Massively Parallel Rendering of Complex Closed-Form Implicit Surfaces

MATTHEW J. KEETER, Independent researcher

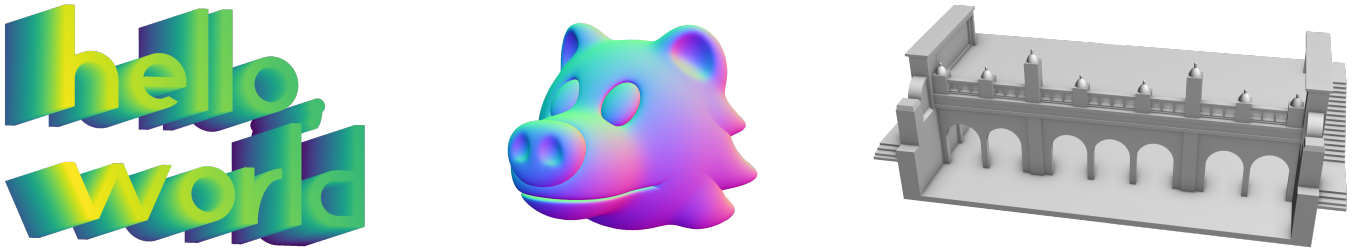


Fig. 1. An assortment of implicit surfaces rendered using our technique. Left: an extruded text string, rotated and rendered as a heightmap. Center: a bear head sculpted using smooth blending operations, with normals found by automatic differentiation. Right: a complex architectural model rendered with screen-space ambient occlusion and perspective. All models are rendered directly from their mathematical representations, without triangulation or raytracing.

We present a new method for directly rendering complex closed-form implicit surfaces on modern GPUs, taking advantage of their massive parallelism. Our model representation is unambiguously solid, can be sampled at arbitrary resolution, and supports both constructive solid geometry (CSG) and more unusual modeling operations (e.g. smooth blending of shapes). The rendering strategy scales to large-scale models with thousands of arithmetic operations in their underlying mathematical expressions. Our method only requires C^0 continuity, allowing for warping and blending operations which break Lipshitz continuity.

To render a model, its underlying expression is evaluated in a shallow hierarchy of spatial regions, using a high branching factor for efficient parallelization. Interval arithmetic is used to both skip empty regions and construct reduced versions of the expression. The latter is the optimization that makes our algorithm practical: in one benchmark, expression complexity decreases by two orders of magnitude between the original and reduced expressions. Similar algorithms exist in the literature, but tend to be deeply recursive with heterogeneous workloads in each branch, which makes them GPU-unfriendly; our evaluation and expression reduction both run efficiently as massively parallel algorithms, entirely on the GPU.

The resulting system renders complex implicit surfaces in high resolution and at interactive speeds. We examine how performance scales with computing power, presenting performance results on hardware ranging from older laptops to modern data-center GPUs, and showing significant improvements at each stage.

CCS Concepts: • **Computing methodologies** → **Rasterization**; **Volumetric models**.

Additional Key Words and Phrases: implicit surface, signed distance field, freps, octrees, rasterization, gpu, cuda

Author's address: Matthew J. Keeter, Independent researcher, matt.j.keeter@gmail.com.

0730-0301/2020/7-ART141 $15.00
https://doi.org/10.1145/3386569.3392429

ACM Reference Format:
Matthew J. Keeter. 2020. Massively Parallel Rendering of Complex Closed-Form Implicit Surfaces. *ACM Trans. Graph.* 39, 4, Article 141 (July 2020), 10 pages. https://doi.org/10.1145/3386569.3392429

1 INTRODUCTION

Implicit surfaces and functional representations are a powerful way to represent solid models [Bloomenthal and Wyvill 1997; Gomes et al. 2009]. Compared to boundary representations (e.g. triangle meshes or NURBS surfaces), they offer unambiguous inside-outside checking, easy constructive solid geometry (CSG) operations, and arbitrary resolution. In recent years, functional representations (f-reps) have been used as the kernel of both commercial [Courter 2019] and open-source [Keeter 2019] CAD packages. They are a fundamental building block in the demoscene community [Burger et al. 2002; Quilez 2008], used as a representation for generative art [Moen 2019], and even as the underlying technology for a recent PlayStation 4 game [Evans 2015].

Unlike boundary representations, implicit surfaces cannot easily be rendered in their native forms. This paper presents a new method for rendering the family of implicit surfaces represented by arbitrary closed-form arithmetic expressions, i.e., representing a sphere as

$$f(x,y,z) < 0 \text{ where } f(x,y,z) = \sqrt{x^2 + y^2 + z^2} - 1$$

This representation is particularly flexible and can be treated as an "assembly language for shapes" which is targeted by higher-level tools. The space of higher-level tools spans the gamut from advanced solid modeling packages [Allen 2019] to user-friendly content generation tools [Keeter 2015].

Our rendering strategy runs in both 2D and 3D, making efficient use of modern GPU hardware and APIs. Unlike previous work, it scales to complex expressions, maintaining interactive framerates while rendering models built from hundreds or thousands of arithmetic operations. It requires no continuity higher than C^0, which allows for extremely flexible modeling and unusual spatial transformations. Finally, it scales well with GPU power; as GPU performance

continues to increase, this rendering method will become increasingly approachable.

1.1 Overview

A 3D shape is defined by a closed-form expression of three arguments, e.g. $f(x,y,z)$. A specific point (x,y,z) is considered to be inside the model when $f(x,y,z) < 0$, and outside the model when $f(x,y,z) > 0$. This expression is converted into a linear *tape* of operations which can be executed by an interpreter running in a GPU thread.

The region of interest is divided into a set of *tiles* of a fixed size, e.g. 64^2 pixels or 64^3 voxels. Each tile is evaluated by a single GPU thread using interval arithmetic [Moore 1966], where the input intervals are the spatial region occupied by the tile. This evaluation checks whether the tile is inside the surface, outside the surface, or ambiguous. Inside tiles are marked, and outside tiles are ignored. For each ambiguous tile, we construct a shortened tape containing only parts of the expression which are active in that tile's region.

Each ambiguous tile is split into subtiles with a high branching factor. Using the shortened tapes from the previous step, each subtile is again evaluated using interval arithmetic, using the same logic as above. The process of evaluating, building shortened tapes, and subdividing is repeated twice in 2D (64^2 and 8^2 pixels) and three times in 3D (64^3, 16^3, 4^3 voxels); these sizes are chosen so that each subdivision has an even multiple of 32 children, which corresponds to GPU warp size and prevents thread divergence.

After the final round of interval evaluation, threads evaluate individual pixels or voxels in the remaining ambiguous subtiles using the last set of per-subtile shortened tapes. A visualization of subdivision levels is shown in Figure 2.

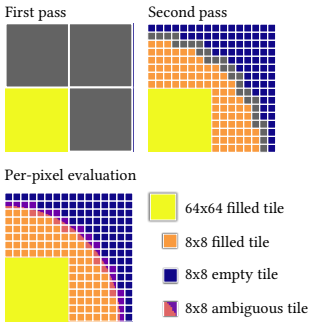


Fig. 2. Levels of subdivision in 2D rendering. The first pass (top left) renders 64×64 pixel tiles using interval arithmetic. Ambiguous tiles are subdivided into 8×8 pixel subtiles and checked with interval arithmetic again (top right). Subtiles which remained ambiguous are then evaluated pixel by pixel to produce the final image (bottom left).

1.2 Related work

Implicit surfaces, functional representations, and distance fields date back to the 1960s, and have been the subjects of extensive research in the past decades [Bloomenthal and Wyvill 1990; Gomes et al. 2009; Jones et al. 2006; Pasko et al. 1995].

This work builds most directly on Duff [1992], which describes a robust algorithm for rendering CSG models using interval arithmetic and recursion. The algorithm is not GPU-friendly, as it uses deep recursion into an octree and leads to heterogeneous workloads on each branch. Our work presents an adaptation of this algorithm which is efficiently parallelizable; in addition, we expand the model representation from CSG to pure arithmetic expressions, which increases its descriptive power. Dyllong and Grimm [2007] use a similar strategy of storing reduced shape representations in an octree, reducing the representation at each level of subdivision. Their representation is at the level of CSG primitives, rather than arithmetic expressions, and does not extend to GPU-based rendering.

Our work is also inspired by more recent research on GPU-accelerated rendering of vector graphics. Though our underlying representation is very different, we borrow the core strategy of Nehab and Hoppe [2008], which is to divide space into tiles and build a per-tile command list to execute on the GPU. Other work in this genre includes Ganacim et al. [2014] and Li et al. [2016]; we first learned of these papers from Levien [2019].

There is significant literature on converting implicit surfaces into meshes for visualization [Ho et al. 2005; Ju et al. 2002; Lorensen and Cline 1987; Schaefer et al. 2007]. Having implemented many of these algorithms, we've found it extremely difficult to make them robust. Detecting and placing vertices on sharp features is easy to get *mostly right*, and very challenging to solve in the general case, particularly if the resulting models must be manifold and not self-intersecting [Bhattacharya et al. 2015]. Detecting thin features requires a high sampling resolution; an alternative strategy is proposed in Manson and Schaefer [2010] but their algorithm leads to crinkly edges when the underlying fields are curved. The challenges of robustly meshing an arbitrary implicit surface motivate this work on direct rendering, which provides a compelling alternative.

Adaptively sampled distance fields are an alternative representation for volumetric data with a host of related research [de Figueiredo et al. 2001; Frisken et al. 2000]. ASDFs have been applied to sculptural modeling [Perry and Frisken 2001] and simulation of NC milling [Sullivan et al. 2012]. Bastos and Celes Filho [2008] describes an algorithm for GPU-accelerated rendering of ASDFs, but the ASDFs are generated from triangle meshes in an offline preprocessing step. More broadly, ASDFs are a fundamentally different representation: they're closer to voxel data, in that they're sampled at a particular resolution. Our model representation is more complex, but also encodes more information, and is not fixed to a particular sampling resolution.

Much of the existing literature on rendering implicit surfaces focuses on raytracing; sphere tracing [Hart 1995] is a common technique. Seyb et al. [2019] extends sphere tracing to compensate for non-linear deformations of signed distance fields (SDFs), but does not otherwise overlap with our functional representation. Knoll et al. [2009] describes a GPU-native algorithm for raytracing implicit surfaces using interval and affine arithmetic. This work limits itself to relatively short (< 100 operation) arithmetic expressions, and as such, does not explore strategies for reducing the evaluation workload by disabling inactive parts of the expression. Ganacim et al. [2011] builds on this work with adaptive beam sizes, while

Fryazinov et al. [2010] improves this raytracing strategy with revised affine arithmetic, but neither discusses reducing the model's representation; Ritter [2016] shows a 2D version of Fryazinov's algorithm running in WebGL.

2 GRAPH, TAPE, AND INTERPRETER

Rather than compiling a model-specific shader program / kernel, we design a general-purpose interpreter and an encoding format for the arithmetic expression that defines a shape. This is less efficient than executing a model-specific program, but our core optimization is to reduce the size of the expression (entirely on the GPU) at each recursion level and region; it would be infeasible to recompile shaders for tens of thousands of specialized programs per frame.

Our reference implementation supports algebraic (`+`, `-`, `*`, `/`, `sqrt`), transcendental (`sin`, `cos`, `asin`, `acos`, `atan`, `exp`, `log`), and piecewise (`abs`, `min`, `max`) functions. The interpreter architecture is not limited to this list; other functions that support interval evaluation and partial differentiation could also be implemented.

The input arithmetic expression can be seen as a directed acyclic graph (DAG). For example, Figure 3 shows the DAG for the annulus

$$\max\left(0.5 - \sqrt{x^2 + y^2}, \sqrt{x^2 + y^2} - 1\right) < 0 \tag{1}$$

Note that constructive solid geometry (CSG) operations are implemented with `min` (union) and `max` (intersection) operations [Pasko et al. 1995]. Our convention is to treat values < 0 as inside the shape; this is not a universal choice, e.g. Ricci [1973] uses values < 1 as inside, but this has no major implications.

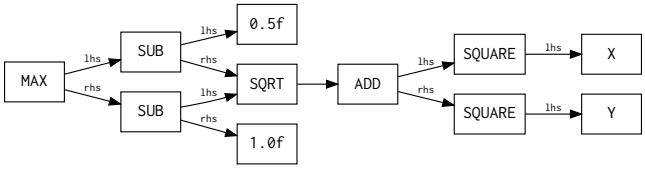


Fig. 3. Directed acyclic graph (DAG) of function 1. The abbreviations `lhs` and `rhs` refer to "left-hand side" and "right-hand side", i.e., the two arguments to a binary operation.

The raw math expression is comparable to an assembly language for solid modeling: it is the underlying truth of what's being evaluated, but humans would prefer higher-level representations. Prior to this work, we developed an open-source library and collection of shapes and transforms which can be composed to generate complex models. Details are outside the scope of this paper, but the library is documented and published online [Keeter 2019]. Our library automatically builds DAGs from mathematical expressions, storing intermediate expressions as pointers to nodes in a hash table indexed by `[opcode, lhs, rhs]` or `[constant value]`. This ensures that expressions with the same equation point to the same object, achieving common subexpression elimination.

In a preprocessing step, the input DAG is converted into a flat array of data in GPU memory. This *tape* contains an array of clauses, which each encode an operation (e.g. +, −, ∗, /, stored as an integer opcode), its arguments, and where to write its result.

This instruction tape will be interpreted by various GPU kernels, running in parallel across many threads. Each thread, when evaluating a tape, has exclusive access to a block of memory. Clause arguments and output locations are indexes into that memory, referred to as *slots*. Treating the evaluator as a primitive virtual machine, slots are equivalent to machine registers.

The graph is converted into a tape through a simple topological sort [Kahn 1962]. Deciding where to store intermediate results is more complex, and is equivalent to the problem of register allocation [Chaitin et al. 1981]. There is existing literature on simultaneous re-ordering and register allocation (e.g. [Rawat et al. 2018]), but that was deemed outside the scope of this work. Instead, we use a simple greedy algorithm: the live range of each output in the tape is calculated, and its slot is made available for reuse outside of that range. If an argument is the last use of a slot, that slot is made available *before* assigning the output of the clause; this means that a single clause may read and write the same slot.

A tape for Formula 1 is shown in Table 1. Notice that the tape uses only two slots, even though the graph has nine clauses; this shows slot re-use in action, which is critical when scaling to large expressions.

Table 1. Instruction tape for the graph in Fig. 3

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

In Figure 4, we show text rendered in a hand-made f-rep font, using a monologue from 'The Tempest' by Shakespeare. This is a large model (6056 clauses) and serves as a useful stress test. The model is built by taking the union of many individual characters, themselves unions and intersections of primitives shapes (e.g. circles and rectangles). The resulting union is a black box; tracking meta-data (e.g. bounding boxes of each character) through arbitrary equation transformations is challenging, and could be a direction for future research.

Table 2 shows the number of slots required for this model, as well as the models in Figure 1. The clause-to-slots ratio varies between models, but is significantly higher (i.e., better) for the 2D text benchmark. We hypothesize that this is because each character is an independent CSG primitive, which makes it easier to re-use slots.

Generating the initial tape is the only step in our algorithm that is executed on the CPU. In addition, these steps are only run when the model changes; panning and zooming are handled through a separate transformation described in Section 4. On a recent desktop PC, generating the tape and sending it to the GPU took 7 ms for our text benchmark, while all other examples took 1-2 ms. This is

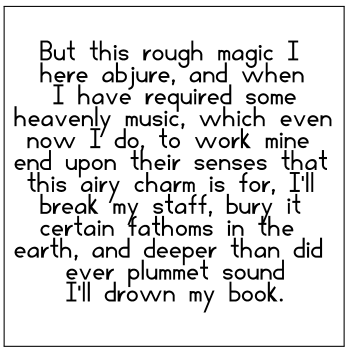


Fig. 4. Textual benchmark rendered at 2048^2 pixels, using a hand-made f-rep font. The expression has 6056 clauses, of which 2354 are `min` or `max` operations.

Table 2. Required slots for various models. A higher ratio implies more efficient slot allocation. The first three models are shown in Fig. 1; the 2D text benchmark is Fig. 4.

Model	Clauses	Slots	Ratio
"Hello, world"	328	26	12.6
Bear head sculpt	541	22	24.6
Architectural model	961	75	12.8
2D text benchmark	6056	84	72.1

acceptable for interactive editing of designs, though for very large models, it could be a target for future optimization.

2.1 Interpreter overhead

To examine the overhead of our interpreter, we modified our implementation to skip the interval arithmetic, tape shortening, and spatial subdivision; instead, it simply evaluates pixel-by-pixel, using the interpreter and encoded tape. We compare this simplified, brute-force interpreter against a kernel which directly evaluates the shape shown in Figure 4; the latter was built by pasting the mathematical expression into a CUDA C function and compiling it with `nvcc` [Nickolls et al. 2008]. As motivation for not compiling kernels while rendering, note that this offline compilation took 3.45 seconds.

The results are shown in Figure 5, which also includes our complete algorithm. The frame times for the brute-force interpreter and compiled expression are relatively linear with total pixels. Fitting a straight line to each, we see that the brute-force interpreter has a slope of roughly 58 milliseconds per megapixel, while the compiled expression is approximately 3 ms/MP, implying a 19× overhead in executing the interpreter.

This would be prohibitive, were it not for the interpreter allowing for algorithmic optimizations. The same graph shows our complete algorithm outperforming the compiled expression kernel at around 1536^2 pixels and remaining below 8 ms/frame even at 4096^2, with a negligible slope.

With this as motivation, we'll move on to explaining the other core pieces of our algorithm.

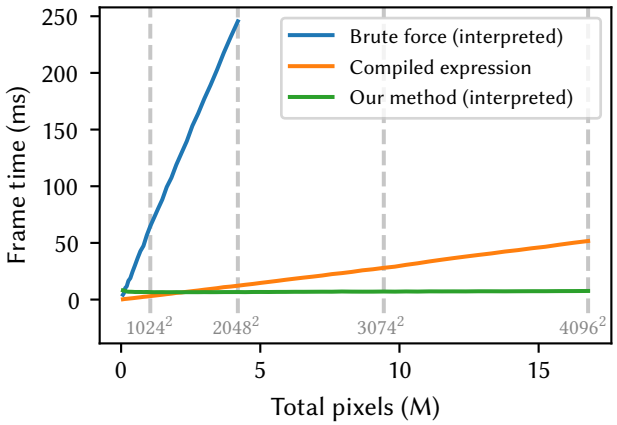


Fig. 5. Performance of brute-force interpreter, compiled expression kernel, and our complete algorithm, rendering the benchmark model from Fig. 4. Measured on an NVIDIA GTX 1080 Ti GPU, which was a 2017 flagship gaming model.

3 INTERVAL EVALUATION AND TAPE PRUNING

Interval arithmetic [Moore 1966; Moore et al. 2009; Tucker 2011] is a strategy for tracking the bounds of a computation. It is often used to track calculation errors, but also has applications in computer graphics, including CAD, rendering, and raytracing [Duff 1992; Knoll et al. 2009; Snyder 1992a,b].

In our case, we use axis-aligned spatial regions as input intervals, i.e., evaluating on intervals $X = [x_{\min}, x_{\max}]$, $Y = [y_{\min}, y_{\max}]$, $Z = [z_{\min}, z_{\max}]$. The output of $f(X, Y, Z)$ is an interval which is guaranteed to contain any value that could be found by evaluating the function at a single point (x, y, z) within the spatial region. As such, the output indicates whether the spatial region is completely inside or outside the surface:

- Upper bound of the output interval $< 0 \rightarrow$ Inside
- Lower bound of the output interval $> 0 \rightarrow$ Outside
- Otherwise $\rightarrow$ Ambiguous

Ambiguous regions either contain the shapes boundary or are a false positive (due to interval arithmetic's conservative behavior).

In addition, interval arithmetic allows us to detect which clauses in the tape are inactive (and may therefore be ignored within the region), producing a shorter version of the tape. This core optimization is detailed in Section 3.2.

To perform interval arithmetic on the GPU, we implemented an interval arithmetic library based on Melquiond et al. [2006], using GPU floating-point intrinsics [NVIDIA 2019] to control rounding modes. Though rounded intrinsics are required for mathematical correctness [Duff 1992; Mitchell 1990], they may not be necessary in practice: modifying our implementation to use normal operations showed the same performance and identical output. This is reassuring, as not all GPU programming environment include rounded intrinsics [Apple 2019]. Domain errors are handled per Melquiond et al. [2006]: partially valid inputs return an output for the valid

sub-interval, while wholly invalid inputs return NaN, which is propagated and treated as ambiguous.

3.1 Evaluation

Pseudo-code for interval evaluation is shown in Alg. 1. It is a simple interpreter loop, applying mathematical operations based on clause opcodes, with one complication: at each min or max clause, we examine the input intervals to see if either the left or right-hand argument is unambiguously selected. If that is the case, then we record the choice as CHOICE_LHS or CHOICE_RHS; otherwise, we record CHOICE_BOTH. This data is used later when shortening the tape.

Algorithm 1: Evaluate a tape, recording which branch is taken at every min and max node

```
choices ← an empty stack
foreach clause in tape do
    lhs ← getValue(clause.lhs)
    rhs ← getValue(clause.rhs)
    switch clause.opcode do
        case OP_MIN
            if lhs.upper < rhs.lower then
                choices.push(CHOICE_LHS)
            else if rhs.upper < lhs.lower then
                choices.push(CHOICE_RHS)
            else
                choices.push(CHOICE_BOTH)
            slots[clause.out] ← min(lhs, rhs)
        case OP_MAX
            Similar logic to push a choice slots[clause.out]
            ← max(lhs, rhs)
        case OP_ADD
            slots[clause.out] ← lhs + rhs
        case OP_SUB
            ...and so on for other opcodes
clause ← the last clause in the tape
return (slots[clause.out], choices)
```

This pseudo-code is simplified to present the core algorithm. For example, the getValue(...) function checks whether a clause reads from a slot or immediate and returns the appropriate value. In addition, the pseudo-code does not handle unary opcodes, making the simplifying assumption that every opcode has two arguments.

3.2 Tape pruning

The process of constructing a shortened tape is shown in Alg. 2. This pass walks through the clauses in reverse, tracking which slots are *active*. Initially, only the output slot of the final clause (which is the output of the calculation) is marked as active. Each clause unconditionally marks its arguments as active, with two exceptions: min and max. Clauses with these opcodes choose whether arguments are active by popping choices from the stack constructed in the forward evaluation pass (Alg. 1).

To provide an intuition for this pass, remember that min and max act as CSG operations: union and intersection, respectively. From this perspective, the tape pruning algorithm is checking which CSG primitives are active within a particular spatial region, and creating a reduced tape that only contains those primitives. Unlike in CSG modeling, the algorithm is operating at the level of individual arithmetic clauses, since that is our underlying representation.

Algorithm 2: Use the choice data from Alg. 1 to construct a shorter tape which only contains active clauses

```
output ← an empty tape
active ← an array of all false
active[final output slot] ← true
foreach clause in tape.reversed() do
    if clause.opcode ∈ [OP_MIN, OP_MAX] then
        choice ← choices.pop()
    else
        choice ← CHOICE_BOTH
    if active[clause.out] then
        active[clause.out] ← false
        if choice == CHOICE_LHS then
            active[clause.lhs] ← true
            clause.rhs ← clause.lhs
        else if choice == CHOICE_RHS then
            active[clause.rhs] ← true
            clause.lhs ← clause.rhs
        else
            active[clause.lhs] ← true
            active[clause.rhs] ← true
        output.push_back(clause)
return output
```

This algorithm dramatically reduces the number of active clauses, though precise improvements depend on the number of CSG operations in a tape. Figure 6 shows our text benchmark with 64^2 tiles and 8^2 subtiles colored according to how many clauses remain active in their tapes. We see a 17× reduction in average tape length for tiles and a 216× reduction for subtiles.

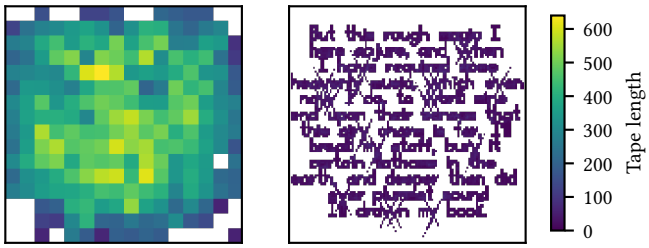


Fig. 6. Tape length in active 64×64 tiles (left) and 8×8 subtiles (right) when rendering the image shown in Figure 4 at 1024×1024 pixels. The original tape is 6056 clauses in length. Tiles have an average tape length of 356 ± 125 (mean ± standard deviation); subtiles have an average tape length of 28 ± 13. White space represents regions that have been proven inside or outside (and are therefore skipped).

3.3 Data structures

The algorithms above use abstract data structures which do not necessarily map to a GPU's architecture. Details of our implementation are given below:

- Each clause is an 8-byte value. It contains an opcode (1 byte), the output slot (1 byte), and either two input slots (each 1 byte) or one input slot (1 byte) and an immediate floating-point constant (4 bytes). The opcode encodes both the mathematical operation and whether to use the immediate constant.
- The storage slots are per-thread arrays of 128 values. For simplicity, this is hard-coded, but could be dynamically sized with the addition of `load`/`store` opcodes.
- The choices stack is also implemented as a fixed-size, per-thread array for simplicity. Position within the stack is tracked by a separate index value. Each choice only requires two bits to encode, so up to 4096 choices are packed into an array of 256 `uint32_t` integers.
- The output tape is an unrolled linked list [Shao et al. 1994], with 64 clauses per list node. We allocate a large scratch buffer (1 GB or larger), then individual GPU threads claim 64-clause chunks by atomically incrementing a single integer. This allows for tapes of arbitrary length while preserving cache-friendly access patterns.

4 RENDERING PROCEDURE

With these building blocks of interpreter, interval evaluation, and tape pruning, we can now present the full 2D rendering procedure, which was illustrated earlier in Fig. 2. The procedure evaluates and produces an $n \times n$ grid of pixels, with a fixed viewport spanning ± 1 on both axes. To allow for interactive panning and zooming, we premultiply x and y pixel coordinates with a $2x2$ transformation matrix; `X` and `Y` opcodes in the tape return these transformed coordinates, which effectively applies the transform to the entire shape.

Rendering happens in three passes:

- Interval evaluation on 64×64 pixel tiles
- Interval evaluation on 8×8 subtiles (64 per active tile)
- Per-pixel evaluation on 8×8 subtiles (64 pixels per active subtile)

Each pass after the first uses shortened tapes from the previous pass; this is the optimization that makes the algorithm efficient. Using 64× subdivision ensures that exactly two warps are mapped to each subdivided tile or subtile. This prevents thread divergence, because every thread in the warp is evaluating the same shortened tape.

This procedure is formalized in Alg. 3. Note that each of the stages (i.e., the **for..do in parallel** loops) is executed in parallel by many threads on the GPU.

Algorithm 3: Rendering procedure (2D). The subroutines evalInterval and pruneTape are shown in Algs. 1 and 2 respectively; evalPixel is identical to evalInterval, but performs floating-point (rather than interval) evaluation

```
activeTiles ← an empty list
shortenedTapes ← an empty map from regions to shortened tapes
tape ← our initial input tape
image ← a blank image (all 0) of our target size
Split the input region into a set of 64 × 64 pixel tiles
for tile 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 ← an empty list
for tile in activeTiles do in parallel
    shortenedTape ← shortenedTapes[tile.region]
    Subdivide the tile into 64 subtiles (each 8 × 8 pixels)
    for subtile 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)
for subtile in activeSubtiles do in parallel
    shortenedTape ← shortenedTapes[subtile.region]
    Subdivide the subtile into 64 pixels
    for pixel do in parallel
        result ← evalPixel (pixel, shortenedTape)
        if result.upper < 0 then
            image[pixel] = 1
```

4.1 Data structures

Lists in Alg. 3 are implemented as arrays in GPU RAM, allocated to contain the maximum possible number of items. For example, before the first pass, we allocate enough space in the activeTiles list for each tile ($1024^2/64^2 = 256$ items for a 1024×1024 2D image). When multiple threads can write to the same list, they atomically increment a global integer to uniquely "claim" an index in the list.

The shortenedTapes map is hierarchical, with one level for tiles and one for subtiles. Each level is an array of items which store both the index for a shortened tape chunk (as described in Section 3.3) and an optional index into the following level (populated only if the tile is active).

The first level of the map contains one item per tile, indexed based on that tile's position in the spatial region. Indexing into the second level is more complex, because we only want to allocate space for subtiles that are within *active* tiles. Note that each active tile can be assigned a unique value i between 0 and n_{active}. The second level of the map contains $64 \times n_{\text{active}}$ items, and the index for subtile j of active tile i is $i \times 64 + j$, where $0 \le j < 64$.

For precise implementation details, please refer to our open-source reference implementation, which is linked at the beginning of Section 5.

4.2 Implementation of 3D rendering

In 3D, our algorithm produces both a heightmap and a set of surface normals, both as 2D images at the target resolution. Figure 7 shows sample output images.

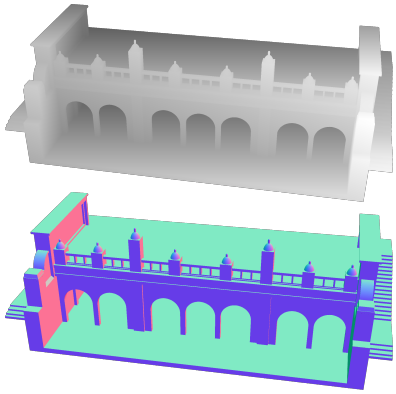


Fig. 7. Heightmap and normal images, rendering an architectural model

The render region is an $n \times n \times n$ voxel cube spanning ± 1 on all three axes, with voxels in this region mapping directly to pixels in the final 2D images (which are each $n \times n$). Coordinates within the voxel space are screen-aligned and orthographic: each pixel in the output images represent the "highest" (closest to the screen) voxel in its screen-aligned ray.

As in the 2D case, we use a transformation matrix to move the camera without modifying the model. Each voxel's original coordinates ($-1 \le (x, y, z) \le 1$, linearly spaced) are transformed with a 4×4 matrix. This transformation uses OpenGL-style homogeneous coordinates (i.e., each transformed coordinate is divided by w) to allow for rendering with perspective. Transformed coordinates are calculated and saved before tape evaluation; when opcodes in the tape ask for `X`, `Y`, or `Z`, they receive these transformed coordinates, transforming the entire shape.

The rendering procedure for the 3D heightmap is broadly the same as Alg. 3, with tiles and subtiles representing 3D volumes rather than 2D areas. Rendering uses four passes to preserve 64× subdivision at each level:

- Interval evaluation on $64 \times 64 \times 64$ tiles
- Interval evaluation on $16 \times 16 \times 16$ subtiles (64 per active tile)
- Interval evaluation on $4 \times 4 \times 4$ microtiles (64 per active subtile)
- Per-voxel evaluation on $4 \times 4 \times 4$ microtiles (64 voxels per active microtile)

The result of these four passes is a 2D heightmap of the model. Values in the heightmap are written using atomic `max` operations, as tiles may not be evaluated in a fully Z-sorted order. As a further optimization, each pass checks whether the target tile or voxel is completely occluded, returning early if that is the case. The atomic operations are not a performance bottleneck; the vast majority of effort goes into walking the tapes, which is limited by local memory bandwidth (due to reading and writing of slots).

After the heightmap is generated, a final pass calculates surface normals of visible voxels and writes those normals into a separate 2D image. Normals are found by calculating the partial derivatives $\partial f/\partial x$, $\partial f/\partial y$, $\partial f/\partial z$ near the surface of the model [Hart et al. 2002]. To calculate these derivatives, we walk through the tape in the same manner as Alg. 1, performing forward-mode automatic differentiation [Rall 1981] for each opcode rather than interval evaluation. For speed, we use the shortened tape from the smallest region containing the target pixel, found in the hierarchical map described in Section 4.1. Normals are evaluated as 32-bit floating-point values, then discretized to 8 bits for drawing.

Finally, it's possible to use the heightmap and normal images as inputs to a standard deferred rendering pipeline. For example, the architectural model in Figure 1 is drawn with screen space ambient occlusion [Bavoil and Sainz 2008], applied as a post-processing step.

5 PERFORMANCE EVALUATION

Performance was characterized across three machines, to examine a range of computing power:

- 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.

We implemented our algorithm in C++ using CUDA for GPU acceleration. This reference implementation is available online at https://github.com/mkeeter/mpr. The repository includes our implementation, benchmarks, models, and instructions to reproduce our results on AWS.

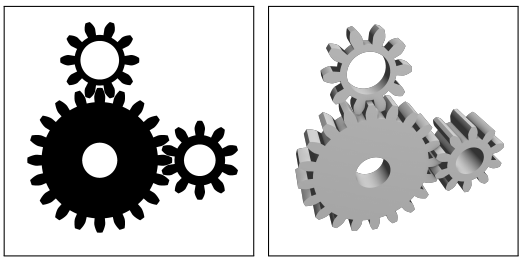
Models used in benchmarking are detailed in Table 3. The text and architectural models are large hard-surface CSG models, which are well-suited to our tape pruning algorithm. The bear sculpture is a benchmark which is less perfectly matched to our algorithm: it includes relatively few CSG clauses, and makes extensive use of smooth blending operations. These blends are both computationally expensive (using `exp` and `log`) and cannot be culled like `min` and `max` clauses. The gear model shown in Figure 8 is primarily a CSG model, but uses a mathematically exact representation for the involute curve of each tooth, which requires computing `acos` and `atan`.

Table 3. Parameters of models used in benchmarking. "Clauses" is the number of clauses in the tape; "CSG" is the number of `min` and `max` operations. The first two models are shown in Fig. 1, the 2D text benchmark in Fig. 4, and the gears in Fig. 8.

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

5.1 2D benchmarks

Table 4 shows 2D benchmarking results. The GT 750M reaches 30 frames per second for all sizes below 4096×4096, while both of the



(a) 2048^2 pixels (b) 2048^3 voxels

Fig. 8. Gear benchmark rendered in 2D and 3D. The involute curve of each tooth is mathematically exact, calculated using trigonometric functions.

more powerful GPUs render up to 4096×4096 without dipping below 60 FPS. For the larger GPUs, performance is only loosely correlated with image size, which indicates that we're far from saturating the GPU.

Table 4. 2D benchmarking results

Text benchmark

	Frame time (ms)		
Size	GeForce GT 750M	GTX 1080 Ti	Tesla V100
256^2	17.5	8.3	5.2
512^2	14.8	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

Gears (2D)

	Frame time (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

To examine the detailed behavior of the algorithm, we visualize a heatmap of work in Figure 9. From this visualization, we see that the gear model is less efficient at pruning the tape than the text benchmark. This matches our intuition: the text benchmark has many CSG operations, which are easy to simplify. Our algorithm's advantage over naive evaluation is also apparent in these heatmaps: each pixel evaluates the full tape much less than once, with work amortized over many pixels in a region.

5.2 3D benchmarks

For consistency, all 3D models are rendered with a perspective projection, which matches how they are shown in figures throughout this document. Benchmarking only includes rendering the heightmap and normal images, without any post-processing (e.g. SSAO).

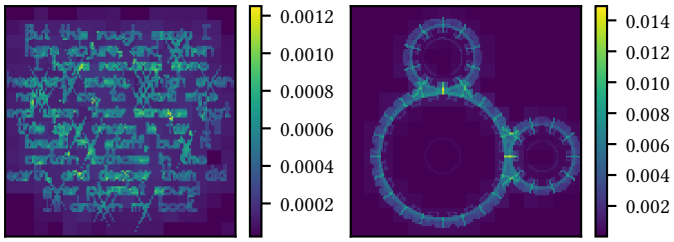


Fig. 9. Work per pixel, rendering 1024 × 1024 images. Scores are normalized by tape size: a score of 1 indicates that the tape has been fully walked once. In addition, work is amortized across pixels during interval evaluation.

Results are shown in Table 5. As expected, the CSG-heavy architecture scene performs the best, followed by the `acos`/`atan`-heavy gear, with the smoothly blended bear sculpture performing the worst. On the most powerful GPU, both the architecture and gears model exceed 40 FPS at 1024^3 voxels (producing a 1024×1024 image); the bear lags behind at 11 FPS.

These results are extremely promising in terms of GPU scaling: the Tesla V100 is around 2× faster than the GTX 1080 Ti and $15 - 25\times$ faster than the GT 750M. This implies that as GPU power and parallelism continues to improve, our algorithm will be able to scale with the additional resources.

Table 5. 3D benchmarking results

Architectural model

	Frame time (ms)		
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

Gears (3D)

	Frame time (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

Bear head sculpt

	Frame time (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	88.3
1536^3	—	504.2	228.3
2048^3	—	1053.2	437.3

A heatmap of effort is shown in Figure 10, using the same amortized and normalized scale as Fig. 9. In the 3D case, each pixel in the

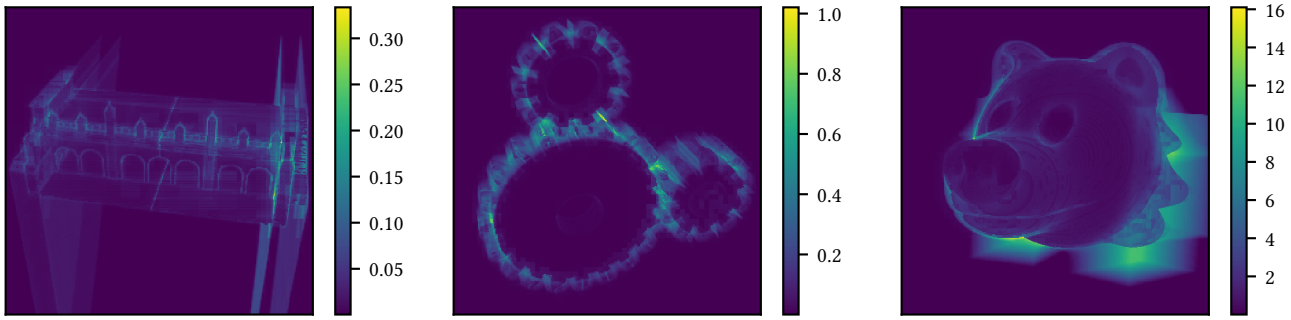


Fig. 10. Normalized amortized work per pixel, rendering 1024^3 voxels

heatmap sums work for every voxel in its stack. As expected, the models with more CSG operations perform dramatically less work per pixel. Interestingly, we see behavior similar to that of marching spheres: extra work is done at the edges of the model, where interval evaluation can't skip entire regions and must evaluate down to smaller regions or individual voxels. Finally, the visualizations hint at inefficiencies in the shapes themselves: the "echoes" below the architectural model suggest that adding a Z clipping plane to the model would improve its rendering time, while the glow on the bear's right side suggests that the many smooth blends are leading to less useful interval evaluation results.

6 CONCLUSIONS AND FUTURE WORK

We have presented a new method for rendering complex closed-form implicit surfaces on the GPU, without triangulation or conventional raytracing. This work has immediate applications for fast visualizations in solid modeling and computer-aided design, but could also be used for simulation or as a lightweight, robust kernel for user-generated content. Our algorithm scales from laptop to desktop to server graphics cards, reaping speed improvements at each step; we anticipate it being increasingly useful as GPUs continue to increase in power and parallelism.

Though we used it to render a 2D image from a particular camera angle, this algorithm can also perform a full voxelization (in the 3D case) by turning off depth culling. This suggests potential synergies with other GPU-first algorithms for rendering voxel data, e.g. Sparse Voxel Octrees [Laine and Karras 2010] and GVDB [Hoetzlein 2016]. One particularly interesting direction is to use our algorithm for fast voxelization at a lower resolution (e.g. 512^3), then render at full screen resolution using conventional raytracing on the voxel data.

Performance improvements to our core algorithm are another direction for future research. The interpreter loop is one high-value target for further optimization. The ideal would be to dynamically generate executable code on the GPU, eliminating interpreter overhead altogether; unfortunately, there is no support for on-device JIT compilation in modern GPU APIs. A second optimization could be exploring reduced affine arithmetic [Fryazinov et al. 2010] to more tightly bound intervals. Additionally, our implementation of depth culling is relatively naive; scheduling work on the GPU to avoid evaluating occluded voxels could improve performance.

Finally, our algorithm is limited to rendering pure functional representations; a powerful representation, but not one in common use. In previous work [Keeter 2019], we sidestepped this limitation with black-box "oracles", which are an escape hatch for interoperability with other representations: anything that cannot be expressed as a pure math expression can be boxed into an oracle, which exposes point, interval, and derivative evaluation. However, this work was limited to meshing on the CPU; integrating this kind of API with our GPU rendering pipeline would allow for interoperability with other representations, including meshes and raw voxel data.

ACKNOWLEDGMENTS

I'm incredibly grateful to everyone who read drafts and gave feedback as I worked on this project: Jonathan Bachrach, Blake Courter, Neil Gershenfeld, Raph Levien, Brian Merchant, Doug Moen, and Amira Abdel Rahman. Special thanks to Martin Galese, who both provided feedback and loaned me time on his VR/ML workstation for benchmarking.

The architectural model in Figure 1 is based on a design by Jennifer Keeter, for whom I am very grateful. The bear head is based on a design by Hazel Fraticelli and Anthony Taconi; thanks for providing a particularly challenging benchmark! The gear model uses a clever closed-form expression for an involute curve derived by Peter Fedak.

I'd like to thank the anonymous reviewers for their feedback and insights. Finally, thanks to my colleagues at Formlabs for encouraging my independent research, and the folks at nTopology for their support of the `libfive` kernel.

REFERENCES

George Allen. 2019. nTopology Modeling Technology. https://ntopology.com/wp-content/uploads/2019/12/nTop-Modeling-Tech-WhitePaper-v3.pdf.

Apple. 2019. Metal Shading Language Specification (version 2.2). https://developer.apple.com/metal/Metal-Shading-Language-Specification.pdf.

Thiago Bastos and Waldemar Celes Filho. 2008. GPU-accelerated Adaptively Sampled Distance Fields. *2008 IEEE International Conference on Shape Modeling and Applications* (2008), 171–178.

Louis Bavoil and Miguel Sainz. 2008. Screen Space Ambient Occlusion - Nvidia. (Nov. 2008). https://developer.download.nvidia.com/SDK/10.5/direct3d/Source/ScreenSpaceAO/doc/ScreenSpaceAO.pdf

Arindam Bhattacharya, Ross Vasko, and Rephael Wenger. 2015. *Shrec: Sharp reconstruction of isosurface.* Technical Report. The Ohio State University. Report TR: OSU-CISRC-11/15-TR22.

Jules Bloomenthal and Brian Wyvill. 1990. Interactive techniques for implicit modeling. *SIGGRAPH Comput. Graph.* 24, 2 (Feb. 1990), 109–116. https://doi.org/10.1145/91394.91427

Jules Bloomenthal and Brian Wyvill. 1997. *Introduction to Implicit Surfaces*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

Boris Burger, Ondrej Paulovic, and Milos Hasan. 2002. Realtime Visualization Methods in the Demoscene. http://old.cescg.org/CESCG-2002/BBurger/index.html. *Proceedings of the Central European Seminar on Computer Graphics* (April 2002), 205–218.

Gregory J. Chaitin, Marc A. Auslander, Ashok K. Chandra, John Cocke, Martin E. Hopkins, and Peter W. Markstein. 1981. Register Allocation via Coloring. *Comput. Lang.* 6, 1 (Jan. 1981), 47–57. https://doi.org/10.1016/0096-0551(81)90048-5

Blake Courter. 2019. How implicits succeed where B-reps fail. https://ntopology.com/blog/2019/03/28/how-implicits-succeed-where-b-reps-fail/.

L.H. de Figueiredo, L. Velho, and J.B. de Oliveira. 2001. Revisiting adaptively sampled distance fields. In *Computer Graphics and Image Processing, 2001 Proceedings of XIV Brazilian Symposium on*. 377–. https://doi.org/10.1109/SIBGRAPI.2001.963083

Tom Duff. 1992. Interval arithmetic recursive subdivision for implicit functions and constructive solid geometry. In *Proceedings of the 19th annual conference on Computer graphics and interactive techniques (SIGGRAPH '92)*. ACM, New York, NY, USA, 131–138. https://doi.org/10.1145/133994.134027

Eva Dyllong and Cornelius Grimm. 2007. Verified Adaptive Octree Representations of Constructive Solid Geometry Objects. 223–236.

Alex Evans. 2015. Learning from Failure: a Survey of Promising, Unconventional and Mostly Abandoned Renderers for 'Dreams PS4', a Geometrically Dense, Painterly UGC Game. http://media.lolrus.mediamolecule.com/AlexEvans_SIGGRAPH-2015.pdf.

Sarah F. Frisken, Ronald N. Perry, Alyn P. Rockwood, and Thouis 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 (SIGGRAPH '00)*. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 249–254. https://doi.org/10.1145/344779.344899

Oleg Fryazinov, Alexander Pasko, and Peter Comninos. 2010. Fast reliable interrogation of procedurally defined implicit surfaces using extended revised affine arithmetic. *Computers & Graphics* 34 (Dec. 2010), 708–718. https://doi.org/10.1016/j.cag.2010.07.003

F. Ganacim, L. H. Figueiredo, and D. Nehab. 2011. Beam Casting Implicit Surfaces on the GPU with Interval Arithmetic. In *2011 24th SIBGRAPI Conference on Graphics, Patterns and Images*. 72–77.

Francisco Ganacim, Rodolfo S. Lima, Luiz Henrique de Figueiredo, and Diego Nehab. 2014. Massively-Parallel Vector Graphics. *ACM Trans. Graph.* 33, 6, Article Article 229 (Nov. 2014), 14 pages. https://doi.org/10.1145/2661229.2661274

Abel J. P. Gomes, Joaquim Jorge Voiculescu, Brian Wyvill, and Callum Galbraith. 2009. *Implicit Curves and Surfaces: Mathematics, Data Structures and Algorithms*. Springer.

John Hart. 1995. Sphere Tracing: A Geometric Method for the Antialiased Ray Tracing of Implicit Surfaces. *The Visual Computer* 12 (June 1995). https://doi.org/10.1007/s003710050084

John Hart, Ed Bachta, Wojciech Jarosz, and Terry Fleury. 2002. Using particles to sample and control more complex implicit surfaces. *Proceedings - SMI 2002: Shape Modeling International 2002*, 129 – 136. https://doi.org/10.1109/SMI.2002.1003537

Chien-chang Ho, Fu-che Wu, Bing-yu Chen, and Ming Ouhyoung. 2005. Cubical marching squares: Adaptive feature preserving surface extraction from volume data. *Computer Graphics Forum* 24 (2005), 2005.

Rama Karl Hoetzlein. 2016. GVDB: Raytracing Sparse Voxel Database Structures on the GPU. In *Proceedings of High Performance Graphics (HPG '16)*. Eurographics Association, Goslar Germany, Germany, 109–117. https://doi.org/10.2312/hpg.20161197

Mark W. Jones, J. Andreas Baerentzen, and Milos Sramek. 2006. 3D Distance Fields: A Survey of Techniques and Applications. *IEEE Transactions on Visualization and Computer Graphics* 12, 4 (July 2006), 581–599. https://doi.org/10.1109/TVCG.2006.56

Tao Ju, Frank Losasso, Scott Schaefer, and Joe Warren. 2002. Dual Contouring of Hermite Data. In *Proceedings of the 29th Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH '02)*. ACM, New York, NY, USA, 339–346. https://doi.org/10.1145/566570.566586

A. B. Kahn. 1962. Topological Sorting of Large Networks. *Commun. ACM* 5, 11 (Nov. 1962), 558–562. https://doi.org/10.1145/368996.369025

Matthew Keeter. 2015. Antimony. https://mattkeeter.com/projects/antimony.

Matthew Keeter. 2019. libfive: Infrastructure for solid modeling. https://libfive.com.

Aaron Knoll, Younis Hijazi, Andrew Kensler, Mathias Schott, and Charles Hansen. 2009. Fast Ray Tracing of Arbitrary Implicit Surfaces with Interval and Affine Arithmetic. *Comput. Graph. Forum* 28 (March 2009), 26–40. https://doi.org/10.1111/j.1467-8659.2008.01189.x

Samuli Laine and Tero Karras. 2010. Efficient Sparse Voxel Octrees. In *Proceedings of the 2010 ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games (I3D '10)*. ACM, New York, NY, USA, 55–63. https://doi.org/10.1145/1730804.1730814

Raph Levien. 2019. 2D Graphics on Modern GPU. https://raphlinus.github.io/rust/graphics/gpu/2019/05/08/modern-2d.html.

Rui Li, Qiming Hou, and Kun Zhou. 2016. Efficient GPU Path Rendering Using Scanline Rasterization. *ACM Transactions on Graphics* 35, 6 (2016).

William E. Lorensen and Harvey E. Cline. 1987. Marching cubes: A high resolution 3D surface construction algorithm. *SIGGRAPH Comput. Graph.* 21, 4 (Aug. 1987), 163–169. https://doi.org/10.1145/37402.37422

Josiah Manson and Scott Schaefer. 2010. Isosurfaces Over Simplicial Partitions of Multiresolution Grids. *Comput. Graph. Forum* 29 (May 2010), 377–385. https://doi.org/10.1111/j.1467-8659.2009.01607.x

Guillaume Melquiond, Sylvain Pion, and Hervé Brönnimann. 2006. Interval Arithmetic Library. https://www.boost.org/doc/libs/1_71_0/libs/numeric/interval/doc/interval.htm.

D. P. Mitchell. 1990. Robust Ray Intersection with Interval Arithmetic. In *Proceedings on Graphics Interface '90*. Canadian Information Processing Society, CAN, 68–74.

Doug Moen. 2019. curv: a language for making art using mathematics. https://github.com/curv3d/curv.

Ramon E. Moore. 1966. *Interval Analysis*. Prentice-Hall.

Ramon E. Moore, R. Baker Kearfott, and Michael J. Cloud. 2009. *Introduction to Interval Analysis*. Society for Industrial and Applied Mathematics. https://doi.org/10.1137/1.9780898717716 arXiv:https://epubs.siam.org/doi/pdf/10.1137/1.9780898717716

Diego Nehab and Hugues Hoppe. 2008. Random-access Rendering of General Vector Graphics. In *ACM SIGGRAPH Asia 2008 Papers (SIGGRAPH Asia '08)*. ACM, New York, NY, USA, Article 135, 10 pages. https://doi.org/10.1145/1457515.1409088

John Nickolls, Ian Buck, Michael Garland, and Kevin Skadron. 2008. Scalable Parallel Programming with CUDA. *Queue* 6, 2 (March 2008), 40–53. https://doi.org/10.1145/1365490.1365500

NVIDIA. 2019. CUDA Math API. https://docs.nvidia.com/cuda/pdf/CUDA_Math_API.pdf.

Alexander Pasko, Valery Adzhiev, Alexei Sourin, and Vladimir Savchenko. 1995. Function representation in geometric modeling: concepts, implementation and applications. *The Visual Computer* 11 (Aug. 1995), 429–446. https://doi.org/10.1007/BF02464333

Ronald N. Perry and Sarah F. Frisken. 2001. Kizamu: a system for sculpting digital characters. In *Proceedings of the 28th annual conference on Computer graphics and interactive techniques (SIGGRAPH '01)*. ACM, New York, NY, USA, 47–56. https://doi.org/10.1145/383259.383264

Inigo Quilez. 2008. 3D SDF functions. https://www.iquilezles.org/www/articles/distfunctions/distfunctions.htm.

Louis B. Rall. 1981. *Automatic Differentiation: Techniques and Applications*. Lecture Notes in Computer Science, Vol. 120. Springer. https://doi.org/10.1007/3-540-10861-0

Prashant Singh Rawat, Aravind Sukumaran-Rajam, Atanas Rountev, Fabrice Rastello, Louis-Noël Pouchet, and P. Sadayappan. 2018. Associative Instruction Reordering to Alleviate Register Pressure. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis (SC '18)*. IEEE Press, Piscataway, NJ, USA, Article 46, 13 pages. https://doi.org/10.1109/SC.2018.00049

A. Ricci. 1973. A constructive geometry for computer graphics. *Comput. J.* 16, 2 (Jan. 1973), 157–160. https://doi.org/10.1093/comjnl/16.2.157 arXiv:http://oup.prod.sis.lan/comjnl/article-pdf/16/2/157/1060001/160157.pdf

Leonard Ritter. 2016. Affine Arithmetic Joint Range. https://www.shadertoy.com/view/4sV3zm.

Scott Schaefer, Tao Ju, and Joe Warren. 2007. Manifold Dual Contouring. *IEEE Transactions on Visualization and Computer Graphics* 13, 3 (May 2007), 610–619. https://doi.org/10.1109/TVCG.2007.1012

Dario Seyb, Alec Jacobson, Derek Nowrouzezahrai, and Wojciech Jarosz. 2019. Nonlinear Sphere Tracing for Rendering Deformed Signed Distance Fields. *ACM Trans. Graph.* 38, 6, Article 229 (Nov. 2019), 12 pages. https://doi.org/10.1145/3355089.3356502

Zhong Shao, John H. Reppy, and Andrew W. Appel. 1994. Unrolling Lists. In *Proceedings of the 1994 ACM Conference on LISP and Functional Programming (LFP '94)*. ACM, New York, NY, USA, 185–195. https://doi.org/10.1145/182409.182453

John M. Snyder. 1992a. *Generative Modeling for Computer Graphics and CAD: Symbolic Shape Design Using Interval Analysis*. Academic Press Professional, Inc., USA.

John M. Snyder. 1992b. Interval Analysis for Computer Graphics. In *Proceedings of the 19th Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH '92)*. Association for Computing Machinery, New York, NY, USA, 121–130. https://doi.org/10.1145/133994.134024

Alan Sullivan, Huseyin Erdim, Ronald N. Perry, and Sarah F. Frisken. 2012. High accuracy NC milling simulation using composite adaptively sampled distance fields. *Comput. Aided Des.* 44, 6 (June 2012), 522–536. https://doi.org/10.1016/j.cad.2012.02.002

Warwick Tucker. 2011. *Validated Numerics: A Short Introduction to Rigorous Computations*. Princeton University Press. http://www.jstor.org/stable/j.ctvcm4g18