Synchronized tracing of primitive-based implicit volumes

Cédric Zanni

Synchronized tracing of primitive-based implicit volumes

ZANNI CÉDRIC, Université de Lorraine, CNRS, Inria, LORIA, France

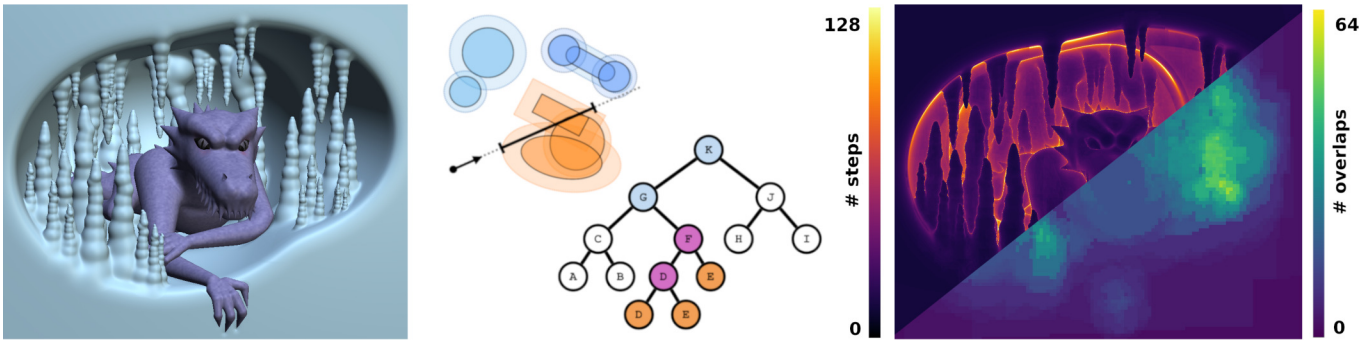


Fig. 1. We propose a new implicit rendering pipeline that is well suited for modeling. Thanks to a sparse expression tree traversal and synchronized ray processing, the pipeline scales well to trees consisting of thousands of primitives. *Left:* Dragon model consisting of 860 primitives combined with smooth unions and differences. *Middle:* When processing a ray subrange, we want the field evaluation complexity to be proportional to the local number of primitive overlaps and not the full blobtree size. *Right:* Number of field evaluations per ray (top) and maximal number of primitives overlap processed per screen tile (bottom).

Implicit volumes are known for their ability to represent smooth shapes of arbitrary topology thanks to hierarchical combinations of primitives using a structure called a blobtree. We present a new tile-based rendering pipeline well suited for modeling scenarios, i.e., no preprocessing is required when primitive parameters are updated. When using approximate signed distance fields (fields with Lipschitz bound close to 1), we rely on compact, smooth CSG operators - extended from standard bounded operators - to compute a tight augmented bounding volume for all primitives of the blobtree. The pipeline relies on a low-resolution A-buffer storing the primitives of interest of a given screen tile. The A-buffer is then used during ray processing to synchronize threads within a subfrustum. This allows coherent field evaluation within workgroups. We use a sparse bottom-up tree traversal to prune the blobtree on-the-fly which allows us to decorrelate field evaluation complexity from the full blobtree size. The ray processing itself is done using the sphere tracing algorithm. The pipeline scales well to volumes consisting of thousands of primitives.

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

Additional Key Words and Phrases: implicit surfaces, signed distance field, blobtree, ray-tracing

ACM Reference Format:
Zanni Cédric. 2018. Synchronized tracing of primitive-based implicit volumes. 1, 1 (October 2018), 15 pages. https://doi.org/XXXXXXX.XXXXXXX

Author's address: Zanni Cédric, cedric.zanni@loria.fr, Université de Lorraine, CNRS, Inria, LORIA, Nancy, France.

XXXX-XXXX/2018/10-ART $15.00
https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

Implicit volumes are functional representations that allow modeling without topological concerns. They are defined as the set of points $\mathbf{p} \in \mathbb{R}^3$ satisfying $f(\mathbf{p}) \leq c$ (or $\geq c$), where f is a continuous scalar function, usually smooth by part, and c is a fixed isovalue. A broad range of representations is built from functions representing primitives, like spheres and cubes, combined hierarchically using operators like sharp union and smooth blending. This forms a construction tree known as a Blobtree [Wyvill et al. 1997, 1999], a compact shape representation suitable for non-destructive workflows. Blobtree encompass key conventions: density fields [Wyvill et al. 1986] and functional representations [Pasko et al. 1995], a superset of signed distance fields (SDF). SDFs find applications in various domains: CAD packages [Courter 2019; Keeter 2019a], demoscene [Burger et al. 2002; Quilez and Jeremias 2013], modeling applications [mag 2021] and games [Molecule 2020; Order 2018]. Implicit surfaces are expensive to visualize, so they are typically meshed or voxelized in modeling frameworks. To enable interactive feedback, a low resolution is often required, resulting in lost details. Ray-tracing algorithms, such as sphere tracing [Hart 1996], alleviate these precision issues. However, when used on GPU, the field function is often encoded within the ray-tracing shader, causing significant increases in rendering and shader compilation times for large expressions, which hinders interactive modeling.

Recently, many data-oriented visualization methods have emerged to avoid shader recompilation during shape updates. Most methods target a single primitive type combined with an n-ary summation blend [Aydinlilar and Zanni 2021; Bruckner 2019]. The summation commutativity enables *sparse field evaluation* from a subset of primitives, decoupling evaluation complexity from expression size – a property sought for blobtrees (see Figure 1, center). Notable exceptions by Grasberger et al. [2016] and Keeter [2020] efficiently handle more general implicit volumes on the GPU. They represent the field expression as data (a tree or operation tape) interpreted on the

GPU, avoiding shader recompilation during shape updates. However, both methods use either progressive expression simplification (on the fly during rendering) or static acceleration structures (as preprocessing); requiring costly processing of the entire tree to limit subsequent evaluations complexity, hindering usage for modeling.

Our contributions. We propose a new implicit rendering pipeline influenced by aforementioned data-oriented approaches but requiring limited preprocessing. Drawing from compactly supported density fields, we design *compact*, smooth constructive solid geometry (CSG) operators for approximate signed distance field (A-SDF). These operators enable the computation of tight augmented bounding volumes for primitives. By utilizing these volumes, we *decorrelate field complexity from the full expression size* (see Figure 1) by on-the-fly tree pruning via a sparse bottom-up tree traversal, accessing only a subset of blobtree nodes. Building on these contributions, we present a GPU raytracing pipeline that employs a tile-based view-dependent acceleration structure, storing primitives of interest along subfrustums. This structure facilitates workgroup synchronization, ensuring coherent sparse field evaluation within workgroup threads. We demonstrate the efficiency and versatility of our approach using sphere tracing on A-SDF examples including large blobtrees, procedural nodes, and voxelized SDFs. The pipeline scales well to trees with thousands of primitives. While we focuse on A-SDFs, the overall pipeline is useable for density fields.

2 RELATED WORK

2.1 Implicit Modeling

Implicit surface modeling is a paradigm taking root in CSG with the functional definition of smooth and sharp union and intersection [Ricci 1973]. An in-depth introduction is available in [Bloomenthal and Wyvill 1997]. Field definitions follow two main conventions. On the one hand, *density fields* ($c > 0$) are similar to a material density falling off to zero when going away from the shape. They naturally encompass a radii of influence to localize primitive interaction in space (i.e., compactly supported fields). They provides a straightforward blending operator: the summation [Wyvill et al. 1986]. On the other hand, functional representation, or F-Rep [Pasko et al. 1995], has fields that increase toward infinity when going away from the 0 isosurface defining the shape. A notable subset are SDFs and approximate ones (A-SDF), with fields defined as the (approximate) signed distance to the shape. SDFs verify the Eikonal equation:

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

everywhere in space but at field singularities. Thus, they have a Lipschitz bound - an upper bound on the field gradient norm - of 1, simplifying field processing. In contrast, A-SDFs only provide a Lipschitz bound L_f close to 1 :

$$\|\nabla f(\mathbf{p})\| \leq L_f \tag{2}$$

A-SDFs can be converted to *conservative* SDFs [Takikawa et al. 2022] with $L_f = 1$ by multiplying f by $1/L_f$. Consequently, A-SDFs typically require more field evaluations than exact SDFs. For both conventions, complex shapes can be constructed by combining primitives through a hierarchical blend tree known as a blobtree [Wyvill et al. 1999]. For instance, unions and intersections can be defined by minimum and maximum functions.

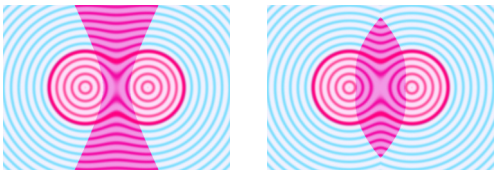


Fig. 2. *Left:* Standard bounded smooth union modify the field in an infinite volume (i.e., they are only bounded at the 0 isovalue level); this is visible in the primitives interaction area displayed in purple. This either increases field evaluation complexity or requires additional preprocessing. *Right:* Inspired by density fields, we propose to localize fully blending regions in space.

An wide range of operators has been developed to create smooth, controllable blends between shapes. When operands are exact SDFs, the resulting field is only a *conservative* SDF. Many F-Rep blending operators feature the broadly adopted bounded blending property [Dekkers et al. 2004; Pasko et al. 2002, 2005], which confines blends to a bounded volume around the intersection of the two input surfaces. However, the interaction between the two input fields is not spatially localized, requiring both field operands to compute the blended field far from the unblended shapes (see Figure 2). This issue arises either from using clean CSG R-function operators [Pasko et al. 2002, 2005] to produce smooth fields everywhere in space or from employing unbounded displacement functions to maintain the Lipschitz bound [Dekkers et al. 2004]. To efficiently process large blobtrees, we propose strengthening the bounded blending property to guarantee a bounded interaction volume, akin to compactly supported density fields and partition of unity blends [Tobor et al. 2004]. While the latter is a *compact* region-based operator, its usage can be challenging for primitive-based modeling, as transitions between regions must be defined based on the primitives in each region to avoid surface artifacts and a significant increase in Lipschitz bounds. Our new operators are based on polynomial bounded operators [Dekkers et al. 2004] that are well-suited for efficient GPU evaluation, as seen in games [Molecule 2020] and the demoscene [Quilez 2008].

2.2 Implicit volume visualization

Direct visualization of implicit shapes requires solving ray isosurface intersection for each pixel of an image:

$$f \circ \mathbf{r}(t) - c = 0 \tag{3}$$

where $\mathbf{r}(t)$ parameterizes the ray for a given pixel. Efficient implementation requires addressing three aspects: the root-finding problem, the representation of the field function, and acceleration strategies for expensive field evaluations.

Ray processing. The simplest yet costly root-finding strategy is to march along rays with a constant step size until a sign change is detected; resulting in a tradeoff between rendering time and artifacts (missed details). To limit the field evaluations, a family of methods relies on polynomial representation [Gourmel et al. 2010; Nishita and Nakamae 1994] or approximation [Aydinlilar and Zanni 2021; Sherstyuk 1999], limiting applicability to specific implicit shapes.

A more general method category employs ray bisection driven by inclusion function bounding f on a ray interval. Inclusion functions are build using either interval arithmeric [Duff 1992; Keeter 2020; Knoll et al. 2009; Mitchell 1990] or variant of affine arithmetic [Fryazinov et al. 2010; Knoll et al. 2009; Sharp and Jacobson 2022]. The recursion stack can be a limiting factor for GPU implementation. A notable exception is the method of Keeter [2020], which is more akin to screen space voxelization.

The last method category relies on the Lipschitz bound of f to compute an intersection-free step size to advance iteratively along rays. The seminal sphere tracing algorithm [Hart 1996] uses either a global bound or a localized one relying on an octree. While overrelaxation can be used to reduce the number of steps [Bálint and Valasek 2018; Keinert et al. 2014], grazing rays still require many steps. Segment tracing [Galin et al. 2020] addresses this issue by computing directional Lipschitz bounds on ray subintervals; this can also be applied to A-SDF. In this paper, we rely on sphere tracing for its simplicity, but we design the pipeline architecture for iterative processing methods in general.

Field expression representation. When implementing a ray processing routine on GPU, the encoding of the field expression has important implications. If represented as function calls hard coded in a shader [Quilez and Jeremias 2013; Reiner et al. 2011], good performance is achievable for reasonably sized expressions, but this requires shader recompilation whenever the expression is updated. Alternatively, the expression can be represented as data in GPU memory. For a single primitive type and commutative operator [Aydinlilar and Zanni 2021; Bruckner 2019; Gourmel et al. 2010], the entire expression can be stored as an array of primitive parameters. For more general implicit shapes, such as blobtrees, an interpreter can dispatch function calls based on a type identifier [Grasberger et al. 2016; Keeter 2020]. Both methods rely on linearized data structures to avoid non-linear memory access and enable stackless expression traversal: post-order tree storage for blobtrees [Grasberger et al. 2016] and tape for arithmetic expressions represented as directed acyclic graph [Keeter 2020]. Our data structure derives from [Grasberger et al. 2016] to prevent shader recompilations during modeling. Compared to mathematical expression interpreter [Keeter 2020], node-level interpreters [Grasberger et al. 2016] decrease the switch execution counts and require less data, while supporting fully procedural nodes. The main downside of node-level interpreters is shader complexity and recompilation whenever the family of nodes is changed.

Acceleration strategies. For complex expressions, direct field evaluation can become prohibitively expensive, but several strategies can mitigate the issue. First, field values can be cached for re-use in subsequent processing [Reiner et al. 2012; Schmidt et al. 2005]; this is of limited use when shapes undergo significant modifications.

Most strategies target compactly supported density fields and rely on tree pruning [Fox et al. 2001] combined with spatial acceleration structures. A *pruned tree* is a subtree providing an equivalent expression for a given volume; it is selected using the acceleration structure at field query time. Bounding Volume Hierarchy (BVH) [Gourmel et al. 2010], BSP-tree/kD-tree [Grasberger et al. 2016], and octree [Dyllong and Grimm 2007] have been used. Such techniques require full tree preprocessing when the primitive parameters change. BVHs can also be used to stop top-down tree traversal on the fly [Grasberger et al. 2016]. This restricts the shape of the BVH; the latter and blobtree have opposed optimal shapes (balanced versus left-heavy).

An alternative approach by Keeter [2020] relies on on-the-fly expression simplification. It uses a multigrid space subdivision to simplify the expression progressively in subcells using interval arithmetics. This requires processing the whole expression in the first step of the rendering pipeline. A key feature of this approach is coherent processing, particularly well suited for GPU architecture, a property shared by our pipeline.

When using a single commutative operator, an A-buffer – per-pixel linked list of fragments – can be used to store primitives of interest per rays [Aydinlilar and Zanni 2021; Bruckner 2019]. We adapt this solution to more general implicit volumes represented by blobtrees, taking special care about ray processing coherency.

Secondary rays. Ray isosurface intersection can be used to compute light visibility and global illumination by generating new rays at isosurface crossings. However, this introduces challenges, such as preventing self-intersections [Keinert et al. 2014]. One common approach, used in the demoscene [Quilez and Jeremias 2013], relies on hand-optimized field evaluations hardcoded within a fullscreen shader [Reiner et al. 2011]. This method is limited to fully procedural scenes and lack strategies for field evaluation coherence. The secondary rays tend to be less coherent than primary rays from the initial viewpoint, making coherent field queries even more challenging when implemented on GPUs. The main solution for GPU evaluation coherence is to use discrete SDFs precomputed in 3D textures [Liktor 2008]; it suffers from required precomputation and loss of details due to resolution limits, particularly with reflective surfaces. Similarly, neural implicit volumes offer a more coherent evaluation of the field function [Sharp and Jacobson 2022]. Finally, using a single primitive type with a commutative operator allows for BVHs to leverage ray-tracing hardware acceleration [Gourmel et al. 2010]. However, this restricts the types of surfaces that can be visualized. Currently, no methods exist for supporting secondary rays on GPUs with fully dynamic blobtrees without field discretization.

Our approach focuses on primary ray coherence to enhance rendering for modeling tasks. Extending it to secondary rays is beyond this paper's scope; potential benefits are discussed in Section 8.3.

3 METHOD OVERVIEW

We design our rendering pipeline for coherent and sparse blobtree evaluation, aiming to decorrelate the complexity of evaluation from the blobtree size and limit the processing of empty space.

Terminology. Our pipeline relies on the notion of local equivalence between A-SDFs to reduce field evaluation complexity. Given f, a reference A-SDF defined in $\mathbb{R}^3$, we define the local equivalence of an A-SDF f_a with f in the volume $\mathcal{V}$ as:

$$f_a \sim_{\mathcal{V}} f \Leftrightarrow \forall \mathbf{p} \in \mathcal{V}\ ,\ \mathrm{sign}(f_a(\mathbf{p})) = \mathrm{sign}(f(\mathbf{p})) \tag{4}$$

This implies that f and f_a define a common 0 isovolume in $\mathcal{V}$; solving $f_a \circ \mathbf{r}(t) = 0$ within $\mathcal{V}$ provides the same root as solving $f \circ \mathbf{r}(t) = 0$. Examples of equivalent fields are depicted in Figure 4.

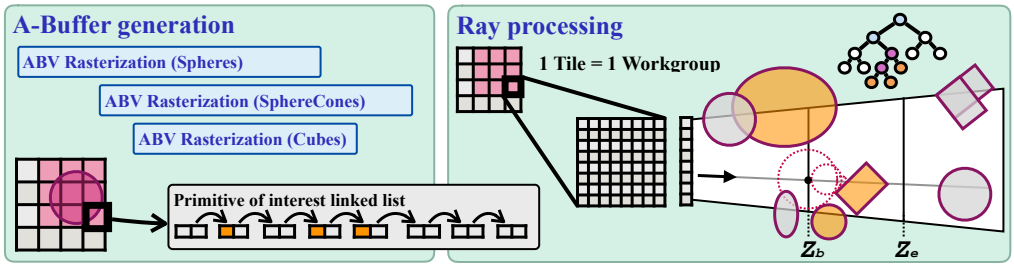


Fig. 3. Our tile-based rendering pipeline consists of two steps. *Left:* First, a rasterization step is used to populate linked-lists of a low resolution A-buffer with primitive ABV (one list per screen tile). *Right:* Then, a compute shader is used to process tiles (1 workgroup per tile); the linked-list is used to segment the tile's frustum. Threads of a workgroup therefore access to the blobtree in a coherent fashion. Using the active primitives of a subfrustum - colored in orange - we introduce a sparse blobtree evaluation to limit field evaluation complexity (see Figure 1). Ray subintervals are processed using the sphere tracing algorithm.

Given a volume $\mathcal{V}$, we define its *active primitives* as the minimal set of primitives $\mathcal{A}$ required to define an equivalent field $f_{\mathcal{A}}$ using a pruned version of the original blobtree. Although $f_{\mathcal{A}} \neq f$ in $\mathcal{V}$, it remains an A-SDF by construction and, therefore, well suited for sphere tracing in $\mathcal{V}$. For each primitive, we define its *augmented bounding volume* (ABV) as the primitive bounding volume augmented by the blending area of its ancestor operators (see Figure 4d); if the query volume $\mathcal{V}$ intersects the primitive ABV, the primitive is considered active. We propose a simple modification of standard bounded operators to obtain tight ABVs when used with infinite-support A-SDF primitives.

GPU pipeline. We use a tile-based pipeline to ensure coherent processing within GPU workgroups. Rays are processed in buckets corresponding to screen tiles T_{uv} and frustums $\mathcal{F}_{T_{uv}}$. Our pipeline consists of two steps: first, we create a *tile-based view-dependent acceleration structure* - an A-Buffer - that stores a depth-sorted list of primitives per tile (see Figure 3 - left). Second, we perform ray processing to locale the isosurface crossing (see Figure 3 - right). The primitives associated to a tile T_{uv} in the A-buffer correspond to those whose ABV intersects $\mathcal{F}_{T_{uv}}$. We populate the A-buffer through low-resolution rasterization of the primitive ABVs.

Ray processing is performed in a compute shader, with each workgroup corresponding to a tile T_{uv}. This allows us to *synchronize ray traversal* for coherent blobtree access. We iterate over the primitives in the A-Buffer to segment $\mathcal{F}_{T_{uv}}$ into depth ranges where a fixed set of *active primitives* $\mathcal{A}$ defines an equivalent field $f_{\mathcal{A}}$. The latter is defined using a sparse bottom-up blobtree traversal, only traversing branches terminated by active primitives (see Figure 1 *middle* and 7). This traversal can be used for direct field evaluation or on-the-fly building of pruned blobtrees in GPU shared memory (as done in our complete pipeline). Finally, sphere tracing is applied in each subinterval until an iso-crossing is detected.

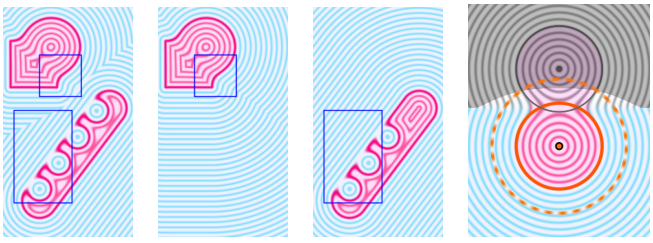


Fig. 4. *a:* An A-SDF generated from 7 primitives. *b,c:* Equivalent fields for two query volumes (blue boxes) using respectively 2 and 4 primitives; the 0 isosurface at the exterior of the boxes is not preserved. *d:* a primitive ABV (orange dashed circle) is defined by augmenting the primitive bounding volume (orange circle) by the operator blending area.

We first discuss in Section 4 the operator property that allow for trivial localization of primitive influence in space. This section presents simple extension of existing smooth CSG operators that verify this property. Next, we describe the sparse bottom-up tree traversal in Section 5. In Section 6, we present all aspects related to synchronized processing, including the construction of the A-buffer. For clarity, Sections 5 and 6 focus solely on processing union-like operators. Minor changes required to support intersection and difference-like operators are presented in the Supplemental Material (Section A.2). Finally, important implementation details are discussed in Section 7, and results are presented in Section 8.

4 TAXONOMY OF IMPLICIT BLENDS

To minimize preprocessing during modeling, we aim to compute the ABVs 1) in a single tree traversal for efficiency, and 2) independently from the spatial embedding of primitives, ensuring no preprocessing is needed when only the primitives are modified. Assuming well-behaved A-SDF primitives (Lipschitz bound ≤ 1), the ABV of a primitive p can be constructed by offsetting a bounding volume of its 0 isosurface by the largest field value $r_p^+ \geq 0$ at which p can influence the 0 isosurface of f (see Figure 4d).

Range of interaction. The value of r_p^+ depends on the property of the operators above p in the blobtree and can be computed in a single tree traversal by propagating a range of field values from the blobtree root. Each blobtree node n defines a real-valued function $f_n : \mathbb{R}^k \rightarrow \mathbb{R}$ where $k = 3$ for primitives (space coordinates) and $k = 2$ for binary operators (two children field operands). Let us call the node *range of interaction* (or ROI):

$$R_n = [r_n^-, r_n^+] \text{ , where } r_n^- \leq 0 \text{ and } r_n^+ \geq 0 \tag{5}$$

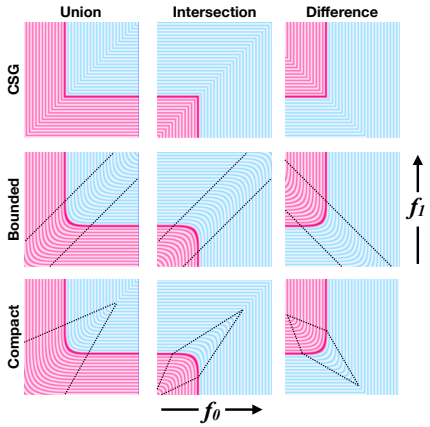


Fig. 5. Blend graphs: The operator's operands f_0 and f_1 correspond to the x and y axes. The interior of the resulting shape is shown in red, the exterior in blue. *Top:* CSG operators acts as operand selectors, returing either one of the operands up to a sign change. *Middle:* Bounded smooth operators do not restrict operand interactions in space: evaluating the operator can require both operands at an arbitrary distance of the operand 0 isosurface, *Bottom:* We propose modified operators to localize field interactions in space. Outside of the interaction area, operators act as field selectors.

the subrange of the codomain of f_n where an equivalent field $\tilde{f}_n$ must be equal to f_n to ensure $\tilde{f}_{root} \sim_{\mathcal{V}} f$ for any volume $\mathcal{V}$ (independence to space embedding). Outside this range, $\tilde{f}_n$ must be less than r_n^- if negative and superior to r_n^+ if positive. The lower bound is necessary since difference-like operators can invert signs.

By definition of the ROI and Equation (4), the ROI of the root node is defined by $r_n^- = r_n^+ = 0$. Children's ROIs can then be computed solely based on their parent node's ROI and mathematical property.

CSG operators. The union of two A-SDF field f_0, f_1 is defined as:

$$\mathrm{csg}_\cup(f_0, f_1) = \min(f_0, f_1)\,, \tag{6}$$

the intersection (resp. difference) as:

$$\mathrm{csg}_\cap(f_0, f_1) = \max(f_0, f_1), \text{ and } \mathrm{csg}_\setminus(f_0, f_1) = \max(f_0, -f_1)\,. \tag{7}$$

Blend graphs of all discussed operators are shown in Figure 5. These operators act as operand *selectors* - up to a sign inversion for difference-like operators - effectively preventing operand interactions. In the absence of one of the operands, an equivalent field can always be deduced by treating the missing operand as having an infinite value. When using only CSG operators, all ROI satisfy $r_n^- = r_n^+ = 0$.

Bounded operators. A polynomial C^2 smooth union [Dekkers et al. 2004; Molecule 2020; Quilez 2008] can be defined from CSG as:

$$g_\cup(f_0, f_1, \beta) = \mathrm{csg}_\cup(f_0, f_1) - d(f_0, f_1, \beta) \tag{8}$$

with

$$d(f_0, f_1, \beta) = \frac{\beta}{6} \max\left(1 - \frac{|f_0 - f_1|}{\beta}, 0\right)^3 \tag{9}$$

where β is a parameter defining the blend range (see Figure 5 - middle row). The smooth intersection is similarly defined as:

$$g_\cap(f_0, f_1, \beta) = \mathrm{csg}_\cap(f_0, f_1) + d(f_0, f_1, \beta) \tag{10}$$

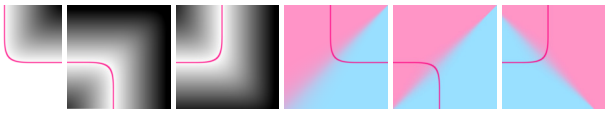


Fig. 6. *Left:* Transition function b_r in R_o^2 for smooth union, intersection, and difference. The blends are properly closed on the edges of the domain due to the base operators' bounded property. *Right:* Color mixing ratio between operands emphasizes the localized influence of each operand in space.

and the smooth difference as:

$$g_\setminus(f_0, f_1, \beta) = \mathrm{csg}_\setminus(f_0, f_1) + d(f_0, -f_1, \beta) \tag{11}$$

When studying ROIs of a node n using $g_\cup$, one can first consider the case where the two operands are equal. In this scenario, we have $f_n = f_c - \beta/6$ with f_c the children field; effectively increasing r_c^+ by at least $\beta/6$ compared to r_n^+. Since this occurs independently of the value of r_n^+, the area of interaction between the primitives extends to infinity (see Figure 2), primitive ROIs grow infinitely depending on subsequent blend operations.

Compact operators. We propose a *compactness* property for operators: an operator should have a range of interest R_o defined solely by its parameters, defined as the largest, possibly lowest, field value at which its operands interact. Outside of this range, the operator should act as an operand selector, effectively preventing primitive interactions outside of R_o, which in turn allows deducing an equivalent field evaluation from active operands only (see Figure 5 - bottom row). As previously discussed, CSG operators satisfy this property. We therefore define our operators as follows:

$$G_{\mathrm{op}}(f_0, f_1, \cdots) = \begin{cases} \bar{g}_{\mathrm{op}}(f_0, f_1, \cdots) & \text{if } f_0 \in R_o \text{ and } f_1 \in R_o, \\ \mathrm{csg}_{\mathrm{op}}(f_0, f_1) & \text{otherwise.} \end{cases} \tag{12}$$

with $\mathrm{op} \in \{\cup, \cap, \setminus\}$, falling back on CSG operators if any operand is outside of R_o. The operator $\bar{g}_{\mathrm{op}}$ is defined on R_o^2 satisfying continuity constraints with $\mathrm{csg}_{\mathrm{op}}$ at the domain boundary, thus ensuring the overall operator is continous on $\mathbb{R}^2$.

With this definition, ROIs growth are effectively bounded by the operator range. Given a blobtree node n, its ROI R_n and the associated operator range R_o, the ROIs of its two children c_0 and c_1 can be defined as:

$$R_{c_i} = [r_n^-, \max(r_n^+, r_o^+)] \tag{13}$$

for union-like operator. For intersection-like operators, we have:

$$R_{c_i} = [\min(r_n^-, r_o^-), r_n^+] \tag{14}$$

For difference-like operators, the rule differ for each children; the left child ROI is defined by:

$$R_{c_0} = [\min(r_n^-, r_o^-), r_n^+] \tag{15}$$

and the right one by:

$$R_{c_1} = [-r_n^+, \max(-r_n^-, r_o^+)] \tag{16}$$

where the interior and exterior limit of the ROI are inverted. The validity of those rules is demonstrated in Supplemental Material.

Adapting existing operators. We only need to design operators $\bar{g}_{op}$ within the operator range $[r_o^-, r_o^+]^2$ such that it continuously transition to CSG operators at the domain limit. We design our new operators based on $g_{\{\cup,\cap,\setminus\}}$ by progressively closing the blend range ($\beta \to 0$) when moving away from the 0 isosurface resulting from the unmodified operator. This can be done by evaluating the unmodified operator twice, e.g., for the smooth union:

$$\bar{g}_\cup(f_0, f_1, \beta, r_o) = g_\cup(f_0, f_1, b_r(g_\cup(f_0, f_1, \beta), \beta, r_o)) \tag{17}$$

Here, the first evaluation is used within a transition function b_r to interpolate the blend parameter from β on the 0 isosurface of $g_\cup(f_0, f_1, \beta)$ to 0 in (r_o, r_o) – see Figure 6 This preserves the same blending behavior when only two primitives are combined. The resulting operator is depicted in Figure 2 and 5.

In practice, we parametrize the blend range by a single value $r_o = r_o^+$. The lower bound is $r_o^- = -r_o$ for difference and intersection-like operators and $r_o^- = -\infty$ for union-like operators; the rationale for the infinite bound is described in Supplemental Material. We define the transition function as follows:

$$b_r(x, \beta, d) = \beta \ \max\left(1 - \frac{6x}{6d - \beta}, 0\right) \tag{18}$$

Note that a *not a number* can appear during the evaluation of b_r; it can be filtered by returning 0 instead. The value of r_o can either be set automatically based on β or by hand, which can help prevent significant field compression when a large number of primitive overlaps are expected (see discussion in Section 8). The smooth intersection and difference differ only in that the first operand of b_r uses an absolute value.

5 SPARSE BOTTOM-UP TREE TRAVERSAL

Our blobtree representation builds on the data-oriented approach of [Grasberger et al. 2016], utilizing post-order tree storage for tree traversals and node-level interpreters for field evaluations. This storage layout removes the need for a traversal stack during complete bottom-up tree traversals, as the nodes are processed in storage order. We instead leverage this ordering to define our sparse tree traversal and provide an alternative way to limit the stack memory requirements (see Section 6.2).

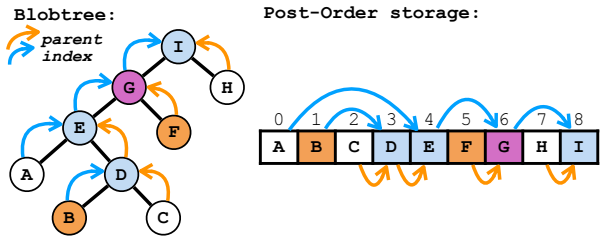


Fig. 7. *Left:* A sparse blobtree traversal should only access colored nodes starting from active primitives (in orange). Nodes in blue should only act as operand selectors, while nodes in purple should be evaluated as usual. This color convention is reused in all figures. *Right:* Similarly to [Grasberger et al. 2016], we store on GPU a linearized version of the blobtree where nodes appear in *post-order* tree traversal order. For each node, we store an ancestor index (initially the index of the parent node for the sake of explanation).

The Blobtree. Our input is a linearized blobtree $\mathcal{B} := \{B_i\}$ where each node B_i is identified by its post-order index $i \in 1..|\mathcal{B}|$ (see Figure 7) with $|.|$ denoting the cardinality of a set. The root of the tree is therefore $B_{|\mathcal{B}|}$. Each node B_i is associated with a set of parameters D_i for primitives or operator. Space deformations are moved to leaf nodes (i.e., primitives) and stored in D_i; our implementation only supports rotation and translation. Given a node B_i, we can query the following information:

- isLeaf(B_i) : whether the node is a leaf node (i.e. a primitive)
- isLeft(B_i) : whether the node is a left child
- type(B_i) : *nodeop* identifier representing the type of the node
- ancestor(B_i) : ancestor index, ancestor informations can, in turn be accessed with $B_{ancestor(B_i)}$

The ancestor index refers to the parent node or another ancestor further up the tree, facilitating sparser traversal (as discussed in Section 5.2). The actual blobtree memory layout used on GPU is derived directly from these notations and is described in Section 7.

As in [Grasberger et al. 2016], we use two distinct *node-level interpreters* for primitive and operator field evaluations. For a node B_i, the interpreter selects the appropriate function based on $type(B_i)$; the chosen function then uses the node data D_i (see Algorithm 1).

Active blobtree nodes. Thanks to the compactness property of our operators, we only need to process a subset of the blobtree nodes to evaluate a local field $f_{\mathcal{A}}$. This set of nodes – called *active nodes* – is defined as the union of $\mathcal{A}$ and the nodes that contains at least one active primitive in their child subtrees (see colored nodes in Figure 7). We rely on the post-order indices $\{p_i\}_{i\in 1..|\mathcal{A}|}$ of active primitives – provided by the A-buffer – to propose a sparse bottom-up tree traversal algorithm that iterates *only* over the *active nodes*. An active operator is evaluated only if all its operands are active; otherwise, it functions as a field *selector*. This traversal serves as the foundation for direct field evaluation and on-the-fly tree pruning.

We focuse on union-like operators, where operand selection consists in active operand propagation if only one is present. Operand selection for other operators is discussed in Supplemental Material.

Algorithm 1 Node interpreters

```
procedure PRIMITIVE(nodeop, D, p)
    switch nodeop do
        case 0 : return sphereSDF(p, D)
        case 1 : return cubeSDF(p, D)
        ...
end procedure
procedure OPERATOR(nodeop, D, f0, f1)
    switch nodeop do
        case 0 : return csg∪(f0, f1)
        case 1 : return G∪(f0, f1, D)
        ...
end procedure
```

5.1 Parent index sparse traversal

Thanks to the post-order linearized layout of $\mathcal{B}$, we can traverse the tree bottom-up in post-order by simply iterating over nodes in storage order. This ensures that each node is processed after its two child subtrees, when subtree field values are available. We must preserve this property to access the memory of active nodes only once. Besides, iteratively moving up from each active primitive using parent indices allows to traverse *all and only active nodes*, but lead to multiple traversals of those nodes (see Figure 8 - left). Therefore, we should interrupt parent traversals before reaching nodes whose operands are not ready. To process primitives in the correct order, they must be stored by increasing post-order indices; we assume that $\{p_i\}$ are sorted accordingly.

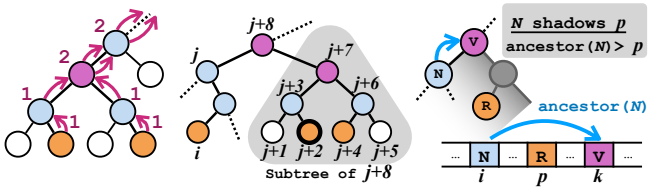


Fig. 8. *Left:* Traversing the tree from each active primitive to the root using parent indices leads to multiple traversals of some operators. *Middle:* The first node to process during sparse traversal of a subtree is always an active primitive. *Right:* A node should not be fetched and processed if it is stored after the first unprocessed active primitive (shadowing condition).

Algorithm 2 Sparse blobtree traversal.
DATA: floating-point for direct field evaluation, void for tree pruning

```
1:  procedure TRAVERSAL(B, A)
2:      S ← {}                                  ▷ Stack of node and processing DATA
3:      for i ∈ 1..|A| do
4:          b ← B_{p_i}
5:          f1 ← PRIMITIVE(b, D_{p_i}, p)
6:          #1 : Clamp ancestor of b                          ▷ Fast traversal
7:          while (i = |A| in ∨ ! shadow(b, p_{i+1}))
8:                ∧(i ≠ |A| ∨ S ≠ ∅ ∨ isValid(ancestor(b))) do
9:              a ← ancestor(b)
10:             b ← B_a
11:             if S ≠ ∅ then
12:                 b_top ← S.top().node
13:                 if popRequired(a, b_top, b) then
14:                     f0 ← S.top().field
15:                     f1 ← OPERATOR(b, D_a, f0, f1)
16:                     S.pop()
17:                 end if
18:             end if
19:             #2 : Clamp ancestor of b                      ▷ Fast traversal
20:         end while
21:         S.push({b, f1})          ▷ shadowing or root is processed
22:     end for
23:     return S.top().field
24: end procedure
```

Shadowing condition. As shown in Figure 8 (middle), the first node to process in a subtree is always a primitive. We cannot process the subtree's parent until its first primitive is. Thus, we should not use a parent index to access a new node if its index is exceeds the next unprocessed active primitive index p. Given p and a node b, we define the *shadowing condition* as:

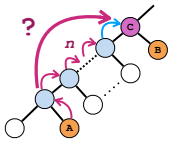
$$\text{shadow}(b, p) \rightarrow \text{ancestor}(b) > p\,, \tag{19}$$

as shown in Figure 8 (right). Following these principles, we present a first version of our tree traversal in Algorithm 2. Figure 9 highlights the main steps of the algorithm. First, the intermediate state – the shadowing node and field value – is stacked once per primitive (line 21 in the algorithm) when the shadowing condition is met (line 7) or the tree root is reached (line 8). Secondly, due to our operator *compactness*, a node b of index a only needs to be evaluated if both its operands are actives. In this case, the right operand corresponds to the currently processed tree branch (field f_1), and the left operand is on top of the stack. This condition can be written as:

$$\text{popRequiredA}(a, b_{top}, \cdots) \rightarrow a = \text{ancestor}(b_{top}) \tag{20}$$

with b_{top} the node at the top of the stack. As we push intermediate nodes on the stack, Algorithm 2 increases the stack memory required for field evaluations: the stack must store both node information and field values. Our complete pipeline avoids this issue (see Section 6.2).

Left heavy tree. Left-heavy trees should be favored to limit field evaluation stack size [Grasberger et al. 2016]; a comb-like tree, as shown in Figure inset, requires a stack size of only 1. However, traversing such trees with Algorithm 2 requires processing all operators stored after the first active primitive. On average, this corresponds to $|\mathcal{B}|/4$ nodes, which can be considerable for large trees. We propose to solve this issue by replacing parent indices with another type of ancestor index.



5.2 Fast ancestor index

We replace parent indices with indices providing longer jumps combined with adapted rules to detect active operators. Given a node index p and its parent index a, we define its fast ancestor index as :

$$\text{fastAncestor}(p, a) \rightarrow \begin{cases} a, \text{if } \text{isLeft}(B_p) \neq \text{isLeft}(B_a)) \\ \text{fastAncestor}(a, \text{ancestor}(B_a)), \text{otherwise} \end{cases}$$

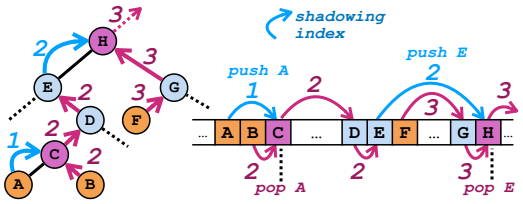


Fig. 9. Starting from active primitives, sparse traversal follows active branches until the shadow condition is met; this triggers the stacking of B_i for later use. Unstacking happens when the current node corresponds to the ancestor stored at the top of the stack.

This allows longer jumps provided the jump direction – left or right – does not change (see Figure 10). These indices only depend on the tree structure and are precomputed. As fast indices cannot decrease the node's ancestor index, the shadowing condition remains unchanged. However, the method for detecting operators to process requires modification to avoid missing operators (see Figure 10). We address this by clamping the fast index of current node b to the ancestor of the node at the top of the stack – ancestor(S.top()) if it exists – at marker #1 and #2 of the algorithm.

After clamping, only two configurations require the processing an operator (see Figure 10 bottom right). The associated condition is the disjunction of *popRequiredA* with:

$$\begin{aligned}\text{popRequiredB}(a, b_{top}, b) \rightarrow \text{ancestor}(b) \geq \text{ancestor}(b_{top}) \\ \wedge (!\,\text{isValid}(\text{ancestor}(b)) \vee \text{isLeft}(b))\end{aligned}$$

This corresponds to the bottom right configuration in Figure 10: the node is a left child (second part of the conjunction) and has an ancestor greater or equal to the one at the stack top (first part).

6 SYNCHRONIZED TRACING

GPU architectures require coherent processing within workgroups to maximize throughput. In our context, this means that all threads of a workgroup should use the same field $f_{\mathcal{A}}$ so that interpreters call the same functions. To achieve this objective, we process rays by buckets corresponding to screen tiles T_{uv} of size 8×8 pixels and associated frustum $\mathcal{F}_{T_{uv}}$. During processing, we segment $\mathcal{F}_{T_{uv}}$ by depth to define volumes $\mathcal{V} = \mathcal{F}_{T_{uv}}[z_b, z_e]$ where a limited set of active primitives $\mathcal{A}$ contributes to the definition of $f_{\mathcal{A}}$. We utilize a tile-based acceleration structure built per frame (first pipeline step) to track active primitives and synchronize workgroup threads during ray processing (second pipeline step, see Figure 3).

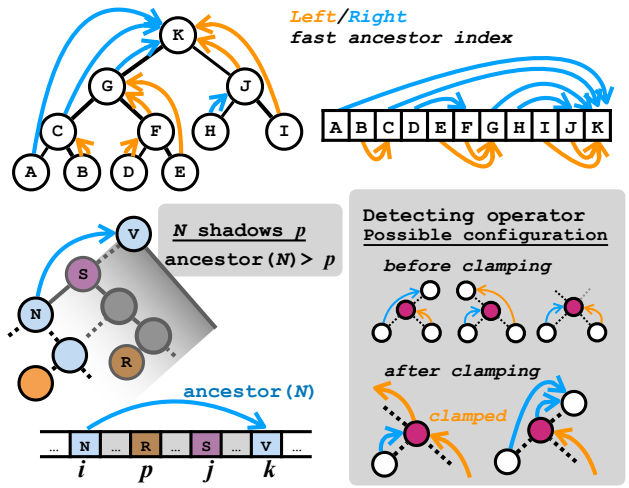


Fig. 10. *Up:* Fast left and right ancestor indices in a blobtree. *Down:* Shadowing condition remains valid when using fast ancestor indices (left). Possible configurations to detect nodes with two operands before and after clamping of ancestor index with the ancestor of the stack top node (right).

6.1 Acceleration structure

To efficiently identify active primitives and their indices in $\mathcal{B}$ for a volume $\mathcal{F}_{T_{uv}}[z_b, z_e]$, we use an A-buffer built from primitive ABVs, as described in [Aydinlilar and Zanni 2021; Bruckner 2019], focusing on density fields. The *A-buffer* is a per-pixel linked list of fragments; our implementation maps entries to screen tiles instead of pixels, reducing memory usage and minimizing atomic operations during linked-list creation. Each fragment is linked to its primitive index in $\mathcal{B}$ and its ABV entry/exit depth in normalized device coordinates (NDC). We build the A-buffer by rasterizing ABVs using the parallel sort described in [Lefebvre et al. 2014]; with entry depth as the sorting key. To reduce overlaps, we keep ABVs tight by using several volume shapes, including spheres, sphere-cones, and boxes; the system can be easily extended to support additional shapes.

ABV rasterization. We use the same approach for all ABV types, using a rendering pass at downsampled resolution where pixels maps to tiles. A geometry shader generates screen-space quad imposters whose rasterization instantiates intersection computations between ray buckets and the ABV. To avoid duplicate fragments on quad diagonals, we avoid conservative rasterization and instead enlarge quads by rounding their coordinates. The fragment shader computes 8^2 ray/volume intersections per fragment, conservatively aggregating depth (min for entry, max for exit) before inserting results into the sorted list. For each ABV type, we store a mapping between the primitive index within the ABV family and its index in $\mathcal{B}$, which we apply in the geometry shader to limit memory reads.

6.2 Synchronized ray processing with pruned tree

Ray buckets are processed in a compute shader to ensure coherent memory access within workgroups. We use a workgroup of size 8^2 per tile T_{uv} leveraging shared memory and on-the-fly tree pruning for faster field evaluation. The compute shader's main loop, detailed in Algorithm 3, includes three stages: tracking active primitives (line 9), building a pruned blobtree view (lines 10 and 13), and processing ray interval using sphere tracing (line 18). Each workgroup thread handles one ray. We *avoid thread divergence* by segmenting the workgroup frustum $\mathcal{F}_{T_{uv}}$ and associated rays into subintervals using a shared depth range $[z_{begin}, z_{end}]$ defined in NDC. As the linked list associated with T_{uv} also uses NDC, this defines a common set of active primitive $\mathcal{A}$ for all ray subintervals to process in a loop turn.

Active primitives. We track the set of active primitives $\mathcal{A}$ in shared memory. Let $z_{b,i}$ and $z_{e,i}$ denote the entry and exit depth of the i-th primitive of tile T_{uv}. At the start of each loop turn (fetchPrimitives, line 8), we remove from $\mathcal{A}$ all primitives verifying $z_{e,i} \leq z_{end}$. To avoid processing empty space, the next range start is defined as:

$$z_{begin} \leftarrow \begin{cases} z_{b,k+1}, \text{if } z_{e,k} \leq z_{end} \\ z_{end}, \text{otherwise} \end{cases} \tag{21}$$

with k the index of the last fetched primitive. Next, we fetch new primitives using a simple heuristic, continuing until any of the following conditions are false:

- $z_{b,k+1}$ <= $\max_{i<k} z_{e,i}$ (do not process empty space)
- $|\mathcal{A}| < M_{overlap}$ (limit the number of active primitives)
- less than 8 primtive fetches

Algorithm 3 ComputeShaderMainLoop(u,v).

1: $r_{found} \leftarrow false$
2: shared: $root\_found[gl\_SubgroupID] \leftarrow false$
3: shared: $z_{begin} \leftarrow 0, z_{end} \leftarrow 0$ ▷ coordinates in NDC
4: shared: $k \leftarrow 0$ ▷ index of last fetched primitive
5: shared: $\mathcal{A} \leftarrow \emptyset$
6: **while** $k < n_{prim} \wedge !\bigwedge_{w \in 1..2} root\_found[w]$ **do**
7: **if** $subgroupID = 0$ **then**
8: **if** glSubgroupElect() **then** ▷ non parallel
9: $\mathcal{A}, k, z_{begin,end} \leftarrow$ fetchPrimitive$(\mathcal{A}, k, z_{begin,end})$
10: buildTreeView($\mathcal{A}$)
11: **end if**
12: synchronize()
13: fetchNodeData() ▷ parallel
14: **end if**
15: synchronize()
16: **if** $!r_{found}$ **then** ▷ process ray subintervals
17: $t_0, t_1 \leftarrow ray(u,v)$.fromNDC$(z_{begin}, z_{end})$
18: $t_{intersect}, r_{found} \leftarrow$ processInterval$(ray(u,v), t_0, t_1)$
19: computeColor$(ray(u,v), t_{intersect})$
20: $root\_found[subgroupID] \leftarrow$ subgroupAnd(r_{found})
21: **end if**
22: synchronize()
23: **end while**

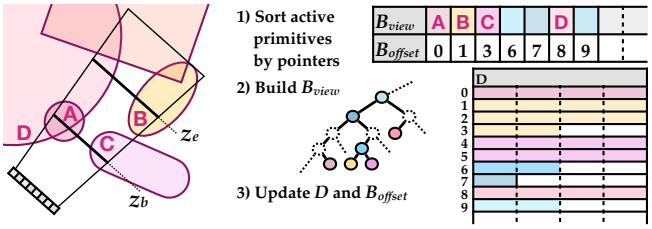


Fig. 11. To process a subfrustum, we first build a pruned blobtree view from the list of active primitives using sparse tree traversal. The view consists of a shared cache storing node parameters D_i and two arrays storing 1) active nodes B_i and 2) offsets of parameters into the local cache.

Here, $M_{overlap}$ is an implementation limit on the number of active primitives that can be stored, and the second condition is crucial to limit shared memory usage. To avoid processing empty space, the end of the subsequent depth range is defined as:

$$z_{end} = \min\left(z_{b,k+1}, \max_{i<k} z_{e,i}\right) . \tag{22}$$

As discussed in Section 5, $\mathcal{A}$ needs to be sorted by increasing indices in $\mathcal{B}$. For simplicity, we perform a single-threaded insertion sort when adding new primitives to $\mathcal{A}$.

Local pruned tree view. Processing grazing ray subintervals requires many field evaluations. To avoid direct access to $\mathcal{B}$ during the evaluation of $f_{\mathcal{A}}$, we use the sparse tree traversal to build a pruned tree view in shared memory. This simplifies the field evaluation loop, enabling a standard bottom-up stackless traversal (outlined in Algorithm 4). As pruning does not require field values, the stack memory

Algorithm 4 Pruned tree evaluation.

$S \leftarrow \{\}$ ▷ empty stack (fixed allocation)
for $i \in [0; 2|\mathcal{A}| - 1[$ **do**
$nodeop \leftarrow$ type$(B_{view}[i])$
$offset \leftarrow B_{offset}[i]$
if isLeaf$(B_{view}[i])$ **then**
$f_{node} \leftarrow$ PRIMITIVE$(nodeop, offset, \mathbf{p})$
else
$f_{right} \leftarrow S$.top(); S.pop()
$f_{left} \leftarrow S$.top(); S.pop()
$f_{node} \leftarrow$ OPERATOR$(nodeop, offset, f_{left}, f_{right})$
end if
S.push(f_{node})
end forreturn S.top()

requirement is reduced, matching a regular bottom-up traversal. Additionally, operators that act as field selectors are processed only during pruning, not during evaluations. We organize the tree view in three arrays (see Figure 11). The largest array, D_{cache}, stores active node parameters $\{D_i\}$. Active nodes $\{B_i\}$ are stored in the B_{view} array, while B_{offset} holds the indices of the corresponding $\{D_i\}$ in D_{cache}. Both B_{view} and B_{offset} have size $2M_{overlap} - 1$.

The tree view is built in two stages after updating the active primitives. First, in line 9 of Algorithm 3, a single thread populates B_{view} via a sparse traversal to store active B_i in traversal order. Next, line 12, D_{cache} is collaboratively filled, and offsets in B_{offset} are updated accordingly. Values of D_i are distributed among threads of a subgroup until all threads have an address, after which parallel read and write operations transfer the data. This is repeated until all nodes in B_{view} are processed, or D_{cache} is full. In the latter case, we record $\{D_i\}$ addresses in B_{offset}.

7 IMPLEMENTATION DETAILS

Our pipeline is implemented in C++ using the OpenGL library (version 4.4), with shaders programmed in GLSL using extensions for subgroup instructions. We present here the GPU memory layout for the blobtree, the default parameters for most of our tests, and their implications for our rendering pipeline.

Blobtree memory layout. We store the blobtree $\mathcal{B}$ in an unsigned integer array and the set of node parameters $\mathcal{D} := \{D_i\}_i$ in a floating-point array (both are GLSL SSBO). Allocation and indexing use a stride of 64-bit for $\mathcal{B}$ and 128-bit for $\mathcal{D}$ which reduces the size of ancestor indices and improves memory alignment. In $\mathcal{B}$, In $\mathcal{B}$, the first 32 bits of each entry store node information B_i, encoded as a bitfield with the following layout:

isLeaf(1)	*isLeft*(1)	*type*(8)	*ancestor*(22)

where values in parenthesis denote the number of bits used per field. We store the starting memory offset of D_i in the next 32-bits. The 22 bits of the ancestor index, combined with the 64-bit stride, allow the indexing of blobtrees containing up to 4.19 million nodes. For operators, we subdivide the *type* property into three parts:

ignoreMod(2)	*isUnary*(1)	*nodeop*(5)

where the two bits of *ignoreMod* encode operand selection based on the operator behavior (union, difference, intersection) and *isUnary*

allows to support unary nodes (more details in Supplemental Material). This bitfield layout can encode up to 256 primitive and 32 operator types. We compile our ray-tracing shader with the 27 primitives and 18 operators shown in Figure 12.

Ray processing. Subintervals along rays are processed using over-relaxed sphere tracing, with an overrelaxation factor of 1.7. We set the Lipschitz bound to 1.45 to obtain artifact-free rendering in all presented examples, accommodating field compression associated with our compact operators. Replacing sphere tracing with segment tracing would offer a parameter-free solution. Instead of limiting the maximum number of steps, as is common in sphere tracing implementations, we only set a minimal step size to 0.005 in world coordinates. For all our examples, the step size is between 1 : 22000 and 1 : 64000 of the implicit volume's depth range.

Memory allocation. Shader register and shared memory usage significantly affect GPU occupancy. In our implementation, register allocation depends on the field evaluation stack size and the chosen primitives and operators. We limit the stack to 8 entries; for *left* comb trees, this does not limit the number of primitives. Right-heavy trees can be re-balanced into left-heavy ones to reduce stack requirements without affecting the field definition [Grasberger et al. 2016]. Shared memory usage is influenced by the size of D_{cache} and $M_{overlap}$ (the size of B_{view} and B_{offset} being twice this value). We set $M_{overlap}$ to 128 and D_{cache} size to 3072 bytes; reaching $M_{overlap}$ results in an average of 6 floats per node.

8 RESULTS

We tested our implementation on two nVidia graphic cards with a significant difference in CUDA cores: the nVidia GeForce RTX 4080 (9728 cores, 16 GB memory) and the nVidia Quadro P1000 (512 cores, 4 GB memory). Most tests were conducted on the GeForce RTX 4080, which runs on a Ubuntu workstation with an AMD Ryzen 9 7950x (4.5GHz, 16-core). The Quadro P1000, running on a Ubuntu laptop with an Intel Core i-7 8850H (2.6GHz, 6 core), was used to validate usability on low-end GPU hardware. We first provide general runtime statistics on complex implicit models, followed by a comparison to a recent state-of-the-art rendering technique for implicit volumes [Keeter 2020].

GPU runtime statistics. We use deferred shading, reporting only G-buffer construction time (i.e., isosurface computation, not shading). Unless otherwise specified, results are averaged over 20 frames at a resolution of 1024 × 1024. Statistics are provided for several user-created examples consisting of hundreds to thousands of primitives (Figure 1, 13, 15 and 14). Runtimes and preprocessing times are detailed in Table 1. Without color computation, rendering times range from interactive on the Quadro P1000 (over 30 fps on simple examples, and 5.5 fps on complex ones) to real-time on the RTX 4080 (over 95 fps on all examples). The RTX 4080 runtimes are 8 to 18 times faster than those of the Quadro P1000, demonstrating good scaling with the number of cores. Computing colors significantly impact the Quadro P1000 runtimes, with only simple examples remaining interactive (over 11fps). In contrast, the RTX 4080 runtimes are only slightly affected (over 83fps). As shown on Figures 1 and 15, per-tile shared memory usage (number of primitive overlaps, node data used) are well below allocated resources (see Section 7).

We also provide more complex procedural examples, see Figures 16 and 18. Table 2 presents associated runtimes measured on the RTX 4080; most examples run over 60 fps. Preprocessing times are listed in Table C.1 of Supplemental Material, they includes ABV generation (with ROI propagation) and GPU blobtree initialization (with fast ancestor computation). This step is not required when only primitives are modified. Our monothreaded implementation does not impair interactivity for volumes with a few thousand primitives; larger trees would benefits from re-using part of the preprocessing.

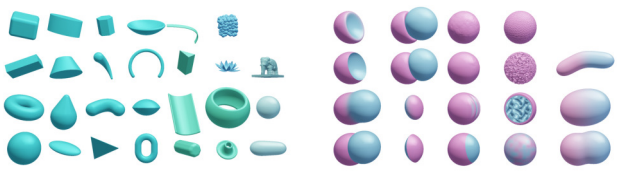


Fig. 12. All primitives (left) and operators (right) loaded in the interpreters for our tests. Primitives include extruded and revoluted 2D SDFs, voxelized SDFs, and normalized density fields. Operators include unary and procedural nodes. See Supplemental Material (Section B).

Fig. 13. Model built mainly from sharp operators; smooth operators are only used for details and have a limited blend range. The model includes procedural primitives (pebbles, flowers) and voxel grids (statues, ducks, and frogs). The environment mapping emphasizes the root-finding precision.

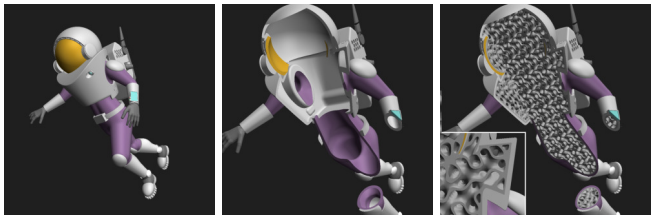


Fig. 14. *Left:* composite model containing multiple unions, intersections, and differences, both sharp and smooth (large blend ranges). *Middle:* We apply a unary operator to carve the shape's interior, then use a difference with a cube to visualize the generated cavity. *Right:* Such an operator can then be extended to generate procedural infills for additive manufacturing, here gyroids. Both images are rendered in less than 5ms.

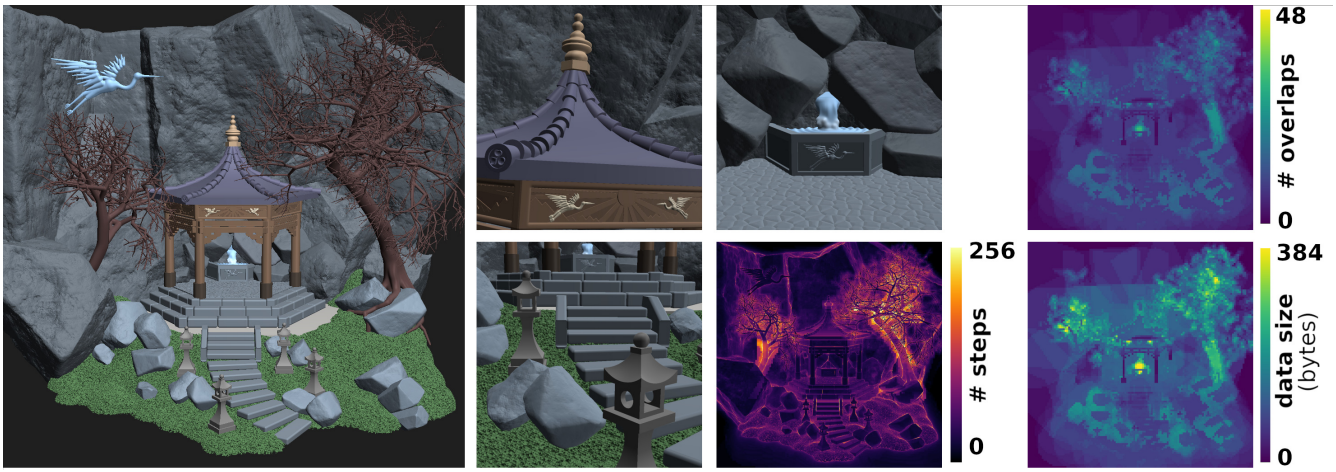


Fig. 15. *Left:* Render of a model consisting of 6275 primitives assembled using smooth and sharp unions, intersections, differences, procedural nodes, and normalized density fields. *Middle:* Close-up views and number of field evaluations per ray (down - clampled for readability), large number of steps can be used locally without impacting interactivity. *Right:* Number of primitive overlaps and size of node parameter cache D per rendering tile, both remain well below implementation limits (color scales correspond respectively to quarter and half of chosen limits).

			nVidia GeForce 4080 RTX						nVidia Quadro P1000	
			Full		Synchronized					
	# of	preprocess	Direct		Direct		View			
Object	primitives	time	ParentID	FastID	ParentID	FastID	FastID	FastID/color	FastID	FastID/color
Dragon (Fig.1)	860	0.5	1755.5	95.1	314.0	18.0	10.5	12.0	169.2	457.5
Mausoleum (Fig.13)	8839	1.8	523.1	33.8	127.9	7.9	4.7	5.2	84.4	118.1
Temple (Fig.15)	6275	1.4	>2s	85.5	872.5	15.6	9.7	11.5	176.9	493.4
Astronaut (Fig.14)	226	0.2	48.6	17.5	15.3	4.9	2.9	3.1	23.2	60.2

Table 1. Rendering and preprocessing times in milliseconds. All variants presented here use the sparse bottom-up traversal for direct field evaluation or tree pruning. *Full/Syncrhonized* correspond to regular processing with a full-resolution A-buffer versus our synchronized tracing. *ParentID/FastID* correspond to the usage of regular parent index versus *fast* ancestor index during the tree traversal. Only variants with *View* build a local pruned tree in shared memory. The last four columns, therefore, correspond to our full pipeline with and without color computations. Preprocessing time of Fig. 13 does not include voxels upload.

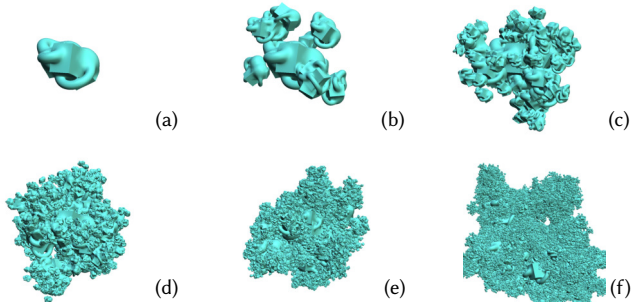


Fig. 16. Synthetic examples displayed with smooth blending. Starting from a base aggregate (a), we generate more complex shapes by instancing 10 subobjects around it. We recursively generate each subobject using the same procedure; recursion levels range from 1 (b) to 5 (f). All resulting aggregates are combined from largest to smallest to limit ROI propagation and detail over-smoothing. Object (f) requires $M_{overlap} = 256$ to be properly rendered.

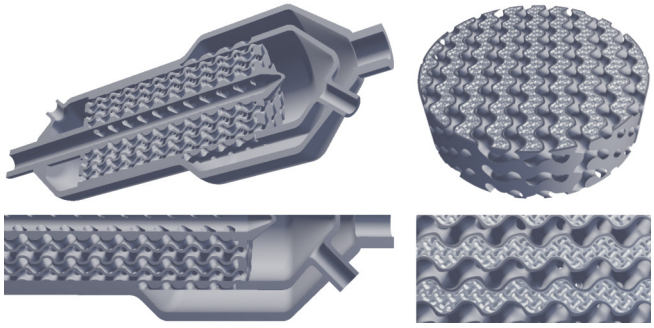


Fig. 17. *Left:* SDFs are used in CAD softwares targeting additive manufacturing [Courter 2019], we reproduce here a heat exchanger using gyroids. *Right:* Implicit shapes also have the capability to represent nested structures [Fryazinov et al. 2013]. Both images are rendered in less than 3.5ms, and 5ms when including the blobtree generation from a lua script.

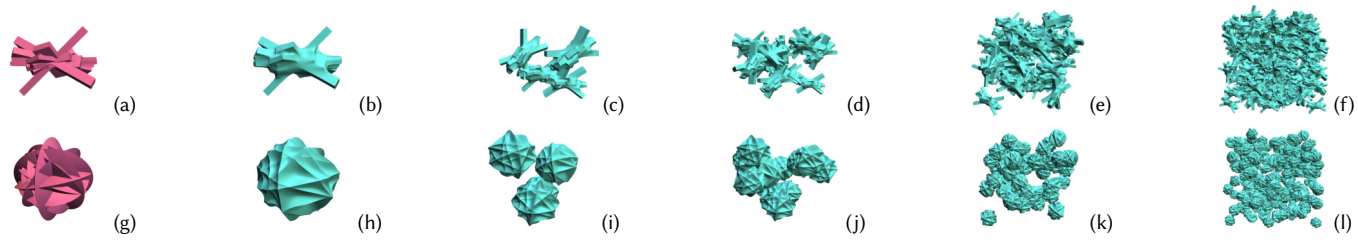


Fig. 18. Synthetic examples for comparison with [Keeter 2020] are displayed here with smooth blending (teal). We build base aggregates as the union (a,g), smooth union (b,h) – of boxes (up) and intersected spheres (down). We generate more complex objects by uniformly distributing base shape instances and combining them with a left-heavy tree. Smooth variants use a blend range proportional to the scaling factor of the repeated object.

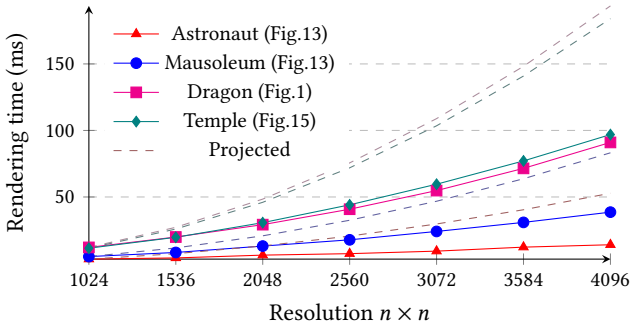


Fig. 19. Rendering times as a function of resolution. Projected graphs correspond to expected growth given runtime time at 1024^2.

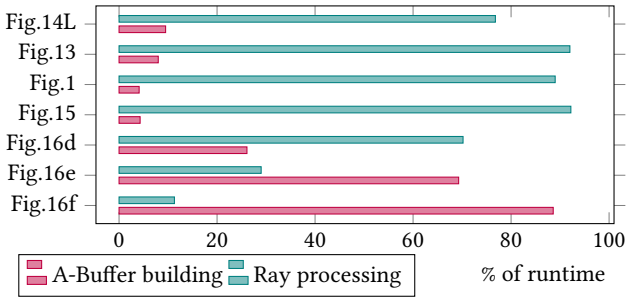


Fig. 20. Repartition of rendering time between A-Buffer creation and ray processing. In the presence of large number of blending primitives the creation of the A-Buffer becomes the bottleneck of the pipeline.

Functional representations are also recognized for their capacity to represent procedural infills for additive manufacturing. This structures can be encoded in the blobtree representation and visualized in real time using our method (see Figure 14 and 17).

Animation. Accompanying videos are generated on the RTX 4080 at a resolution of 2560×1440. For static objects, runtimes are slightly higher than in the paper due to the recording tool and VSync activation. The main video shows assembly of examples presented in Figure 1, 14, and 15. After initial preprocessing (tree linearization and memory allocation), updating primitive and operator parameters take less time, with updates from $24\mu s$ to $450\mu s$. In the two videos of the Figure 13 example, UI sliders trigger the generation of a new blobtree via a Lua script, which is then processed and sent to GPU. This still allows interactive updates for a large number of primitives (up to 0.5 million).

8.1 Analysis

We present runtimes for variants of our pipeline to highlight the impact of our contributions (see Table 1). The results shows that all aspects are crucial for reducing runtime by up to two orders of magnitude compared to non-synchronized tracing using a full-resolution A-buffer and sparse tree traversal with parent indices.

Positioning with respect to [Grasberger et al. 2016]. We base our method on [Grasberger et al. 2016], which employs a spatial acceleration structure that requires updates whenever primitives are modified (i.e., during modeling). In contrast, our approach requires preprocessing only when the tree itself is updated; this involves a simple tree traversal with linear complexity, independent of spatial embedding. Rendering-wise, the acceleration structure in Grasberger et al. [2016] complicates coherent ray traversal, as a given screen tile can span multiple spatial subdivisions simultaneously.

Dependence to resolution. We tested our pipeline at larger resolutions (see Figure 19) and observed that runtimes increase less rapidly than the number of pixels. Although runtimes remain quadratic, they only triple when the number of pixels is quadrupled.

Large A-Buffer. In scenarios with many small blended primitives, as shown in Figures 16e and 16f, the local number of primitive overlaps can increase drastically. In these cases, building the A-buffer becomes the pipeline bottleneck (see Figure 20). When the scene contains only union-like operators, this could be mitigated with an occluder depth pass, rendering primitives without blend [Bruckner 2019]. This optimization could similarly be applied to all primitives with only union-like operators among their ancestors.

Impact of tree structure. We examined how tree layout affects runtime; in Figure 18e, the variation between a left-heavy and a fully balanced tree is less than 9% for both sharp and smooth unions. As previously discussed, the tree structure also impacts stack size requirement, and left-heavy trees are preferable. Due to ROI propagation, adding an operator with a large blend range at the top of the tree can increase runtimes. Therefore, primitives representing details should not be placed below operators that blend coarse shapes: details should modify coarse shapes, not the other way around.

Object	# of primitives	min		smooth and min			smooth only	
		Ours	[Keeter 2020]	Ours $G_\cup$	[Keeter 2020] $g_\cup$	[Keeter 2020] g_{logexp}	Ours $G_\cup$	[Keeter 2020] g_{logexp}
Fig.18b	16	**0.8**	2.1	**6.0**	22.0	8.74	**6.0**	8.74
Fig.18c	64	**1.2**	3.7	**4.4**	33.4	11.9	**4.4**	41.6
Fig.18d	256	**1.7**	11.6	**6.9**	-	24.6	**6.6**	212.4
Fig.18e	1024	**2.0**	53.5	**12.4**	-	81.7	**13.3**	*
Fig.18f	4096	**3**	*	**23**	-	*	**19.0**	*
Fig.18h	24	4.6 / 2.9	**2.6**	**5.0 / 5.8**	16.7	6.4	**5.0** / 6.7	6.4
Fig.18i	96	6.8 / **3.2**	**3.6**	**11.6 / 6.8**	19.6	**7.6**	**13.4 / 6.3**	26.4
Fig.18j	384	10.4 / **4.0**	**5.9**	24.4 / **9.1**	**23.8**	**12.2**	**23.7 / 9.0**	110.6
Fig.18k	1536	15.4 / **4.1**	**13.7**	**36.5 / 8.9**	42.8	**21.9**	**37.1 / 8.6**	*
Fig.18l	6144	**34 / 7.8**	*	**77.4 / 21.6**	*	**39.9**	**61.8 / 14.6**	*
Fig.16a	10	**0.6**	1.2	**1.7**	3.8	2.9	**1.7**	2.9
Fig.16b	110	**0.9**	4.7	**3.5**	-	9.7	**3.4**	6.6
Fig.16c	1110	**1.4**	24.1	**5.6**	-	36.1	**6.0**	71.1
Fig.16d	11110	**2.4**	-	**12.2**	-	-	**11.2**	-
Fig.16e	111110	**6.8**	-	**61.1**	-	-	**58.8**	-
Fig.16f	1111110	**85.3**	-	**468.5**	-	-	**476.0**	-

Table 2. Comparison of rendering times (in milliseconds) using the RTX 4080. If preprocessing takes over 12 minutes or returns a runtime error, runtimes are not provided "-". Columns marked with "*" indicate visualization errors of unknown origin. For Fig.18h-18l, the second value is obtained by replacing intersected spheres with an equivalent lens primitive, providing tighter ABV.

8.2 Comparison to [Keeter 2020]

Before discussing the runtime comparison with [Keeter 2020], it is essential to remember that both methods explore different tradeoffs; the overlap in the types of shapes they can process is partial. The approach in [Keeter 2020] handles any field functions defined through arithmetic operations, excluding procedural operations based on control flow (if, for). In contrast, our method supports the latter, but requires the field function to be an A-SDF defined from a finite set of node types. We use the available CUDA implementation of [Keeter 2020] at [Keeter 2019b] for our comparisons using the RTX 4080.

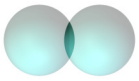
In all examples, every primitives except spheres have their own rotation, a configuration typical in free-form modeling. We test both sharp and smooth unions. For the latter, our implementation utilizes our operator $G_\cup$ (Equation (12)). Since $G_\cup$ relies on if and *isnan* functions, we use the operator $g_\cup$ (Equation (8)) for [Keeter 2020]. Using $g_\cup$ leads to prohibitive preprocessing times, limiting our ability to run comparisons on many examples. Therefore, we also measure the runtime of [Keeter 2020] using another classical blend based on the LogSumExp function, with parameters adjusted to align as closely as possible with $G_\cup$. It is defined as: defined as:

$$g_{logexp}(f_0, f_1, \beta) = -\beta \log(\exp(-f_0/\beta) + \exp(-f_1/\beta)) \qquad (23)$$

Preprocessing. With adequate data management, both methods can change primitive parameters without preprocessing making them well-suited for modeling tasks. For small numbers of primitives combined with unions, which can represent many arithmetic clauses, we observe similar preprocessing times that allow for interactive editing (see Supplemental Material). For [Keeter 2020], preprocessing times exceed 30ms when the number of primitives exceeds a few thousand for union and g_{logexp}, impairing interactivity. For $g_\cup$, most examples could not be preprocessed. In constrast, our implementation remains below 1ms up to a few thousand primitives with smooth and sharp unions, showing linear growth for more complex examples. Our method requires between two and three times less memory to store the expression (below 1MB for all examples but the multiresolution one); however, studied examples are not well-suited for the subexpression re-use property of [Keeter 2020]. It is important to note that preprocessing is not a central aspect of [Keeter 2020], and it could be optimized for both methods.

CSG union. We present rendering time comparisons in Table 2. For models consisting solely of CSG unions (Figure 16 and 18 - top), we obtain runtime reductions from −50.0% to −96.2%. The difference increases with the number of primitives: unlike [Keeter 2020], we never process the full expression during rendering. We observe the largest reduction for models containing around a thousand primitives as larger models result in preprocessing or rendering error when used with [Keeter 2020]. This is among the most favorable cases for our method, as the union generates tight ABV, limiting the number of primitive overlaps and maximizing empty space skipping.

CSG intersection. We designed the model in Figure 18 (g) to challenge our ABV system, using intersections between two overlapping spheres to discard 99% of the primitives' volume (see inset). We then increase the number of overlaps by duplicating the base shape. For a limited number of instances, [Keeter 2020] achieves a rendering time reduction of −43% due to more efficient space discarding. However, using specialized nodes for two sphere intersections significantly improves our method's runtime; the time reduction for [Keeter 2020] drops to −10.3% for a single subobject, while our method shows runtime reductions of −12.5% to −70.1% for larger subobject counts. This optimization can only be applied in specific contexts.



Object	# of primitives	min		smooth and min	
		Ours 1K / 2K / 4K	[Keeter 2020] 1K / 2K / 4K	Ours $G_\cup$ 1K / 2K / 4K	[Keeter 2020] g_{logexp} 1K / 2K / 4K
Fig.22(left)	226	**2.2 / 3.3 / 8.5**	6.3 / 11.0 / 35.5	**3.4 / 5.5 / 13.3**	8.6 / 17.9 / 57.8
Fig.22(middle)	855	**2.7 / 7.4 / 22.2**	11.7 / 17.4 / 56.7	**8.2 / 29.2 / 85.7**	* / * / *
Fig.22(right)	478	**2.2 / 5.1 / 16.8**	13.5 / 24.4 / 84.4	**2.7 / 7.4 / 24.9**	28.3 / 84.8 / 245.5

Table 3. Comparison of rendering times (in milliseconds) using the RTX 4080 for resolution of 1024 × 1024, 2048 × 2048 and 4096 × 4096.

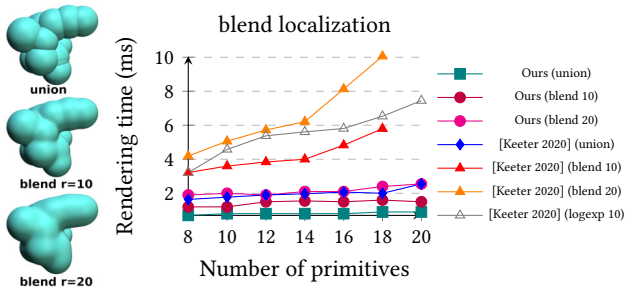


Fig. 21. *Left:* Renders of 16 spheres showing the three operator ranges used for comparison. *Right:* Runtime measured while progressively adding new primitives in the scene. Using classical smooth bounded blending, runtime scales linearly with the number of primitives in the scene. With our compact operators, we observe similar runtimes as the one of [Keeter 2020] using sharp CSG union (min function), which emphasizes the importance of ROIs.

Smooth operator. When using the smooth union $g_\cup$ within subobjects, complex examples could not be rendered with [Keeter 2020]. We observed runtime improvement for all other examples except Figure 18j. Replacing $g_\cup$ by g_{logexp} yields similar results to union and intersection. Although g_{logexp} is unbounded, the union used to combine subobjects allows [Keeter 2020] to discard subexpressions. This is no longer the case when combining subobjects with smooth unions, enabling our method to be over an order of magnitude faster and scale to a larger number of primitives. The importance of compactified blending is emphasized in Figure 21.

User-created examples. To be able to run comparisons on examples of Figures 1, 14 and 15, we stripped the trees of procedural nodes and unsupported primitives, also converting smooth intersections and differences to their sharp variants. We then removed several primitives from two examples to ensure error-free visualization with [Keeter 2020]. Comparisons were conducted with and without smooth unions, showcasing sharp versions of simplified objects in Figure 22. We observed runtime reduction of −64.4% to −86.2% with our method for sharp operators only, and −60.5% to −65.8% for sharp and smooth operators (see Table 3).

8.3 Limitations

Let us briefly remind previously discussed limitations of our rendering pipeline and its implementation: it supports a limited number of primitive and operator types simultaneously, requires preprocessing when the blobtree structure changes, and utilizesa single thread per workgroup for on-the-fly tree pruning. Additionally, A-buffer building time can significantly increase when combining many small primitives with large blend ranges. We also assume a

Fig. 22. Simplified version of the examples of Figure 1, 13 and 15 used for comparison to [Keeter 2020], displayed here with sharp unions.

maximal number of primitive ABV overlaps in space. This scenario is also a bottleneck for most existing techniques as the field evaluation complexity grows linearly with overlaps. Besides the last drawback, these issues stem from our implementation rather than the method itself, strategies like storing partial preprocessing reuse and processing by depth slabs could help alleviate them.

Non compactly supported smooth operators. The proposed approach for AVBs is not directly compatible with C^∞ operators such as g_{logexp} (see Equation 23), which are necessary for defining the smoothest blends. However, these operators can still be utilized within our pipeline if they are restricted to small subtrees just above tree leaves. In this case, local preprocessing can be used to compute ABVs, typically defined by extending the *union* of the ABVs of interacting primitives. While this allows for the use of C^∞ operators, it can lead to an increased number of ABV overlaps in the scene, resulting in longer runtimes. Thus, they should be used sparingly. A similar behavior is observed in [Keeter 2020] (see Table 2).

Secondary rays. The main downside of the proposed pipeline is its focus on primary rays; managing secondary rays would require building a BVH from our ABVs. To leverage hardware acceleration, our implementation should be ported to the Vulkan API. Tracking the set of primitives of interest $\mathcal{A}$ becomes more challenging and typically necessitates the use of an Any Hit shader as the complete list of interacting primitives is needed to evaluate the field on ray subintervals. The evaluation would then benefits from our sparse tree traversal.

Moreover, reduced ray coherency will lead to threads of a workgroup traversing different ABVs, potentially exceeding the $M_{overlap}$ limit of a workgroup. This requires exploring various trade-offs in shared memory allocation, such as sacrificing D_{cache} to allocate more shared memory to $\mathcal{A}$ and B_{view}. An alternative to reduce memory pressure would be to share active primitives among threads during tree traversal using subgroup instructions. Alternatively, a discretized SDF can be computed asynchronously by applying jump flooding to a dense set of isosurface intersections obtained from our pipeline.

9 CONCLUSION AND FUTURE WORK

We have introduced a new rendering pipeline for direct visualization of A-SDF structured in blobtrees. Our pipeline is built on three key ideas: 1) compactified blending operators for defining augmented bounding volumes, 2) a sparse bottom-up tree traversal removing the need for preprocessing when only primitive parameters change, and 3) coherent ray processing using a low-resolution A-buffer for synchronization. Combining these elements significantly improves runtime compared to state-of-the-art methods, especially for large blobtrees. Our pipeline can be easily adapted to density fields and more advanced iterative ray processing algorithms – such as segment tracing (see Section C.2 of Supplemental Material) – though we believe further enhancements are possible for better integration of density fields and A-SDF. Several extensions warrant exploration, including support for ternary blobtree nodes [Pasko et al. 2005], gradient-based operators [Gourmel et al. 2013], as well as interval queries in volumes to preprocess subfrustums more efficiently. Finally, since we store all primitives – not just the visible ones – in the A-buffer, the framework can be adapted for slicing implicit volumes using the direct slicing approach of [Lefebvre 2013].

ACKNOWLEDGMENTS

I would like to thank Sylvain Lefebvre, Xavier Chermain and David Jourdan for their helpful feedback during the writing of this paper. The work is supported by the Agence nationale de la recherche of France under Grants No.: ANR-18-CE46-0004, ANR-22-CE33-0018.

REFERENCES

2021. MagicaCSG. https://ephtracy.github.io/index.html?page=magicacsg

Melike Aydinlilar and Cédric Zanni. 2021. Fast ray tracing of scale-invariant integral surfaces. *Computer Graphics Forum* 40, 6 (Sept. 2021), 117–134. https://doi.org/10.1111/cgf.14208

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

Stefan Bruckner. 2019. Dynamic Visibility-Driven Molecular Surfaces. In *Computer Graphics Forum*, Vol. 38(2). Wiley Online Library, 317–329. https://doi.org/10.1111/cgf.13640

Boris Burger, Ondrej Paulovic, and Milos Hasan. 2002. Realtime visualization methods in the demoscene. In *Proceedings of the central european seminar on computer graphics*.

Csaba Bálint and Gábor Valasek. 2018. Accelerating Sphere Tracing. In *EG 2018 - Short Papers*, Olga Diamanti and Amir Vaxman (Eds.). The Eurographics Association. https://doi.org/10.2312/egs.20181037

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

Daniel Dekkers, Kees Van Overveld, and Rob Golsteijn. 2004. Combining CSG modeling with soft blending using Lipschitz-based implicit surfaces. *The Visual Computer* 20, 6 (2004), 380–391. https://doi.org/10.1007/s00371-002-0198-3

Tom Duff. 1992. Interval arithmetic recursive subdivision for implicit functions and constructive solid geometry. *SIGGRAPH Comput. Graph.* 26, 2 (jul 1992), 131–138. https://doi.org/10.1145/142920.134027

Eva Dyllong and Cornelius Grimm. 2007. Verified Adaptive Octree Representations of Constructive Solid Geometry Objects.. In *SimVis*. Citeseer, 223–236.

Mark Fox, Callum Galbraith, and Brian Wyvill. 2001. Efficient Use of the BlobTree for Rendering Purposes. In *Proceedings of the International Conference on Shape Modeling & Applications (SMI '01)*. IEEE Computer Society, Washington, DC, USA.

Oleg Fryazinov, Alexander A. Pasko, and Peter Comninos. 2010. Fast reliable interrogation of procedurally defined implicit surfaces using extended revised affine arithmetic. *Computers & Graphics* 34 (2010), 708–718.

Oleg Fryazinov, Turlif Vilbrandt, and Alexander Pasko. 2013. Multi-scale space-variant FRep cellular structures. *Computer-Aided Design* 45, 1 (2013), 26–34.

Eric Galin, Eric Guérin, Axel Paris, and Adrien Peytavie. 2020. Segment Tracing Using Local Lipschitz Bounds. *Computer Graphics Forum* (2020).

Olivier Gourmel, Loïc Barthe, Marie-Paule Cani, Brian Wyvill, Adrien Bernhardt, Mathias Paulin, and Herbert Grasberger. 2013. A gradient-based implicit blend. *ACM Trans. Graph.* 32, 2 (April 2013), 12:1–12:12. https://doi.org/10.1145/2451236.2451238

Olivier Gourmel, Anthony Pajot, Mathias Paulin, Loïc Barthe, and Pierre Poulin. 2010. Fitted BVH for Fast Raytracing of Metaballs. *Computer Graphics Forum* 29, 2 (2010).

Herbert Grasberger, Jean-Luc Duprat, Brian Wyvill, Paul Lalonde, and Jarek Rossignac. 2016. Efficient data-parallel tree-traversal for BlobTrees. *Computer-Aided Design* 70 (2016), 171–181.

John C. Hart. 1996. Sphere tracing: a geometric method for the antialiased ray tracing of implicit surfaces. *The Visual Computer* 12, 10 (1996), 527–545. https://doi.org/10.1007/s003710050084

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

Matthew Keeter. 2019b. mpr : Reference implementation for "Massively Parallel Rendering of Complex Closed-Form Implicit Surfaces". https://github.com/mkeeter/mpr

Matthew J. Keeter. 2020. Massively Parallel Rendering of Complex Closed-Form Implicit Surfaces. In *Proceedings of SIGGRAPH*.

Benjamin Keinert, Henry Schäfer, Johann Korndörfer, Urs Ganse, and Marc Stamminger. 2014. Enhanced Sphere Tracing. In *Smart Tools and Apps for Graphics - Eurographics Italian Chapter Conference*. https://doi.org/10.2312/stag.20141233

Aaron Knoll, Younis Hijazi, Andrew Kensler, Mathias Schott, Charles Hansen, and Hans Hagen. 2009. Fast ray tracing of arbitrary implicit surfaces with interval and affine arithmetic. In *Computer Graphics Forum*, Vol. 28. Wiley Online Library, 26–40.

Sylvain Lefebvre. 2013. IceSL: A GPU Accelerated CSG Modeler and Slicer. In *AEFA'13, 18th European Forum on Additive Manufacturing*. Paris, France.

Sylvain Lefebvre, Samuel Hornus, and Anass Lasram. 2014. Per-Pixel Lists for Single Pass A-Buffer. In *GPU Pro 5: Advanced Rendering Techniques*, Wolfgang Engel (Ed.). A K Peter / CRC Press. https://hal.inria.fr/hal-01093158

Gábor Liktor. 2008. Ray tracing implicit surfaces on the GPU. In *12th Central European Seminar on Computer Graphics (CESCG 2008 Proceedings). Technische Universität Wien, Institut für Computergraphik und Algorithmen.*

Don P. Mitchell. 1990. Robust ray intersection with interval arithmetic.

Media Molecule. 2020. Dreams. http://dreams.mediamolecule.com/

Tomoyuki Nishita and Eihachiro Nakamae. 1994. A Method for Displaying Metaballs by using Bezier Clipping. *Comput. Graph. Forum* 13 (08 1994), 271–280. https://doi.org/10.1111/1467-8659.1330271

Second Order. 2018. Claybook. https://www.claybookgame.com/

A. Pasko, V. Adzhiev, A. Sourin, and V. Savchenko. 1995. Function representation in geometric modeling: concepts, implementation and applications. *The Visual Computer* 11, 8 (1995), 429–446.

G. Pasko, A. Pasko, M. Ikeda, and T. Kunii. 2002. Bounded Blending Operations. In *Proceedings of the Shape Modeling International 2002 (SMI'02) (SMI '02)*. IEEE Computer Society, Washington, DC, USA, 95–.

Galina I. Pasko, Alexander A. Pasko, and Tosiyasu L. Kunii. 2005. Bounded Blending for Function-Based Shape Modeling. *IEEE Comput. Graph. Appl.* 25, 2 (March 2005), 36–45. https://doi.org/10.1109/MCG.2005.37

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

Inigo Quilez and Pol Jeremias. 2013. Shadertoy. https://www.shadertoy.com/

Tim Reiner, Sylvain Lefebvre, Lorenz Diener, Ismael García, Bruno Jobard, and Carsten Dachsbacher. 2012. A runtime cache for interactive procedural modeling. *Computers & Graphics* 36, 5 (2012), 366–375.

Tim Reiner, Gregor Mückl, and Carsten Dachsbacher. 2011. Interactive modeling of implicit surfaces using a direct visualization approach with signed distance functions. *Computers & Graphics* 35, 3 (2011), 596–603.

A. Ricci. 1973. Constructive geometry for computer graphics. *Computer Journal* 16, 2 (1973), 157–60.

Ryan Schmidt, Brian Wyvill, and Eric Galin. 2005. Interactive Implicit Modeling with Hierarchical Spatial Caching. In *Proceedings of the International Conference on Shape Modeling and Applications 2005 (SMI '05)*. IEEE Computer Society. https://doi.org/10.1109/SMI.2005.25

Nicholas Sharp and Alec Jacobson. 2022. Spelunking the deep: guaranteed queries on general neural implicit surfaces via range analysis. *ACM Transactions on Graphics (TOG)* 41, 4 (2022), 1–16.

Andrei Sherstyuk. 1999. Fast Ray Tracing of Implicit Surfaces. *Comput. Graph. Forum* 18 (1999), 139–147.

Towaki Takikawa, Andrew Glassner, and Morgan McGuire. 2022. A dataset and explorer for 3D signed distance functions. *Journal of Computer Graphics Techniques* 11, 2 (2022).

Ireneusz Tobor, Patrick Reuter, and Christophe Schlick. 2004. Multi-scale reconstruction of implicit surfaces with attributes from large unorganized point sets. In *Proceedings Shape Modeling Applications, 2004.* IEEE, 19–30.

Brian Wyvill, Eric Galin, and Andrew Guy. 1997. The Blob Tree, Warping, Blending and Boolean Operations in an Implicit Surface Modeling System. *University of Calgary technical report* (July 1997).

Brian Wyvill, Andrew Guy, and Eric Galin. 1999. Extending the CSG Tree. Warping, Blending and Boolean Operations in an Implicit Surface Modeling System. *Computer Graphics Forum* 18, 2 (1999), 149–158. https://doi.org/10.1111/1467-8659.00365

Geoff Wyvill, Craig McPheeters, and Brian Wyvill. 1986. Data structure for soft objects. *The Visual Computer* 2, 4 (Aug. 1986), 227–234.