isosurface because the maximum step size is overestimated. When the function is lipschitz continuous, convergence can still be guaranteed by dividing the estimated step size by the lipschitz constant (Hart, 1996; Seyb et al., 2019), though this might lead to suboptimal step sizes. The lipschitz constant might also be hard or impossible to compute in general (Seyb et al., 2019). Conversely, if $\|\nabla f\| \leq 1$, the step size is not optimal and the number of iterations to achieve convergence increases, but convergence is still guaranteed.

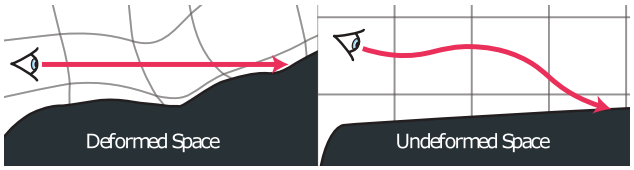


Figure 4: Nonlinear spheretracing: Instead of tracing rays through deformed space, follow the inversely deformed ray by solving an initial value problem. (Seyb et al., 2019, Fig. 4)

Seyb et al. (2019) extended the sphere tracing approach to non-linear deformations as they occur, for instance, in skeletal animation or modelling soft objects. The idea is to invert the deformation of the volume of interest and the trace deformed rays instead as shown in Figure 4. By formulating the problem as an initial value problem, Seyb et al. avoid the computation of the inverse deformation; instead they only require the inverse jacobian of the deformation. As the initial value problem is solved explicitely using euler integration, small step sizes are necessary for accurate results, which means that a lot more evaluations of the implicit function must be done compared to naive sphere tracing.

All the approaches based on raytracing of course benefit a lot from using space partitioning, e.g. octrees, kd-trees or bounding volume hierarchies for acceleration. The general weakness of all raymarching (= raytracing with via sampling) related techniques is the number of (expensive) function evaluations necessary.

4.1.2 Indirect Rendering by Extracting Isosurfaces

The other classic family of techniques all follow the same basic idea: First extract the 0-isosurface and convert it to a boundary representation, for instance quad or triangle meshes, and then render these using e.g. the standard rasterization pipeline. A well known method for the extraction and conversion step is the *marching cubes* algorithm (Lorensen and Cline, 1987). It does produce low quality results if the grid resolution is too low or if there are edge or corner features inside the cells. The latter problem can be addressed by explicitely adding edges and vertices if such features are detected in a cell and through using a different representation of the distance field to avoid interpolation errors when calculating the vertex positions (Kobbelt et al., 2001).

Another prominent method which preserves such features is called *dual contouring* (Ju et al., 2002). In contrast to the triangle soups resulting from applying marching cubes, dual contouring produces quad meshes. The vanilla version doesn't guarantee the results to be manifold (when there is more than one feature in a cell), but Greß and Klein (2004) showed how to use the half edge data structure to guarantee

the generation of manifold meshes.

All these techniques rely on a finite sampling of the function of interest. This sampling can be uniform, i.e. using a regular grid, or adaptive, e.g. by using an octree, but the achievable resolution is always finite. This inherently creates aliasing artifacts. In addition, the representation through triangle or quad meshes is already an approximation and also introduces error. Shrinking the primitives down to sub-pixel size is possible but too expensive for interactive applications. This leads to the conclusion that direct rendering methods should be preferred if achieving the most accurate result given a target image resolution is the goal, since the sampling resolution of the function automatically adapts to the viewpoint.

4.2 Particle based Methods

Particle based methods are a more exotic approach to the rendering problem. The idea behind these (Hart et al., 2002; Witkin and Heckbert, 1994) is to simulate a particle system, which is constrained such that the particles are only allowed to move in the zero-level set, i.e. the surface, and add some repulsion to distribute the particles evenly on the surface. To render the surface, one simply visualizes each particle as a small disk. Other visualization approaches are possible to achieve smooth output images. These techniques do not suffer from aliasing artifacts due to the grid structure like the ones in the previous section, but they still limit the resolution of the representation to a constant number of particles. Again, direct rendering is preferrable due to the inherent adaptivity of the sampling.

4.3 Hierarchical Methods

Duff (1992) suggested a rendering method specifically for CSG trees. The main insight of his work is that, if we evaluate an implicit function which defines some volume V for some interval $\mathbf{X}$ (by applying the formalism from Section 3):

$$Y = F(\mathbf{X})$$

, and the resulting upper bound $\overline{Y} \leq 0$ or the lower bound $\underline{Y} > 0$, then we know that this interval is completely inside or outside the volume, respectively. If neither of the above is true, the cell is called *ambiguous*. We subdivide ambiguous cells and subsequently evaluate the intervals for the child cells. This is repeated recursively until a maximum resolution is reached, e.g. when the projected cell footprint is smaller than a screen pixel.

While traversing the spatial hierarchy, we can greatly simplify the expressions to evaluate, because due to the nature of the CSG representation, small cells are only affected by a small number of leaves. Leaves which indicate that the cell is completely inside or outside the volume can be replaced by the empty set $\emptyset$ or its complement U (Duff, 1992). CSG trees for the cell which child S can then be simplified according to the following rules:

$$\emptyset \cap S \rightarrow \emptyset,\ U \cap S \rightarrow S$$
$$\emptyset \cup S \rightarrow S,\ U \cap S \rightarrow U$$

It's important to note that these simplified CSG expressions are only valid in the current cell.

In summary, the approach of Duff gives us two advantages: Firstly, we can classify large portions of the volume as inside or outside with very few evaluations and at the same time achieve adaptive resolutions up to a user defined epsilon. Secondly, while subdividing and thus shrinking the cell sizes, the number of function evaluations increases but the complexity of the expressions we have to evaluate decreases quite drastically. Compared to the raytracing based methods, we still have a direct rendering technique with adaptive resolution, but depending on the CSG tree at hand, we have to do much fewer and less complex evaluations of the underlying function.

Dyllong and Grimm (2007) implemented the idea of Duff using an extended octree data structure while keeping track of any transformations to reduce artifacts on depth-limited octrees.

The work addressed in this report (Keeter, 2020) builds mostly on the fundamental ideas discussed in this subsection. In contrast to these previous works, Keeter extends this idea from the CSG-primitive level to the level of elementary functions and operators, which makes it possible to efficiently implement the evaluation part on the GPU in form of a low-level lightweight interpreter, and avoids the need of implementing lots of primitives like spheres, cuboids and so on by hand. Further, the subdivision scheme is modified to a fixed and more shallow structure but with a high branching factor, which better suits the needs of modern SIMD architectures. Details of the new approach will be discussed in the next section.

5 Algorithm

In this section we discuss the algorithm, Keeter (2020) introduced, in detail. Section 5.1 starts with a high-level overview of the method and the subsequent sections cover details on the different components.

5.1 Overview

The algorithm is limited to the space of functions which have closed-form expressions and which are compatible with interval arithmetic. Instead of implementing a huge number of CSG primitives on the GPU, Keeter opted for a lightweight interpreter running on the GPU which works on the level of elementary functions and operators. Figure 5 illustrates the process. The main steps of the method are:

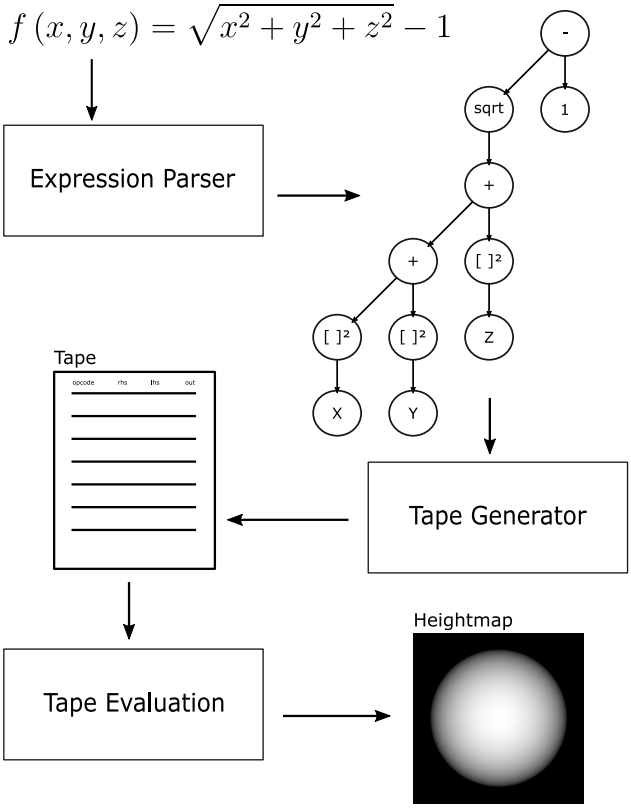


Figure 5: High level overview of the algorithm. First, the expression is parsed into a syntax tree, from that the initial tape is generated, which is then fed into the interpreter to generate the output heightmap.

Parsing the expression

The expression of the implicit function to render is parsed into a directed acyclic graph, or more precisely an abstract syntax tree, having elementary operators and functions as inner nodes and constants or input variables as leaves. This tree is then processed further in the next step.

Generating the tape

The syntax tree from the first step is converted into a sequence of *instructions*[1]: the *tape*. Each instruction consists of an *opcode*, defining the operation the interpreter shall execute, left hand side and right hand side input *slots* and the output *slot*. This sequence of instructions is encoded in a compact data format, suitable for transfer and processing on the GPU. This step, just like the previous one, must only be done once.

Tape evaluation

For some axis-aligned, rectangular or cuboid volume of interest, we now want to identify which parts of it are inside or outside, according to the implicit function encoded by the tape. To harness the computational power of the GPU, the author adopted the the idea of tiled evaluation from Nehab and Hoppe (2008). Following the idea, the input volume is first divided into a number of *tiles*. For each interval spanned by a tile, a GPU thread is launched and evaluates the tape for this tile on the interval. If the result of a tile is either completely inside or outside, we can update all pixels belonging to the tile, after which this part of the volume is finished. Otherwise, if a tile is ambiguous, the subdivision and evaluation step is repeated. This is done a fixed number of times (two subdivisions in the 2D-case, three in 3D). In the last iteration, the tapes are evaluated per pixel, arriving at the desired target resolution.

Rendering 3D imagery

3D rendering utilizes a height map generated by the last step. In the 3D-case, instead of updating an indicator image which encodes the two states "free" and "occupied" per pixel, we incrementally generate a height map by applying atomic max operations for each voxel we evaluate. After evaluation, a normal map can be generated by automatic differentiation of the resulting tapes at the voxels closest to the camera. With these two maps at hand the final rendering step is analoguous to the standard deferred rendering pipeline.

5.2 Syntax Tree

Generating the syntax tree can be done using the standard tools (e.g. derivations of *lex* (*lex(1p) — Linux manual page* 2021) and *yacc* (*yacc(1p) — Linux manual page* 2021)) for generating lexer and parser code, or by implementing them by hand as a simple recursive descent parser, as the grammar for the supported expressions is very simple. This functionality is implemented in one of the author's previous works, the *libfive* library (*libfive: Infrastructure for solid modeling*. 2019). As a simple optimization, pointers to nodes are stored in a hash map, which allows to automatically avoid repeated evaluation of the same expressions (Keeter, 2020).

5.3 Tape Generation

The tape is simply a sequence of instructions and each instruction is made up of an opcode, one or two operands and an output slot. Slots are chunks of memory for storing operands and results. Table 1 shows an example. Each row of the table represents an instruction. `lhs` and `rhs` specify the operands, and `out` specifies the output slot, i.e. the slot

[1] In the original paper the author also calls them *clauses*. In the rest of this report the terms *instruction* and *clause* may be used interchangibly.

the result of executing the instruction is stored in. Operands can be either empty (e.g. a unary operator takes only one operand), a constant value or refer to a slot. The `opcode` then specifies the instruction to be executed. There are input opcodes like `X` and `Y` to initialize a slot with the value of an input variable, the other opcodes are from the set of unary and binary operators or more complex functions like the square function or the square root.

opcode	lhs	rhs	out
X	-	-	slot 0
Y	-	-	slot 1
SQUARE	slot 0	-	slot 0
SQUARE	slot 1	-	slot 1
ADD	slot 0	slot 1	slot 1
SQRT	slot 1	-	slot 1
SUB	slot 1	1.0f	slot 0
SUB	0.5f	slot 1	slot 1
MAX	slot 0	slot 1	slot 1

Table 1: Example of a short tape. Each row represents an instruction and is defined by opcode, operands and output slot. (Keeter, 2020, Table 1)

As readers familiar with assembly language programming[2] will observe, this tape representation closely resembles how "real" CPUs work. The high-level program is first converted to assembly code (still human readable) and than mapped to machine code, a list of instructions consisting of opcodes, operands and outputs. Opcodes refer to instructions directly implemented in hardware[3] and slots would represent physical hardware registers in this analogy.

5.3.1 Syntax Tree to Tape Conversion

For the conversion from syntax tree to a sequence of instructions, Keeter uses an optimized topological sort (Kahn, 1962). This results in a sequence where nodes without dependencies, e.g. input variables and constants, are evaluated first, then the nodes wich depend on these, and so on until the whole tree is processed.

5.3.2 Register Allocation

A fixed number of slots is preallocated in advance (Keeter, 2020), which implies that the number of slots is limited. The slots obviously can and should be reused to save memory. As the whole tape representation is closely related to how assembly language and the execution of the corresponding machine instructions on real ALUs works, this problem is well researched and commonly known as the *register allocation problem* (Chaitin et al., 1981).

To solve this problem optimally, instruction reordering should be taken into account, as it has a significant impact on the result (Rawat et al., 2018). This, however, is quite a complex optimization problem. Instead, the author took a simple greedy approach, only considering the *live ranges* of variables and intermediate results. The live range of a variable (or an intermediate result) is defined as the span from the point (= instruction) of assignment to the last use of the value assigned (Chaitin et al., 1981). As long as the variable or rather its value in this slot is *live* it cannot be reused for other variables and results. Marking all live ranges lets us find the number of slots necessary at every point (= instruction) of the sequence. The number of necessary slots at every point is simply the number of overlapping live ranges, and the maximum number of registers needed is therefore the maximum number of overlapping live ranges over the whole sequence of instructions.

Table 2 shows how the register allocations can be computed by hand for the example tape above (Table 1), following the description of Keeter (2020). This algorithm can be implemented trivially. The calculation of a single live range comes with a cost of $O(n)$ w.r.t. the number of instructions n, and must be done once for every instruction, leading to a worst case runtime complexity of $O(n^2)$. The rest of the process is all linear. In practice, most live ranges in arithmetic expressions tend to be short and thus the average runtime often is closer to linear.

5.3.3 Tape Storage

Now having the sequence of instructions and the slot allocations, the information is encoded in a compact binary format (Keeter, 2020). Each instruction on the tape only takes 8 bytes. Table 3 illustrates the memory layout of a single instruction.

To support long tapes of dynamically varying size, a large chunk of memory, the *scratch buffer* as Keeter calls it, is allocated in advance (Keeter, 2020). To acquire storage for a tape, 64-instruction blocks from the scratch buffer and chained together in a linked list fashion. This scheme is called an *unrolled linked list* (Shao et al., 1994). By working with contiguous blocks of memory, the cache performance when reading and writing these data structures sequentially can be increased dramatically compared to the classical, one-item-per-node linked list scheme. During the tape pruning step, multiple GPU threads generate new, shortened tapes. They request new 64-instruction blocks from the scratch buffer to store the tapes and thus synchronization is necessary to prevent data races. The synchronization achieved by atomically incrementing a globally shared block

[2] Although rarely employed in the field of computer graphics, it is still used frequently on embedded platforms, microcontrollers or hard real-time systems and can teach a lot about memory management and efficient resource handling in general.

[3] either directly in circuitry or via microcode, whose micro instructions are, in turn, implemented in logic circuitry.

# overl. live r.	r0	r1	r2	r3	r4	r5	r6	r7	r8	opcode	lhs	lslot	rhs	rslot	out	outslot	avail. slots
1	s0									X	-	-	-	-	r0	s0	[s1]
2		s1								Y	-	-	-	-	r1	s1	[s0]
2			s0							SQUARE	r0	s0	-	-	r2	s0	[s1]
2				s1						SQUARE	r1	s1	-	-	r3	s1	[s0,s1]
1					s1					ADD	r2	s0	r3	s1	r4	s1	[s0,s1]
1						s1				SQRT	r4	s1	-	-	r5	s1	[s0]
2							s0			SUB	r5	s1	1.0f	-	r6	s0	[s1]
2								s1		SUB	0.5f	-	r5	s1	r7	s1	[s0,s1]
1									s1	MAX	r6	s0	r7	s1	r8	s1	[s0]
Max. overlapping ranges: 2 Init avail. slots with [s0, s1]																	

Table 2: Register allocation example. First, each result of an instruction in column *out* is assigned a unique name *r0 ... r8*. The *lhs* and *rhs* columns are populated with the corresponding result names according to the syntax tree. Then the live ranges are marked green in the columns *r0 ... r8*. The live range of a result *rn* is defined as the span from the first occurence in column *out* to the last occurence in *lhs* or *rhs*, i.e. it last usage, *minus one*. The number of overlapping live ranges is written in the first column. Note that the live ranges end one instruction before the last usage, this is because we can read and write the same slot in the execution of one instruction. The maximum number of overlapping ranges is 2, hence we need at most 2 slots for the whole tape. A stack of available slots is initialized with *[s0,s1]*. Then, for each instruction we pop the next free slot from the stack to store the result, and push back slots assigned to variables whose live range is about to end, making them available in the next instruction. The last cell of column *outslot* finally names the slot, the final result of the tape evaluation is stored in.

counter.

1 b. opcode	1 b. lhs	1 b. rhs	1 b. out	4 b. fp. constant

8 bytes per instruction

Table 3: Memory layout of tape instructions. The first four fields encode an index into an array of preallocated slots, the last four bytes encode an arbitrary 32-bit floating point constant.

5.4 Tape Evaluation

5.4.1 Interpreter

Keeter implemented the interpreter on the GPU using the nVIDIA CUDA GPGPU framework. The interpreter expects an input interval and a tape. It outputs the result interval of the tape and a stack of *choices*. The stack records for every min or max instruction if the choice was ambiguous (remember Section 4.3), i.e. if one of the two operands of a max or min operation was unambiguously selected and which one. A choice is either `CHOICE_BOTH` in the ambiguous case, or `CHOICE_LHS` or `CHOICE_RHS` if the right hand side or left hand side was selected for the whole interval, respectively. The recorded information is later used in the tape pruning step.

Algorithm 1 illustrates the concept in form of simplified pseudo-code. The algorithm requires the input slots for the corresponding input variables to be initialized upfront, as it takes the input intervals via slots instead of explicit parameters. The `getValue()` calls in Lines 3 and 4 return the value of the `lhs` and `rhs` operands of the instruction, either by dereferencing the corresponding slot indices or by returning the constant value contained at the end of the instruction (Keeter, 2020), as explained in Section 5.3.3.

The arithmetic operators in Lines 18 and 20 are actually implemented using the interval arithmetic operations discussed in Section 3. The same goes for all other operators and elementary functions implemented in the interpreter. For the sake of readability, these details are left out in the pseudo-code. For the actual computations, Keeter (2020) an interval arithmetic library implementation based on (Melquiond et al., 2006).

Comparing the unoptimzed, interpreted approach to direct evaluation of the expressions in CUDA (Figure 6), the overhead of the interpreter is clearly noticeable. Using the coarse-to-fine interval evaluation and tape pruning (Section 5.4.2) and having a model which specifically benefits from these optimizations, the approach becomes surprisingly efficient and outperforms the direct implementation by far. Additionally, the interpreter brings in a lot of flexibility and reduces the amount of code on the GPU side, as only few elementary operations have to be implemented to support a

Algorithm 1: The interpreter takes a tape and processes all instructions (or clauses) from beginning to end. The different cases represent opcode implementations. It returns the result interval and the stack of choices made for all min and max instructions. Adapted from (Keeter, 2020, Algorithm 1).

Precondition: Slots of the first input variable instructions initialized with the corresponding input interval vector **X**
Input: tape, preallocated array of slots
Output: result interval, stack of choices

```
1  choices ← an empty stack
2  foreach clause in tape do
3      lhs ← getValue(clause.lhs)
4      rhs ← getValue(clause.rhs)
5      switch clause.opcode do
6          case OP_MIN do
7              if lhs.upper < rhs.lower then
8                  choices.push(CHOICE_LHS)
9              else if rhs.upper < lhs.lower then
10                 choices.push(CHOICE_RHS)
11             else
12                 choices.push(CHOICE_BOTH)
13             slots[clause.out] ← min (lhs, rhs)
14         case OP_MAX do
15             Similar logic to push a choice ...
16             slots[clause.out] ← max (lhs, rhs)
17         case OP_ADD do
18             slots[clause.out] ← lhs + rhs
19         case OP_SUB do
20             slots[clause.out] ← lhs − rhs
21         ... and so on for other opcodes
22 clause ← the last clause in the tape
23 return (slots[clause.out], choices)
```

huge range of expressions without changing the kernels at all.

5.4.2 Tape Pruning

After the evaluation, the tape handed over to the subtiles is *pruned*, i.e. instructions that don't effect the result are removed. Thus, the tapes get shorter and therefore cheaper to evaluate for the children. Pruning large portions of the tapes is possible by exploiting the CSG set operations implemented by min and max (see Sections 3.1.1 and 4.3). The idea is now to remove instructions corresponding to left hand side or right hand side operands of min and max instructions from the tape of the parent tile where for the whole tile

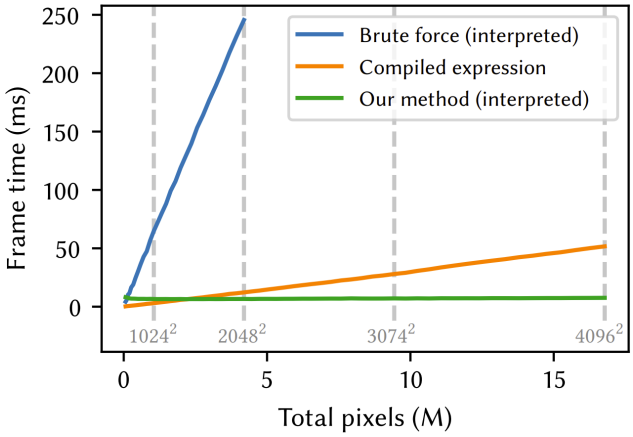


Figure 6: When leaving out tape pruning and coarse-to-fine interval evaluation, effectively using the interpreter to compute the whole expression per pixel (blue), the interpreter performs much worse than the whole expression compiled in CUDA. This clearly shows the non-negligible amount of overhead of the interpreter. Given a model wich benefits from tape pruning though, the optimzed interpreter (red) shows exceptional performance compared to the compiled expression in the CUDA (green). (Keeter, 2020, Fig. 5)

interval the respective other one of the two operands was choosen consistently, as well as all dependees of the removed instruction. This information is available from the choice stack populated by the interpreter.

The pseudo-code of the pruning algorithm is shown in Algorithm 2. First, an empty tape is created and a boolean array is initialized with a `false` entry per slot in Lines 1 and 2. To get things started, the output slot of the last instruction on the tape is marked active (Line 3).

Then the tape is processed in *reverse* order, while marking the corresponding operand slots active if they contributed to the result of the current instruction. The underlying syntax tree is effectively traversed from root to leaves.

When we encounter an instruction whose output slot is marked active, the slot is set inactive again in Line 9, because of the slot reuse induced by the register allocation during tape generation (Section 5.3.2). Then, the operands which contributed to the final result, according to the content of the stack of choices from the previous step, are set active. For `OP_MIN` and `OP_MAX` instructions with choice `CHOICE_LHS` or `CHOICE_RHS`, the inactive operand slot address is set to the respective output slot of the active operand (Lines 12 and 15). This basically removes the inactive branch from the underlying syntax tree, as the instruction only evaluates the active operand. While interpreting the new tape, multiple evaluations of the same operand in this case are avoided due to the order of evaluation from leaves to root.

Algorithm 2: The algorithm processes the tape in reverse order (= the syntax tree from root to leaves) and only adds instructions (clauses) to the new tape if it affects the final result. The active array keeps track of which slots contributed to the result during traversal. Adapted from (Keeter, 2020, Algorithm 2).

Input: tape, choices
Output: shortened output tape

```
1  output ← empty tape
2  active ← array of all false, one item per slot
3  active[final output slot] ← true
4  foreach clause in tape.reversed() do
5      choice ← CHOICE_BOTH
6      if clause.opcode ∈ [OP_MIN, OP_MAX] then
7          choice ← choices.pop()
8      if active[clause.out] then
9          active[clause.out] ← false
10         if choice == CHOICE_LHS then
11             active[clause.lhs] ← true
12             clause.rhs ← clause.lhs
13         if choice == CHOICE_RHS then
14             active[clause.rhs] ← true
15             clause.lhs ← clause.rhs
16         else
17             active[clause.lhs] ← true
18             active[clause.rhs] ← true
19         output.push_back(clause)
20 return output
```

In case of instructions other than `OP_MIN` and `OP_MAX`, both operands contribute and thus are always set active. Finally, the currently processed (active!) instruction is appended to the output tape. Instructions whose output slots where marked inactive are ignored (Line 8) and therefore removed from the original tape. When all instructions from the input tape are processed, the output tape is returned. At this point, basically all inactive branches induced my the min and max CSG operations of the underlying sytax tree are removed.

Figure 7 nicely shows how a (best-case scenario) model can benefit from tape pruning. The tapes in the final per-pixel pass shrink down to only a few instructions.

5.5 Rendering

5.5.1 2D-Rendering

Now, assembling the complete rendering algorithm, we bring in the massive parallelism achievable through our GPU devices. Algorithm 3 shows how the process works for implicit 2D-shapes.

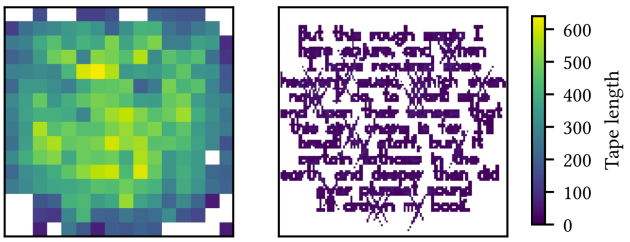


Figure 7: The figure shows the tape length in after the first (left) and second (right) subdivision step. The model at hand heavily benefits from the tape pruning optimization. White regions where identified as unambiguously outside in the respective previous step. (Keeter, 2020, Fig. 6)

Parallel foreach loops indicate the launch of a GPU thread for each iteration. This way, thousands of tiles, pixels or voxels can be processed in parallel. The algorithm can be broken down into the following three passes. The initial tape is the same for all tiles in the first pass:

1. Subdivide the image into 64×64-pixel tiles. We always subdivide into a multiple of 32 children, as this is the number of hardware threads on a SIMD unit on nVIDIA devices. This ensures good utilization of all GPU threads and reduces thread divergence. For each tile in parallel, evaluate the initial tape for the corresponding interval. If the whole resulting interval is inside the shape, color the pixels and proceed with the next tile. If the result is ambiguous, i.e. the tile contains a part of the shape's boundary, prune the tape and store the shortened tape and the ambigous tile. Tiles which evaluate to unambiguously outside are ignored.

2. For all ambiguous tiles from the last step in parallel, subdivide the ambigous tile into 8×8-pixel tiles. Proceed like in the first step: Evaluate the shortened tape from the previous step for the tile's interval. If the result is completely inside, color the corresponding pixels. Else, if the result is ambiguous, prune the tape and store shortened tape and the tile for the next step.

3. Like in the previous step, process all ambiguous tiles. But now we evaluate the tapes per pixel in parallel. If the result indicates that the pixel is inside the shape, color the pixel.

The functions `evalInterval` and `evalPixel` invoke the interpreter discussed above, where `evalPixel` evaluates the tape on the degenerate interval, only containing the pixel position, and `pruneTape` invokes the tape pruning algorithm.

This tiling approach, in contrast to (Duff, 1992) or (Dyllong and Grimm, 2007) leads to fixed and shallow recursion

depths with high branching factor, which allows efficient parallelization and predictable execution paths without the need of a call stack. That is what makes the method implementable efficiently on the GPU.

5.5.2 3D-Rendering

In the 3D-case we evaluate a $n \times n \times n$ voxel volume instead of an $n \times n$ image. The algorithm above can be be applied analogously, but we do one more subdivision step, and instead of writing directly to the 2D output image we update a heightmap using atomic max operations to get the height of the voxel closest to the camera (Keeter, 2020). As for perspective camera transformation and transformations of the objects, the coordinates are transformed before they are fed into the interpreter (by the inverse transformation, see Section 2.4).

For lighting purposes we need to estimate the surface normals. Applying finite difference schemes to the heightmap would be a simple option to do this. Instead, Keeter (2020) uses the gradient of the implicit function (Section 2.3), which leads to much more accurate results. Gradients are computed by automatic differentiation[4] of the shortened tape of the smallest region (voxel) containing the target pixel, using the map of tapes built as a side-product of Algorithm 3. Because we already have the height map, the corresponding, closest voxel can be found easily. An example of the resulting height and normal map is shown in Figures 8 and 9.

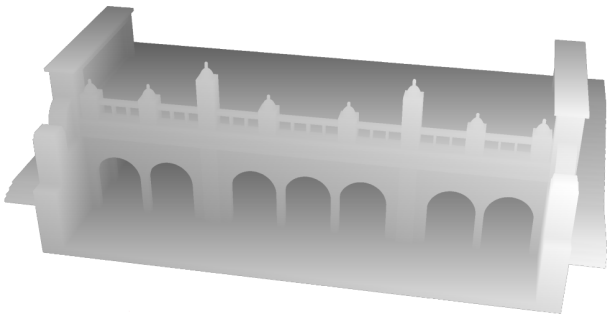


Figure 8: Height map generated by the interpreter during the 3D-rendering process. (Keeter, 2020, Fig. 7, top)

Algorithm 3: 2D-Rendering procedure. The 3D-case works analogously, but one more subdivision step is made and a height map is updated instead of a 2D-image. Adapted from (Keeter, 2020, Algorithm 3).

Input: initial tape, input region (interval vector) to be rendered
Output: Final image with all pixels colored that are located inside the implicit shape defined by the input tape.

```
activeTiles ← empty list
shortenedTapes ← empty map from regions to shortened tapes
tape ← initial input tape
image ← blank image
tiles ← input region split into a set of 64 × 64 pixel tiles
foreach tile in tiles do in parallel
    result ← evalInterval(tile, tape)
    if result.upper < 0 then
        image.fill(tile.region)
    else if result.lower < 0 then
        activeTiles.push(tile)
        shortenedTapes[tile.region] ← pruneTape(tile, tape)
activeSubtiles ← empty list
foreach tile in activeTiles do in parallel
    shortenedTape ← shortenedTapes[tile.region]
    subtiles ← tile subdivided into 8 × 8-pixel subtiles
    foreach subtile in subtiles do in parallel
        result ← evalInterval(subtile, shortenedTape)
        if result.upper < 0 then
            image.fill(subtile.region)
        else if result.lower < 0 then
            activeSubtiles.push(subtile)
            shortenedTapes[subtile.region] ← pruneTape(subtile, shortenedTape)
foreach subtile in activeSubtiles do in parallel
    shortenedTape ← shortenedTapes[subtile.region]
    subsubtiles ← subtile subdivided into 64 pixels
    foreach pixel in subsubtiles do in parallel
        result ← evalPixel(pixel, shortenedTape)
        if result.upper < 0 then
            image[pixel] ← 1
```

[4]There is a lot to say about automatic differentiation, but covering this topic in depth would be beyond the scope of this report. Basically it is the repeated application of the chain rule of differentiation to the computational graph.

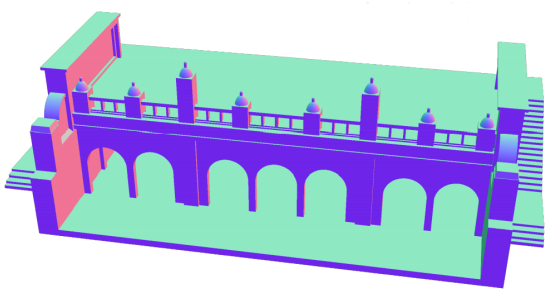


Figure 9: Normal map calculated by automatic differentiation using the results of tape evaluation. (Keeter, 2020, Fig. 7, bottom)

6 Evaluation

Keeter tested the algorithm implemented in nVIDIA CUDA on three different machines to test how the method scales on hardware of varying computational power. The three systems that were used are (Keeter, 2020):

- Macbook Pro (2013) with an NVIDIA GeForce GT 750M GPU
- Workstation built for VR/ML with an NVIDIA GTX 1080 Ti; this was a 2017 flagship desktop GPU
- AWS `p3.2xlarge` instance with a NVIDIA Tesla V100 GPU; this is the most powerful single GPU available on Amazon Web Services.

The repository of the reference implementation is available online at `https://github.com/mkeeter/mpr`. Table 4 lists the models used for evaluation. The architectural model, the text benchmark and the gears feature a high number of CSG operations and thus benefit a lot from tape pruning. The bear head sculpt model is composed out of smooth blending operations and contains relatively few CSG operations. Hence, it doesn't benefit much from tape pruning and functions like exp are expensive to evaluate. The gears model, although being relatively benign due to the number of CSG operations, features a lot of arccos and arctan evaluations which are, again, expensive to evaluate.

6.1 Visual Results

The resulting renders can be seen in Figure 10. Notice that the results are pixel-perfect, even for the calculated normals for the bear model. No discretization artifacts are visible as they would be if techniques e.g. from Section 4.1.2 were used instead.

Model	Clauses	CSG	Dimensions
Architectural model	961	465	3D
Bear head sculpt	541	27	3D
Text benchmark	6056	2354	2D
Gears	1735	374	Both

Table 4: Table of the test cases. Columns from left to right: Name of the model, Number of clauses (instructions) in the initial tape, Number of CSG operations (min and max instructions), Dimension of evaluation and rendering. (Keeter, 2020, Table 3)

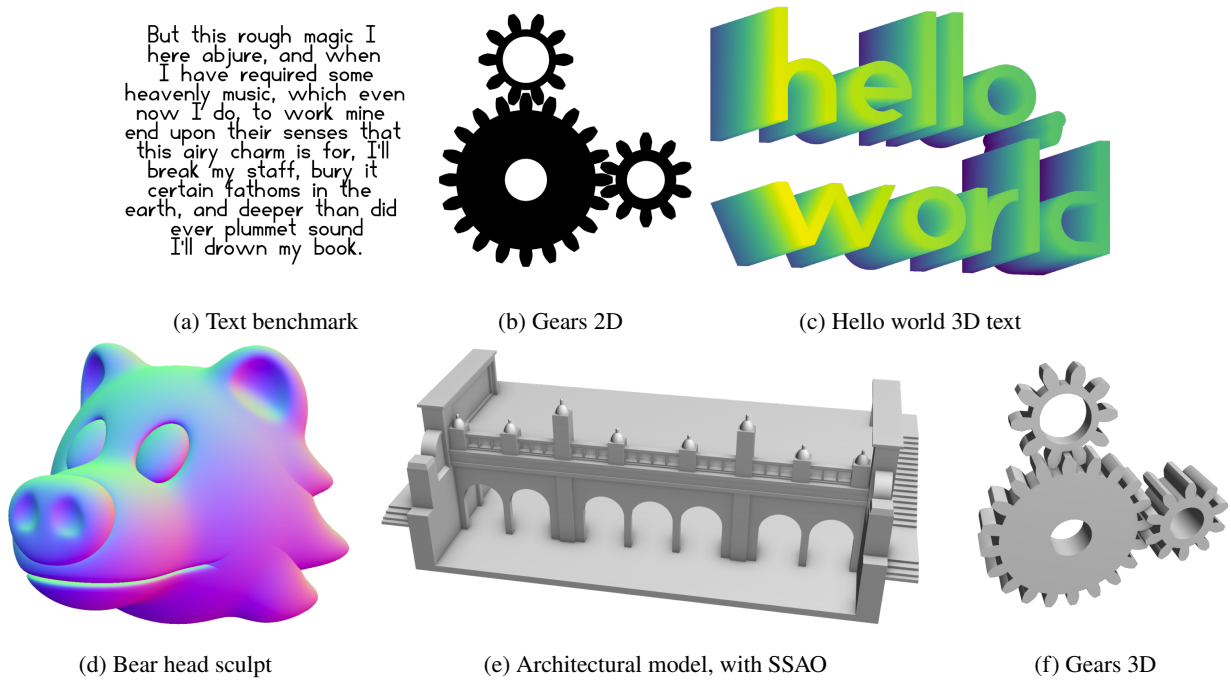
6.2 Performance

The effectiveness of the optimizations is nicely visualized by the heat maps provided in Figure 11. All the models benefit from tape pruning except the bear head. Here, the tapes don't get shortened as efficiently by the pruning algorithm due to the lack of CSG operations. Im combination with the expensive evaluation of the exp function responsible the smooth look, the model poses a worst-case scenario for the proposed method. In 3D the amount of work along object edges is higher because voxel stacks along the camera rays in these regions contain many ambiguous tiles and voxels, caused by their vicinity to the surface (Keeter, 2020).

As for a raw performance evaluation, Keeter tested different pixel / voxel grid resolutions on the three platforms specified above. Only the time to render the binary images in the 2D-case, and the time for generating the height and normal maps was measured. The resulting frame times are listed in Tables 5 and 6. The numbers look very promising for the architectural model and the 3D gears, however the frame times for the bear head sculpt model increase quickly as the resolution is increased.

7 Discussion

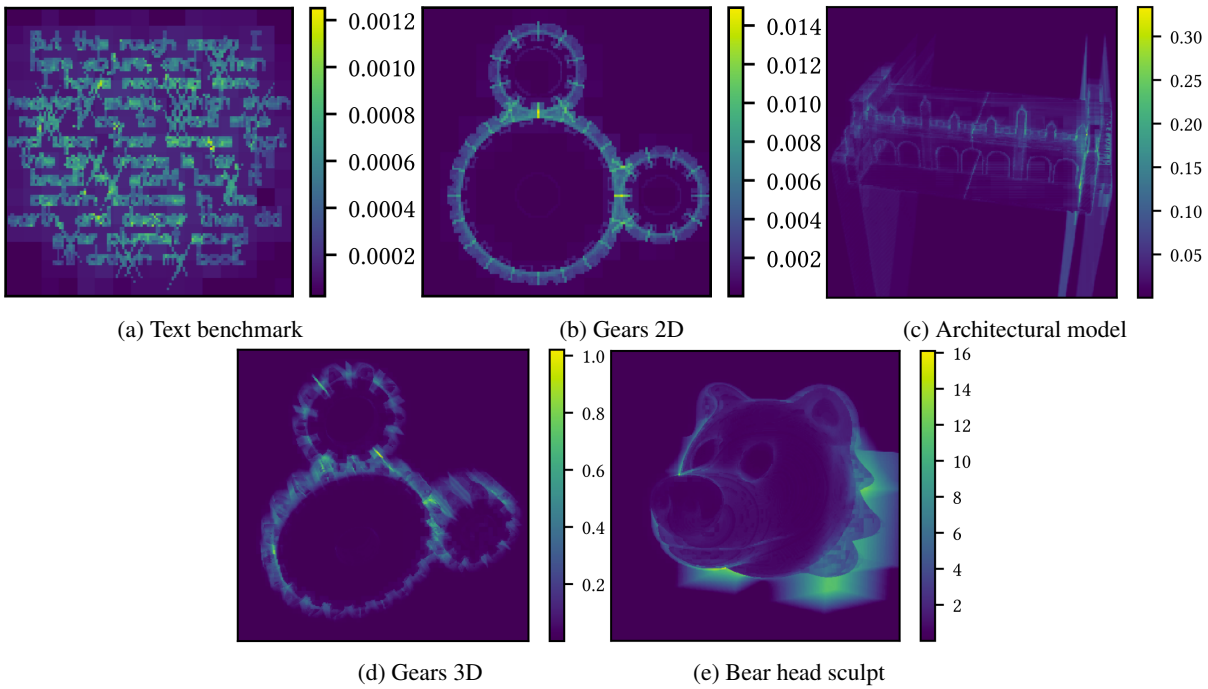
Keeter impressively showed how to translate the method of Duff (1992) to the GPU, creating a highly efficient rendering tool for closed-form implicit surfaces, especially those containing a high number of CSG operations. The method performs extremely well in the 2D case and delivers good performance in the 3D case, which is obviously way more complex due to the extra dimension. The quality of the results is pixel-perfect. Due to the high branching factor and the shallow recursion depth of the tiling approach, the massively parallel compute power of the GPU is utilized efficiently. By conducting benchmarks on three different systems, the author also demonstrated that the method scales well with more powerful hardware. Besides the use in visualization, the algorithm can be used to efficiently voxelize volumes given by implicit functions.



(a) Text benchmark (b) Gears 2D (c) Hello world 3D text

(d) Bear head sculpt (e) Architectural model, with SSAO (f) Gears 3D

Figure 10: Final renders of the tested models. Adapted from (Keeter, 2020, Figs. 1, 4, 8)



(a) Text benchmark (b) Gears 2D (c) Architectural model

(d) Gears 3D (e) Bear head sculpt

Figure 11: Heatmaps visualizing the amortized work per pixel/voxel. The 2D-cases have a resolution of 1024^2 and the values express the fraction of the initial tape evaluated at the respective pixel. In the 3D-cases a 1024^3 resolution voxel grid was evaluated and the pixel values sum the fraction of the evaluated tape over the depth stack of voxels. Adapted from (Keeter, 2020, Figs. 9, 10)

Size	GeForce GT 750M	GTX 1080 Ti	Tesla V100
256^2	17.5	8.3	5.2
512^2	14.5	6.8	4.2
1024^2	16.5	6.5	3.9
2048^2	20.7	6.6	3.9
3072^2	27.1	6.9	3.9
4096^2	35.9	7.4	4.1

(a) Text benchmark. Frametimes in ms.

Size	GeForce GT 750M	GTX 1080 Ti	Tesla V100
256^2	9.2	4.0	2.8
512^2	9.3	3.7	2.5
1024^2	12.1	3.4	2.2
2048^2	17.3	3.4	2.2
3072^2	23.4	3.7	2.3
4096^2	30.6	4.0	2.4

(b) Gears 2D. Frametimes in ms.

Table 5: Results of the 2D-benchmarks. Adapted from (Keeter, 2020, Table 4)
.

Size	GeForce GT 750M	GTX 1080 Ti	Tesla V100
256^3	34.3	5.5	3.2
512^3	73.9	9.9	5.3
1024^3	189.9	22.6	12.2
1536^3	331.9	39.3	20.8
2048^3	510.7	60.6	31.9

(a) Architectural model. Frametimes in ms.

Size	GeForce GT 750M	GTX 1080 Ti	Tesla V100
256^3	65.0	9.4	6.2
512^3	154.5	16.6	9.2
1024^3	426.2	40.3	23.1
1536^3	930.3	72.0	39.5
2048^3	–	115.4	62.0

(b) Gears 3D. Frametimes in ms.

Size	GeForce GT 750M	GTX 1080 Ti	Tesla V100
256^3	111.3	11.3	5.2
512^3	503.6	41.1	20.3
1024^3	2352.1	191.0	88.2
1536^3	–	504.2	228.3
2048^3	–	1053.2	437.3

(c) Bear head sculpt. Frametimes in ms.

Table 6: Results of the 3D-benchmarks. Adapted from (Keeter, 2020, Table 5)
.

At the same time though, the method inherits the limitation to closed-form implicit functions which are compatible to interval evaluation. For models occuring in, for instance, the context of CAD, this is not an issue at all. When it comes to data driven modelling like in the field of geometry acquisition, which usually implies octree based SDF representations like adaptively sampled distance fields (Frisken et al., 2000), there is no direct way to apply the new method. The author suggests to treat these sampled representations as black boxes. Interval evaluation would still be beneficial but the tape pruning has no effect anymore and would mostly contribute some unnecessary overhead.

Further, the method performs best when the implicit function at hand contains a lot of CSG operations, because these are the cases where the tape pruning optimization shows its full potential. In case of many overlapping primitves with large support, like in the bear head model, tape pruning doesn't have a huge effect and the performance is negatively affected by the cubic complexity of the evaluated volume. In such cases, the algorithm may be too slow for high resolution rendering purposes in real-time.

8 Future Work

The author suggests a few directions for further research. Using the method as a low-resolution voxelization tool as a preprocessing step to render the resulting voxel data with classical, established tenchiques like raytracing (Section 4.1.1) could lead to an efficient hybrid algorithm. Another idea of Keeter for optimization is the application of a more elaborate error tracking formalism like reduced affine arithmetic (Fryazinov et al., 2010) to achieve tighter bounds compared to interval arithmetic. As the height map update during 3D rendering is inefficient (because of the serialization through the atomic max operations) it may be beneficial to avoid evaluations of occluded voxels by using more advanced scheduling mechanisms. This could potentially avoid a huge number of evaluations on the back side of the models which aren't contributing to the final height map. Finally, an efficient extension of sampled representations like hierarchical voxel data is desirable.

9 Conclusion

We discussed the method of Keeter (2020) along with the necessary foundations of interval arithmetic and implicit surfaces. Classical direct rendering techniques based on raytracing deliver high quality results but need a high number of evaluations to do so. Indirect methods which first convert to discretized boundary representations are fast but suffer from discretization artifacts. The new method gives, thanks to the good utilization of the massively parallel compute capabilities of the GPU, pixel-perfect results and is very efficient for rendering objects comprising many CSG operations, but becomes increasingly slow for objects with a lack of

these. The main limitation is that it is limited to processing closed-form implicit expressions, whereas sampled representations like adaptively sampled distance fields are not supported. Besides these limitations, the use of a low level input representation and a compact interpreter implemented on the GPU greatly improves the flexibility compared to previous approaches, making tedious implementations of dozens of CSG primitives in the kernel unnecessary.

References

Chaitin, G. J., M. A. Auslander, A. K. Chandra, J. Cocke, M. E. Hopkins, and P. W. Markstein (1981). "Register allocation via coloring". In: *Computer languages* 6.1, pp. 47–57.

Curless, B. and M. Levoy (1996). "A volumetric method for building complex models from range images". In: *Proceedings of the 23rd annual conference on Computer graphics and interactive techniques*, pp. 303–312.

Duff, T. (1992). "Interval arithmetic recursive subdivision for implicit functions and constructive solid geometry". In: *ACM SIGGRAPH computer graphics* 26.2, pp. 131–138.

Dyllong, E. and C. Grimm (2007). "Verified Adaptive Octree Representations of Constructive Solid Geometry Objects." In: *SimVis*. Citeseer, pp. 223–236.

Frisken, S. F., R. N. Perry, A. P. Rockwood, and T. R. Jones (2000). "Adaptively sampled distance fields: A general representation of shape for computer graphics". In: *Proceedings of the 27th annual conference on Computer graphics and interactive techniques*, pp. 249–254.

Fryazinov, O., A. Pasko, and P. Comninos (2010). "Fast reliable interrogation of procedurally defined implicit surfaces using extended revised affine arithmetic". In: *Computers & Graphics* 34.6, pp. 708–718.

Greß, A. and R. Klein (2004). "Efficient representation and extraction of 2-manifold isosurfaces using kd-trees". In: *Graphical Models* 66.6, pp. 370–397.

Hanrahan, P. (1983). "Ray tracing algebraic surfaces". In: *Proceedings of the 10th annual conference on Computer graphics and interactive techniques*, pp. 83–90.

Hart, J. C. (1996). "Sphere tracing: A geometric method for the antialiased ray tracing of implicit surfaces". In: *The Visual Computer* 12.10, pp. 527–545.

Hart, J. C., E. Bachta, W. Jarosz, and T. Fleury (Aug. 2002). "Using Particles to Sample and Control More Complex Implicit Surfaces". In: *SMI '02: Proceedings of the Shape Modeling International 2002 (SMI'02)*. Washington, DC, USA: IEEE Computer Society, p. 129.

"IEEE Standard for Floating-Point Arithmetic" (2019). In: *IEEE Std 754-2019 (Revision of IEEE 754-2008)*, pp. 1–84.

Ju, T., F. Losasso, S. Schaefer, and J. Warren (2002). "Dual contouring of hermite data". In: *Proceedings of the 29th annual conference on Computer graphics and interactive techniques*, pp. 339–346.

Kahn, A. B. (1962). "Topological sorting of large networks". In: *Communications of the ACM* 5.11, pp. 558–562.

Keeter, M. J. (2020). "Massively parallel rendering of complex closed-form implicit surfaces". In: *ACM Transactions on Graphics (TOG)* 39.4, pp. 141–1.

Kobbelt, L. P., M. Botsch, U. Schwanecke, and H.-P. Seidel (2001). "Feature sensitive surface extraction from volume data". In: *Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, pp. 57–66.

lex(1p) — Linux manual page (2021). URL: `https://man7.org/linux/man-pages/man1/lex.1p.html` (visited on 03/18/2021).

libfive: Infrastructure for solid modeling. (2019). URL: `https://libfive.com/` (visited on 03/18/2021).

Lorensen, W. E. and H. E. Cline (1987). "Marching cubes: A high resolution 3D surface construction algorithm". In: *ACM siggraph computer graphics* 21.4, pp. 163–169.

May, S., P. Koch, R. Koch, C. Merkl, C. Pfitzner, and A. Nüchter (2014). "A Generalized 2D and 3D Multi-Sensor Data Integration Approach based on Signed Distance Functions for Multi-Modal Robotic Mapping." In: *VMV*, pp. 95–102.

Melquiond, G., S. Pion, and H. Brönnimann (2006). *(Boost) Interval Arithmetic Library*. URL: `https://www.boost.org/doc/libs/1_71_0/libs/numeric/interval/doc/interval.htm` (visited on 03/18/2021).

Moore, R. E., R. B. Kearfott, and M. J. Cloud (2009). *Introduction to interval analysis*. SIAM.

Nehab, D. and H. Hoppe (2008). "Random-access rendering of general vector graphics". In: *ACM Transactions on Graphics (TOG)* 27.5, pp. 1–10.

Newcombe, R. A., S. Izadi, O. Hilliges, D. Molyneaux, D. Kim, A. J. Davison, P. Kohi, J. Shotton, S. Hodges, and A. Fitzgibbon (2011). "KinectFusion: Real-time dense surface mapping and tracking". In: *2011 10th IEEE International Symposium on Mixed and Augmented Reality*. IEEE, pp. 127–136.

Oleynikova, H., A. Millane, Z. Taylor, E. Galceran, J. Nieto, and R. Siegwart (2016). "Signed distance fields: A natural representation for both mapping and planning". In: *RSS 2016 Workshop: Geometry and Beyond-Representations, Physics, and Scene Understanding for Robotics*. University of Michigan.

Rawat, P. S., A. Sukumaran-Rajam, A. Rountev, F. Rastello, L.-N. Pouchet, and P. Sadayappan (2018). "Associative instruction reordering to alleviate register pressure". In: *SC18: International Conference for High Performance*

Computing, Networking, Storage and Analysis. IEEE, pp. 590–602.

Seyb, D., A. Jacobson, D. Nowrouzezahrai, and W. Jarosz (2019). "Non-linear sphere tracing for rendering deformed signed distance fields". In: *ACM Transactions on Graphics* 38.6.

Shao, Z., J. H. Reppy, and A. W. Appel (1994). "Unrolling lists". In: *Proceedings of the 1994 ACM conference on LISP and functional programming*, pp. 185–195.

Witkin, A. P. and P. S. Heckbert (1994). "Using particles to sample and control implicit surfaces". In: *Proceedings of the 21st annual conference on Computer graphics and interactive techniques*, pp. 269–277.

Wyvill, B., A. Guy, and E. Galin (1999). "Extending the csg tree. warping, blending and boolean operations in an implicit surface modeling system". In: *Computer Graphics Forum*. Vol. 18. 2. Wiley Online Library, pp. 149–158.

yacc(1p) — Linux manual page (2021). URL: `https://man7.org/linux/man-pages/man1/yacc.1p.html` (visited on 03/18/2021).