HCSG: Hashing for real-time CSG modeling

Cédric Zanni, Frédéric Claux, Sylvain Lefebvre

HCSG: Hashing for real-time CSG modeling

CÉDRIC ZANNI, Université de Lorraine, CNRS, Inria, LORIA, France
FRÉDÉRIC CLAUX, XLIM - Université de Limoges, France
SYLVAIN LEFEBVRE, Université de Lorraine, CNRS, Inria, LORIA, France

Constructive Solid Geometry models solids as boolean combinations of base primitives. It is one of the classical modeling approaches in Computer Graphics. With the advent of 3D printing, it has received a renewed interest: CSG affords for the robust definition of solids, and fits well with parametric modeling, affording for easy customization of existing designs.

However, the interactive display and manipulation of CSG models is challenging: Ideally, CSG has to be performed between a variety of solid representations (meshes, implicit solids, voxels) and the renderer has to provide immediate feedback during parameter exploration. The end result has to be prepared for fabrication, which involves robustly extracting cross-sections of the model.

In this work we propose a novel screen space technique for the rendering, interactive modeling and direct fabrication of parametric CSG models. It builds upon spatial hashing techniques to efficiently evaluate CSG expressions, checking whether each interval along a view ray is solid in constant time, using constant local shader memory. In addition, the scene is rendered progressively, from front to back, bounding memory usage. We describe how the hash encoding the CSG is constructed on the fly during visualization, and analyze performance on a variety of 3d models.

CCS Concepts: • **Computing methodologies** → **Rendering**; *Shape modeling*;

Additional Key Words and Phrases: CSG, rendering, slicing, hash, customization, parametric modeling

ACM Reference Format:
Cédric Zanni, Frédéric Claux, and Sylvain Lefebvre. 2018. HCSG: Hashing for real-time CSG modeling. *Proc. ACM Comput. Graph. Interact. Tech.* 1, 1, Article 15 (May 2018), 19 pages. https://doi.org/10.1145/3203198

1 INTRODUCTION

The advent of 3D printing has spawned a renewed interest for solid modeling approaches, in particular Constructive Solid Geometry (CSG). Software such as OpenSCAD allow designers and makers to create parametric shapes described as scripts, which once executed produce a geometry that can be fabricated on 3D printers. The parameters can be exposed in specialized interfaces, e.g. [Makerbot 2013; Shugrina et al. 2015], allowing anyone to easily customize a part before its fabrication. Similarly, libraries of parameterized shapes such as gears or pins can be created and shared.

In this paper we focus on the core technology that supports such approaches. Regardless of the choice of modeling interface (scripts, nodes, 3D GUI), these approaches typically output a CSG tree: a hierarchy of composition operations (e.g. union, subtraction, intersection) applied to solid primitives (e.g. spheres, cubes).

From the CSG tree, two essential operations have to be performed. The first is to provide a real time preview of the object being modeled. It is critical for this preview to be fast and responsive to

Authors' addresses: Cédric Zanni, Université de Lorraine, CNRS, Inria, LORIA, F-54000, Nancy, France, cedric.zanni@loria.fr; Frédéric Claux, XLIM - Université de Limoges, F-87000, Limoges, France, frederic.claux@unilim.fr; Sylvain Lefebvre, Université de Lorraine, CNRS, Inria, LORIA, F-54000, Nancy, France, sylvain.lefebvre@inria.fr.

changes in the parameters, especially when customization interfaces are used. The second is to prepare the geometry for fabrication. In many software this is understood as outputting a triangle mesh in the STL format. However, several researchers have advocated for directly producing printer instructions [Pasko et al. 2011; Vidimče et al. 2013]. This avoids the expensive and numerically challenging task of computing a triangulated 3D mesh.

The core operation for preparing the model for 3D printing is called *slicing*. It consists in extracting cross sections of the geometry by intersecting a plane with the solid to be manufactured. Interestingly, the *same* algorithm can be used for both rendering and slicing: slicing amounts to rendering an internal slice of the object from a top view of the print platform [Lefebvre 2013].

Challenges. Our algorithm builds upon fast GPU construction of A–buffers, which have been previously employed for transparency [Maule et al. 2012, 2011; Thibieroz 2011]. The principle is to store, in each pixel, the list of all fragments produced by the rasterization of primitives. Assuming all primitives are closed, after sorting the fragments by depth a *dexel buffer* [Hook 1986] is obtained: a list of overlapping interior intervals belonging to different primitives. CSG operations can be directly applied on the intervals – independently in each pixel – to determine the solid intervals of the resulting volume [Glassner et al. 1989].

This is a versatile approach, which makes it possible to combine triangle meshes with primitives ray-traced from shaders (implicits, volumes). The main drawback stems from the fact that all fragments have to be stored and sorted before resolving the CSG. Indeed, fragments are produced in arbitrary order by the rasterizer (within and across primitives), and thus solid intervals cannot be determined before the end of the rasterization. This incurs a large memory cost for storing all fragments across all pixels. This is problematic for rendering, and becomes a critical issue for slicing which is typically performed at a much higher resolution (e.g. below $50\mu m$ per pixel).

Splitting and rendering the view in multiple XY screen-space tiles partially addresses this problem. However, the issue remains for models with high depth complexity. Furthermore, sorting the fragments can be made very efficient using simple parallel insertions sorts with atomics, but only at low numbers of fragments per rays. Thus, it is desirable to reduce per-ray fragment pressure by splitting the view in slabs along depth. Finally, and most importantly in our context, for slicing it is important to be able to stream slices to the printer as they are produced: slices can represent very large amounts of data, up to terabytes [Vidimče et al. 2013]. For slice extraction, it is best to align the Z axis along the build direction, since the slices may be non uniformly distributed (e.g. adaptive slicing [Dolenc 1994]). In this case XY tiling does not allow for progressive tile extraction as each tile spans the entire depth range.

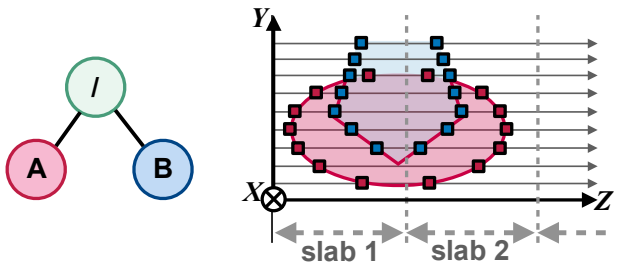


Fig. 1. Two objects in difference and fragments along view rays (aligned with Z). Processing slab 2 requires knowing the in/out state along each ray at the end of slab 1.

If splitting along Z, depth slabs could be rendered from front to back, each containing a small subset of the overall fragments (Figure 1). This would bound storage of fragments to a single slab at a time. However, some primitive interior intervals are likely to span across slab boundaries. Thus, the in/out state of each primitive in between slabs has to be recorded – which quickly becomes impractical as the number of primitives increases. Our approach completely eliminates this issue, allowing to store *a single 64-bits integer* to record the current status in between slabs. This makes the depth slabs approach possible, and hence provides bounds on the memory requirements.

Another difficulty relates to the evaluation of the CSG expression itself: how to determine the final solid intervals given all the interior intervals of all the primitives. This evaluation may require additional memory (e.g. an execution stack). As the number of primitives and the complexity of the CSG expression increases, the size of the required local memory keeps growing. Due to the hardware architecture, using more memory in per-pixel shader invocations reduces parallelism and performance (fewer threads run on a same multi-processor [Ryoo et al. 2008]). Our algorithm makes the evaluation of the CSG expression independent from the number of primitives and scene complexity, effectively determining the result of the CSG expression in subsequent intervals in *constant time*, requiring a *constant amount* of per-shader invocation memory.

Contributions.

- A novel way to encode and access CSG expressions in hash tables, replacing the evaluation of the CSG boolean expression by a constant time lookup, and encoding the current state of the CSG evaluation along a ray as a single integer.
- An algorithm to build the CSG hash tables dynamically, as the scene is explored or modified by the user.
- Implementation details and optional optimizations, as well as a complete analysis of the algorithm behavior.

Our approach allows modeling with a variety of primitives, provides memory bounds during rendering/slicing and affords for fast dynamic updates during parameter exploration. It does not require any transformation of the CSG tree.

2 RELATED WORK

Constructive Solid Geometry is one of the classical way of modeling in Computer Graphics [Ricci 1973]. We focus only on the approaches most closely related to our work.

2.1 Extracting meshes from CSG models

A first line of approaches considers on the problem of performing boolean operations between input meshes, and output a result mesh e.g. [Barki et al. 2015; Feito et al. 2013; Ogayar-Anguita et al. 2015]. This a numerically and computationally challenging task. Recent approaches focus computations around regions of interest [Douze et al. 2015; Schmidt and Brochu 2016] and address robustness issues [Zhou et al. 2016].

An alternative is to discretize the solids into volumes such as voxel distance fields, redefining boolean operations in terms of min/max operators or blends [Wyvill et al. 1999]. The volume resulting from the CSG expression is then polygonized [Newman and Yi 2006]. Modern GPU techniques allow to perform these operations very efficiently [Eisemann and Décoret 2008; Wang 2011].

While impressive given the difficulty of the task they perform, mesh CSG techniques remain too slow for immediate, real-time feedback during parameter exploration.

2.2 Screen-space rendering of CSG models

Another line of research focuses solely on the problem of *displaying* the shape resulting from CSG operations. These approaches are referred to as screen space techniques: the boolean operations are resolved on-screen during display. The works of Goldfeather et al.[1989] and Epstein et al. [1989] pioneered these techniques, that were later improved to run at interactive speeds on available graphics hardware [Hable and Rossignac 2005; Rossignac 2011, 1998; Stewart et al. 1998].

Screen space CSG techniques face two distinct problems. The first is to determine the intersections with the primitives, the second is to evaluate the CSG expression such as to determine which intersections actually belong to the resulting surface[1]. These two problems are often intermixed in screen-space techniques, for maximum efficiency. A third, orthogonal problem is to accelerate the process through spatial data–structures, for instance restricting the local complexity of the CSG scene by splitting it into a pre-computed spatial subdivision [Romeiro et al. 2008]. These approaches could be applied to most of the other techniques – including ours – and we will therefore not detail them.

Determining intersections. Determining intersection with the primitives largely depends on the choice of geometric representation. If only analytic primitives are used intersections can be obtained either directly [Glassner 1989] or by sphere–tracing/ray-marching [Kalra and Barr 1989]. This can have a dramatic impact on the design of the algorithm. For instance Ulyanov et al. [2017] embed intersection computations directly within the evaluation of the CSG expression, building upon the fact that analytic primitives (spheres, cones, boxes) return intersections in constant time. Combined with the fact that only the first intersection is searched (opaque rendering), the algorithm can be further optimized to skip entire CSG subtrees, performing computations in each pixel independently. In such an algorithm, the cost of evaluating intersections is comparable to the cost of evaluating the CSG expression.

However, when triangle meshes or implicits requiring ray marching are also considered, the cost of evaluating intersections significantly increases. It becomes essential to rely on the hardware rasterizer to find all the intersections along view rays – which are referred to as *fragments*. This is much more efficient as computations are amortized across pixels and performed in a coalesced, GPU efficient manner. In particular, the hardware rasterizer is highly optimized to rasterize hundreds of thousands of triangles without any auxiliary spatial data-structure.

For this reason, many algorithms perform depth–peeling through rasterization [Hable and Rossignac 2005; Rossignac 2011, 1998]: surfaces are drawn multiple times and CSG is resolved one depth layer (peel) at a time. Similarly, recent approaches for CSG and order independent transparency build A-buffers on the fly, recording all the incoming rasterized fragments in per-pixel lists, which are later processed for rendering [Maule et al. 2011; Thibieroz 2011]. Each per-pixel list provides the full list of intersections along the ray corresponding to the pixel. Our algorithm builds upon the same advantages, and exploit per-pixel lists of fragments.

An obvious drawback of per-pixel lists is their memory cost, and hence one of our contribution is to bound this cost by processing the view in depth slabs – this also allows to terminate early when only opaque rendering is desired.

We next discuss how the CSG expressions can be evaluated, given the list of fragments.

Evaluating the CSG expression. We now consider how different methods evaluate the CSG expression. The interior intervals of the primitives define a partition of the view ray. The CSG expression has to be evaluated on the intervals of these partitions to determine which are solid. Equivalently, most approaches determine how the CSG expression changes when *crossing* an interval boundary –

[1]It is worth noting that intersections with the CSG result are always a subset of the intersections with the base primitives

a primitive intersection – to find out which are boundaries of the actual solid. Most algorithms test the interval boundaries in a depth ordered sequence, from closest to furthest.

We are interested in two properties to compare the algorithms. The first is the number of interior checks, i.e. how many times the algorithm tests whether a fragment lies inside an input primitive. The second is the memory required to execute the evaluation. Using more memory penalizes parallelism.

The Blister algorithm [Hable and Rossignac 2005; Rossignac 2011, 1998] guarantees that the CSG expression can be evaluated using only 8 bits of temporary memory, making it perfectly suited for depth peeling using a stencil buffer. However, it requires interior checks on *all* primitives each time an interval boundary is evaluated. The algorithm of Kensler [2006], amenable to GPU [Ulyanov et al. 2017], performs interior checks on a subset of the primitives but requires a stack during evaluation – which size depends on the form of the CSG expression. The CSG expression is thus optimized to reduce the stack size and improve performance. The approach of Lefebvre et al. [2014] builds a GLSL boolean expression sent directly to the shader compiler. It is then automatically optimized as any other boolean expression to reduce register usage and perform short circuiting[2]. Thus, the number of interior checks and amount of required memory is minimized by the compiler. This approach tracks the inside/outside status of each primitive in a boolean array while fragments are visited in depth order. This further increases the required memory.

In contrast to these approaches, our algorithm evaluates the expression using constant memory (64 bits) and traverses the ordered fragment list only once (e.g. no additional interior checks). This is made possible by building upon the fact that interval boundaries are tested in an ordered sequence, and by using a hashing scheme to perform the CSG evaluation of the intervals.

3 HASH-BASED CSG EVALUATION

Our technique relies on the rasterizer to produce a list of fragments in each pixel – an A-buffer. This is implemented using texture buffers and atomics to insert and sort fragments by depth, as described in [Lefebvre et al. 2014]. We modify the approach to only record fragments within a given depth slab. Any equivalent technique, such as depth peeling or k-buffers [Vasilakis et al. 2015] could be used. For simplicity, in our implementation the depth slabs uniformly split depth between the near and far planes (the algorithm supports any other subdivision).

After this step, the A-buffer contains a list of fragments per-pixel. Each i-th fragment has a record $(f, d, id)_i$, where $f \in [-1, 1]$ indicates whether the fragment is front facing (1) or back facing (-1), d is its depth, and id is a 32-bits primitive identifier. As fragments are sorted, we have $d_i \leq d_{i+1}$.

The list of sorted fragments divides the view ray into intervals, and our goal is to classify each as either solid or empty following the CSG expression. This is done from the shader which renders the view or the slices during slicing.

The structure of the classification loop is given in Algorithm 1. Note the simplicity of the performed operations: the main variable involved, c, is an integer. It is first read from the previous slab line 2 (initialized to zero for the first slab), then updated with a single addition line 6, and output for the next slab in line 15. This simplicity is due to the table CSGHash, accessed lines 3 and 7, and which returns in constant time whether c represents a solid or an empty interval.

In our terminology the variable c represents a *combination* of primitives: the set of primitives which enclose the current interval in their interior. By using hashing, we are able to compress this information within only 64-bits. This might be surprising, however our scheme is inspired by spatial

[2]The right side of a and is not evaluated if the left side is `false`, see https://en.wikipedia.org/wiki/Short-circuit_evaluation and https://www.khronos.org/registry/webgl/sdk/tests/conformance/glsl/misc/shader-with-short-circuiting-operators.html

ALGORITHM 1: Classify intervals for a pixel in a depth slab.

```
1  Function classifyIntervalsForPixel(x,y)
2      int64 c = ReadValueFromPreviousDepthSlab(x,y);
3      bool prev_status = CSGHash[c];
4      record cur = FirstFragment(x,y);
5      while cur exists do
6          c = c + Value(cur);
7          status= CSGHash[c];
8          if status ≠ prev_status then
9              # perform application dependent action
10             # exit if only first is required
11         end
12         prev_status = status;
13         cur = NextFragment(cur);
14     end
15     WriteValueForNextDepthSlab(x,y,c)
16 end
```

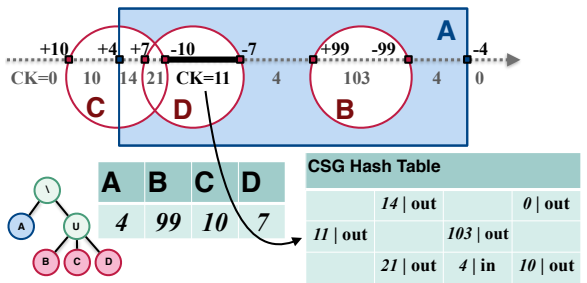


Fig. 2. Processing of a fragment list. The combination key is updated each time a fragment is crossed. The key is used to query the in/out status in the CSG hash table.

hashing techniques which are able to access *millions* of texels from texture coordinates [Alcantara et al. 2009; García et al. 2011].

We further discuss in Section 3.1 how c is computed from primitive values (added to c line 6). The hash construction is discussed in Section 4. Algorithm 1 assumes that CSGHash is pre–computed, but we also discuss its dynamic construction, during display. We discuss further optimizations in Section 5. Finally we analyze our approach and provide performance comparisons in Section 7

3.1 Hashing for classification of CSG intervals

Our hashing scheme is based on two components. The first is the computation of a *combination key* representing the set of primitives enclosing a current interval (c in Algorithm 1). The second is a CSG hash table, accessed with the key, and which uniquely encodes whether a combination of interval is solid or empty, or whether it has never been observed before (CSGHash in Algorithm 1).

For now, let us assume that the CSGHash table is given. We describe in Section 4 its efficient construction, either as a (fast) pre-process, or dynamically during display.

3.1.1 Combination keys. We associate each primitive with a non zero integer, which we call its *primary value*. This integer is chosen randomly, uniformly within the set of 32-bits integers. We

ensure different primitives receive different integers. The primary values are stored in an array P_{val} indexed from the primitive id. The combination key $c(I)$ is the sum of the primary values of all primitives enclosing I, denoted as the set $\mathcal{P}(I)$:

$$c(I) = \left(\sum_{j \in \mathcal{P}(I)} P_{val}[j] \right) \mod 2^n$$

Due to this definition, the empty combination always evaluates to 0. The value of $c(I)$ is directly computed during sequential traversal of the fragment list: in Algorithm 1 we set Value(cur) = $f_i \,.\, P_{val}[id_i]$ with i the index of the current fragment cur.

An important consideration is whether intervals I, J such that $\mathcal{P}(I) \neq \mathcal{P}(J)$ may result in the same combination keys, e.g. $c(I) = c(J)$. We refer to this situation as an *ambiguity* between the keys (not to be confused with collisions in the CSG hash table). We detail how ambiguities are avoided in Section 4.

Assuming no ambiguity exists, we can use the keys c to access a pre-computed hash table which stores whether c is a solid or empty interval, providing the answer in constant time. Access to the hash table is described next.

3.1.2 Accessing the CSG hash table. To store the interval status (solid/empty) in the hash table we rely on coherent parallel hashing [García et al. 2011], a Robin-Hood open addressing scheme.

The access pseudo-code is given in Algorithm 2. With this hashing scheme, elements are inserted at a given *age* : the rank at which insertion succeeded along the pseudo-random insertion sequence. The access algorithm searches the element iteratively starting from age = 0 up to a max age precomputed during insertion (loop in Algorithm 2, line 5). Thus, the age of the elements is the main factor in access performance. The Robin-Hood insertion strategy is designed to keep the age of the keys low (average and max): even in almost full tables, the loop is expected to terminate in a few iterations [García et al. 2011]. However, since our tables are small (< 1 MB) we maintain them only half-filled, which results in even lower ages. The lookup typically returns in less than two iterations (see Table 1). The function *hash* (Algorithm 2, line 6) only involves basic arithmetic on *pk* and *age*, please refer to [García et al. 2011].

ALGORITHM 2: Accessing the hash.

```
1  Function CSGHash(int64 c)
2      int32 pk = PrimaryKey(c);
3      int32 sk = SecondaryKey(c);
4      M = MaxAge(hash(pk, 0));
5      for ( age = 0; age <M; age = age + 1 ) {
6          addr = hash(pk, age) ;
7          uint64 rec = CSGHash[addr];
8          if rec.PKey = pk and rec.SKey = sk then
9              return rec.Solid;
10         end
11     }
12     # Unknown combination
13     return false;
14 end
```

The bit layout of the table records is:

Solid(1)	PKey(32)	SKey(23)	Age(4)	MaxAge(4)

Each record stores the status of the CSG combination (solid or empty, 1 bit), the key used to store the record (primary and secondary to remove combination ambiguities, see Section 4.3), the age (4 bits, only for insertion), and maximal age (4 bits).

Note how the lookup may terminate without finding a key, line 12. We describe next how to use this to discover new combinations on-the-fly, during access.

4 HASH TABLE CONSTRUCTION

In section 3 we have discussed how our approach renders a CSG scene assuming the hash table already exists. The hash table is accessed from combination keys that represent sets of primitives enclosing an interval in their interiors. It returns whether this interval is solid or empty.

Conceptually, the CSG hash table has to contain all possible combinations of primitives. Of course, this is impractical since for P primitives one can construct 2^P arrangements. Fortunately, in a given CSG model only few of these combinations actually exists. However, it would be hard to robustly analyze the geometry and determine these combinations ahead of time. Instead, our hash table construction exploits rendering to discover combinations as they appear during display.

We discuss in Section 4.1 the on-the-fly construction of the hash, starting from an empty table. New combinations of primitives are detected during rendering, and dynamically inserted in the table. We explain in Section 4.2 how to exploit this process to discover combinations in batches, as a quick pre-process. This mitigates the cost of on-line discovery, ensuring a smooth user experience immediately after a new model is loaded. We describe in Section 4.3 how possible ambiguities during access to the hash are addressed. Finally, we discuss in Section 4.4 how to filter rasterization artifacts such as z-fighting.

4.1 On-the-fly discovery of combinations

In this mode, unknown combinations are detected during Algorithm 1, when accessing the hash with Algorithm 2 (see line 12). Whenever CSGHash cannot find a combination, it is recorded in a buffer (GPU side). An algorithm running on the CPU reads back this buffer and inserts the missing combinations in hash table. Rendering is not interrupted: missing combinations default to empty, and the evaluation is pursued. In particular, solid intervals located further down the list will be properly rendered, allowing the view to be progressively refined. The process starts with an empty hash table, allocating a fixed amount of memory (1 KB). Records are progressively inserted, triggering a reallocation (in the std::vector manner) whenever necessary. We detail these steps next.

At a given pixel we only store the first encountered missing combination, and limit the number of new combinations that are discovered during a same frame. We prioritize the missing combinations which are observed in multiple pixels. This is done by performing a hierarchical reduction of the buffer holding missing combinations per-pixel, prioritizing a combination that appears multiple times, and otherwise selecting at random. Combinations not processed at a given frame will be discovered again at the next frame. This process converges quickly (see Section 7.2).

To allow fast transfers between CPU and GPU, we record a missing combination as a difference with respect to the last known combination: We store the key of the last known combination, as well as the fragment record that triggered the unknown combination. In practice, we also send back a few additional fragments after the missing combination (16 in our implementation) as the combinations produced by the next fragments are also likely to be missing. This allows the CPU to process them very efficiently, since they are in sequence along the same view ray.

After having been retrieved on the CPU the last known combination key is used to access the corresponding set of primitives (the mapping between sets and keys is tracked CPU side). The

primitive of the fragment that triggered the unknown combination is added to – or removed from, depending on the facing status – the set. We now have access to the set of primitives of the missing combination, and evaluate the CSG boolean expression on the CPU. This is performed with the ExprTK library [Partow 2012]. The solid/empty status having been determined, the CPU algorithm updates the hash table. This is done in a mirrored copy of the hash table stored CPU side, by performing an insertion using the algorithm described in [García et al. 2011]. The GPU hash table is then updated from its CPU counterpart.

If the table occupancy exceeds a pre-determined percentage (80% in our implementation), we reallocate and rebuild the hash for a lower occupancy (40%). These load ratios provide low ages and fast lookup, which we can afford as our hash tables are small (<1 MB).

4.2 Batch discovery

The drawback of the on-the-fly discovery is that the scene may take some time to converge at startup. In practice this is only a problem for the most complex scenes ; *complex* in terms of the number of combinations to discover (a property not directly related with geometric complexity, see Section 7).

We therefore consider how to quickly pre-discover combinations. The idea is to render three orthogonal views surrounding the object, setting the resolution such as to guarantee all features below a given size (in *mm*) are detected. For each view, the process iterates each depth slab until all configurations are discovered.

Each view is rendered using a variant of the algorithm previously described. In particular, the ray traversal in a pixel is interrupted as soon as an unknown combination is discovered. After insertion, it resumes from this last combination, skipping all fragments and intervals located before since they are already explored. Through the use of depth slabs, discovering all configurations can be done by visiting each fragment only once and with bounded memory. The scene is rasterized once per depth-slabs – culling away primitives which bounding boxes do not intersect the slab.

The overall process is much faster than that of waiting for the standard on–the–fly convergence (Section 7). Therefore, our typical approach is to run batch discovery at startup with a reasonably low resolution, e.g. discovering all combinations occupying a volume bigger than $1mm^3$, and then rely on the on-the-fly discovery during parameter exploration.

This is also used for slicing, using a single orthographic viewpoint which corresponds to viewing the object from below. Slices are rendered as images [Lefebvre 2013] as soon as a depth slab is ready (all combinations discovered), and streamed to the printer. Then, the algorithm proceeds to the next depth slab.

4.3 Avoiding Ambiguities

If two different primitive sets $\mathcal{P}(I) \neq \mathcal{P}(J)$ map to a same key, e.g. $c(I) = c(J)$, there is an ambiguity: upon accessing the hash, we cannot distinguish between a record stored for $c(I)$ or for $c(J)$.

The probability of ambiguities is low. Therefore, when all possible sets are known, removing ambiguities could be simply achieved by regenerating the primitive primary values until no ambiguity remains. The real issue appears with *unknown* sets. During discovery, an unknown primitive set could produce the same combination key as one that is already in the table. In such a case, the algorithm has no way to detect that this set is different from the stored one. The ambiguity could result in the interval being wrongly classified.

To address this issue we introduce *dynamic* secondary values. These are random integers encoded on 23 bits. Their sums define the secondary combination keys (using $n = 23$), which are stored alongside the primary keys in the hash table (Section 3.1.2).

The likelihood that an ambiguity appears on both primary and secondary keys is much lower than that of an ambiguity on the primary keys alone. Furthermore, we change the secondary values *every frame*. Even in the event that an unknown set produces an ambiguity within one frame, the likelihood of this ambiguity surviving across multiple frames quickly vanishes.

Only the primary keys are used to generate addresses within the hash table (Algorithm 2, line 6), but both keys are used to identify whether a record belongs to the lookup key (Algorithm 2, line 8). Thus, the layout of the table is not impacted by a change of secondary values, only the stored secondary keys need an update. This however requires to recompute the sums from the primitives' secondary values, and therefore knowledge of the set of primitives corresponding to each combination. The hash table stored on the GPU does not contain this information: it has no knowledge of the sets themselves, only the keys. Thus, we store in the mirror version of the table on the CPU an additional pointer towards the corresponding set of primitives. The hash table can then be quickly updated at every frame, visiting once each entry and recomputing the sums. This is done in a CPU thread running in parallel to the dexel buffer construction of the first slab.

ALGORITHM 3: Classify intervals for a pixel in a depth slab, with filtering.

```
1  Function classifyIntervalsForPixel(x,y)
2      int64 c, bool status_prev, int32 d_prev;
3      if isFirstSlab then
4          c = 0;
5          status_prev = false;
6          d_prev = near_depth − filter_length;
7      else
8          c, status_prev, d_prev = ReadFromPreviousSlab(x,y);
9      record cur = FirstFragment(x,y);
10     while cur exists do
11         if cur.d ≥ d_prev + filter_length then
12             status = CSGHash[c];
13             if status ≠ status_prev then
14                 # perform application dependent action
15                 # at depth d_prev
16             end
17             status_prev = status;
18             d_prev = cur.d
19         end
20         c = c + Value(cur);
21         cur = NextFragment(cur);
22     end
23     if isLastSlab then
24         status= CSGHash[c];
25         if status ≠ status_prev then
26             # perform application dependent action
27             # at depth d_prev
28         end
29     else
30         WriteForNextSlab(x,y,c,status_prev, d_prev)
31 end
```

4.4 Filtering

Challenging cases of CSG, such as solids nearly overlapping or tiny intersections lead to the apparition of many small intervals along the rays. These are responsible for visual artifacts in screen space methods (z-fighting), and in our case trigger the discovery of additional combinations. In this section we discuss our filtering approach to ignore these spurious, tiny intervals due to numerical instabilities. This improves rendering quality and avoids cluttering the hash table with unnecessary combinations.

Our goal is to only process intervals that have sufficient length, and ignore those which are too small to be meaningful. The pseudo code of the modified algorithm is given in Algorithm 3. It implements a fast filtering, that is performed during the traversal of the fragments, taking into account the subdivision in depth slabs.

The idea is to postpone making a decision on an interval boundary until all fragments within the filter size have been visited. This is done line 11 of Algorithm 3, visiting fragments until one distant enough from the last interval boundary depth is encountered. When a far enough fragment is encountered, *status* (line 12) evaluates the interval located *before cur*. For this reason, a final check is necessary at the very end of the ray (line 23).

The filter size is small: 1 μm in our implementation. We use a constant filter size, which is not ideal for perspective rendering – however this removes most visual artifact and does not incur any inaccuracies during slicing, which uses an orthographic projection.

5 PRODUCING LESS COMBINATIONS

We present two approaches to reduce the number of combinations, and hence reduce the time to discover them.

5.1 Sharing primitive ids

A first approach is to exploit the structure of the CSG tree. In particular, models often contains sets of primitives which are only in union or only in intersection. In such cases, instead of giving the primitives different primary values, we can share the same primary value for all. Let us consider the case of a set of primitives involved in a large union. Normally, for p primitives of arbitrary shapes, up to 2^p combinations are possible. By giving all primitives the same primary value, we reduce this number to o, the maximum number of these primitive overlapping in a same location. For instance, in the simple case of disjoint (non-overlapping) primitives in union, the number of combinations reduces from p to 1 (Figure 3).

This approach works both for unions and intersections, as the (signed) number of times a surface is crossed is sufficient to identify the solid intervals (> 0 for union, equals to o for intersection).

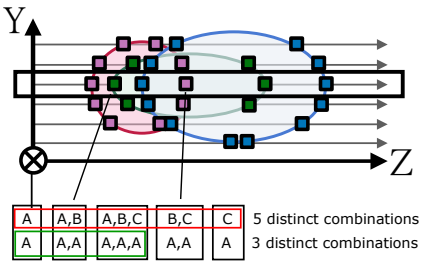
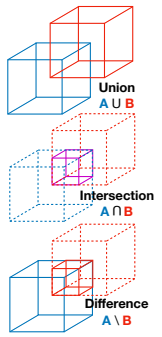


Fig. 3. Sharing primary values.

5.2 Regions of interest



It is often possible to determine that parts of a primitive have no influence on the overall CSG result, e.g. when it is used through intersection and difference operators. For instance, a primitive on the right side of a difference only has an impact where the left side primitive exists.

Thus, we can pre-compute regions of interest (ROI) for each primitive: the regions of space where they *possibly* have an effect. This can be done efficiently, and in a conservative manner, by considering the axis aligned bounding boxes of the primitives. The Figure inset shows the ROI for the standard boolean operators.

This can be used to both reduce the number of inserted fragments, and reduce the number of observed combinations. First, we discard fragments whenever they are located on a view ray that does not intersect the ROI box. Second, if a fragment is located outside the ROI box, we project it along the ray onto the box. The effect is to collapse all intervals outside the ROI box, and thus to remove all the intermediate combinations through filtering (Algorithm 3), see Figure 4. We use the algorithm of Williams et al. [2005] for robust ray–box intersections. To obtain a robust test, it is important that all fragments within a same pixel use the exact same ray definition.

The process is view independent, allowing the use of the same ROI both for batch discovery and on-the-fly hash construction.

6 IMPLEMENTATION

We implement our approach in C++14 and OpengGL 4.2 / GLSL, on both Linux and Windows. The CPU hash table is a one dimensional array with 128-bits entries : the first 64-bits are described in Section 3.1.2, the second half are used as pointers to the set of primitives (Section 4.3). Two threads run on the CPU: one for the rendering loop, and one for secondary keys update (Section 4.3). At each frame, this hides the cost of recomputing secondary sums by running it simultaneously with the first dexel buffer creation.

CPU-GPU communication. Transfers only occur when discovery of combinations is performed (on-the-fly or batch discovery).

Transfers *towards* the GPU occur after an update of the secondary values, and after a change to the hash table (e.g. when inserting new combinations). As insertions into the hash can move already present entries, it is simpler to copy the full hash table (first 64-bits of each entry). We transfer the hash table in between depth slabs, such that newly discovered combinations are immediately available to the next slabs. Typically, the amount of transferred data is smaller than 1MB per frame.

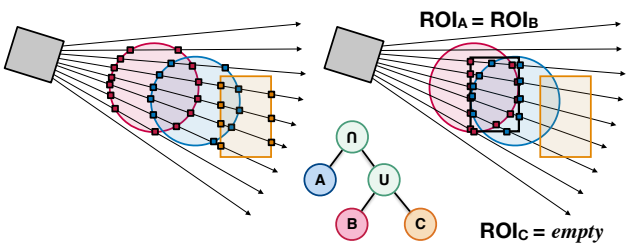


Fig. 4. ROI clipping and projection rules for an intersection (red/blue have same ROI). Many fragments are discarded, while others are projected onto their primitive ROI.

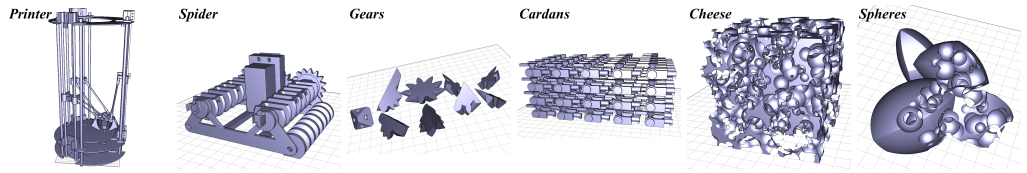


Fig. 5. Set of objects used for performance measurements, in order : *printer, spider, gears, cardans, cheese, spheres.* The view points are representative of the one used in testing.

Transfers *from* the GPU are used to retrieve fragments describing unknown combinations. This requires 1088-bits per-pixel with an unknown combination: 64-bits for the key of the last known combination, 16 times 64-bits for the sixteen recorded fragments (Section 4.1). These informations are stored in a 40KB buffer (fixed size), allowing to record informations for up to 294 pixels per frame. At each frame, this buffer is read back up to the number of depth slabs. Hence, no more that 120KB per frame whit 3 depth slabs. We experimentally chose the number of recorded fragments (16) as well as the minimal downsizing resolution (256x256). Downsizing reduces the likeliness of processing duplicates which occur through spatial coherency. The maximal bound on recorded fragments allows a tradeoff between workload at every frame and how quickly rendering converges towards the final result.

All transfers are performed with `glBindBuffer` and `glMapBuffer`, with the exception of the check for unknown combinations, retrieved with `glGetBufferSubData`. This also provides the amount of data to read when retrieving unknown fragments.

7 DISCUSSIONS AND RESULTS

We tested our method on a variety of models, with varying degrees of complexity in term of number of primitives, number of triangles, and CSG tree size. Statistics on all models are provided in Table 1.

Unless stated otherwise, all tests are done at a resolution of 1024x1024 pixels, with the model fully spanning at least one of the two directions of the rendered image. Typical view points are shown in Figure 5 and 6. Measurements are done on a GeForce Titan Black and an Intel Core i7-4770K (3.5GHz x4 cores). Two threads run on the CPU: one for the rendering loop, and one for secondary keys update (Section 4.3).

We compare our method against that of [Lefebvre 2013]. It builds a full dexel buffer of the scene and uses an integer array to track the in/out status of each primitive (incrementing/decrementing the primitive counter on front/back fragments, inside is > 0). The CSG expression is evaluated as a boolean expression using array entries. It is otherwise comparable in terms of overall rasterization pipeline. We refer to this approach as *array* and to our method as *hcsg* throughout the result section.

All models have at least both unions and differences nodes in their CSG tree. We use three mechanical parts (*printer, spider, gears*) which consist of both imported meshes and tessellated base geometric primitives (cylinder, cones, cubes). The *cardans* example consist only of tessellated geometric primitives (cylinder and boxes). The modeled object is instantiated multiple times and packed into a box – this is typical of 3D printing with powder systems (SLS), maximizing the use of the build volume. The model *cheese* is a large set of random spheres subtracted from a cube. It has a simple CSG tree (requires only 2 integers for *array*), but a high depth complexity. This is also the case for the *spheres* model, but it also has a more complex tree with unions, intersections and difference. Overall, all examples have large numbers of triangles and primitives. A few other examples will be discussed along the section, analyzing specific aspects of the algorithm.

Table 1. Statistics for each models: number of triangles, primitives and unique primary values. Timings (milliseconds) use a resolution of 1024x1024 per viewpoint. Timings are averaged over a full rotation around the object, but for the on-the-fly discovery which uses the viewpoints depicted in Figure 5. Occupancy is the average ratio of used versus unused entries in the hash table, while average/max age indicate the number of required iterations for a lookup.

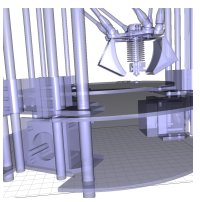
	printer	spider	gears	cardans	cheese	spheres	cubes
nb triangles	489.7k	135.1k	38.5k	614.9k	31337.6k	8008.7k	102.0k
nb primitives	535	295	192	2760	2501	345	8501
nb primary values	101	94	48	1321	2	305	2
on-th-fly (ms)	150	98	240	904	247	4135	324
nb combinations	826	401	1119	6972	21	13451	17
batch 128/1024 (ms)	138/184	107/156	144/198	237/461	174/589	225/569	167/302
nb combinations	659/822	399/400	1018/1099	6937/7263	21/21	9269/11066	20/20
occupancy (%)	53.7	57.5	72	48.5	8.2	40	4
average/max age	1.2 / 5.25	1.4/ 6.3	1.36 / 5.75	1.1 / 5.75	1 / 1	1.1 / 4.8	1 / 1

7.1 Performance measurement

Rendering time : We record the average rendering time for a set of viewpoints (rotating around the object) for both transparent and opaque rendering (respectively Table 2 and 3). We also record the time spent by the GPU for both the dexel buffer rasterization and the processing of the fragment lists.

Our approach is much less sensitive to the topology of the CSG tree: for all scenes where the number of CSG ids increases *array* incurs a large penalty. For instance on *sphere*, it takes 363.5 ms to process the lists versus 6.7 ms for ours (Table 2). With *hcsg*, the time for processing the lists depends solely on the number of fragments, once all combinations are discovered. This confirms the constant time evaluation of the CSG expression. Also note how the average lookup age remains consistently low on all our examples (bottom row of Table 1).

The use of depth slabs – three in these results – reduces the dexel buffer generation cost by up to 60% with transparent rendering (*cubes*), and up to 70% (*cardans, cubes*) with opaque rendering. This is largely due to the reduced cost for sorting the fragments. Another benefit is to allow rendering of scenes that otherwise do not fit memory (Figure 6).



Depth slabs however incur a small CPU overhead. For instance, *hcsg* is slightly less efficient than the array approach for the *printer* model (Table 2). With the measured viewpoint only few pixels are actually covered by primitives, resulting in a small GPU workload revealing the CPU overhead. If we zoom in (see inset), the rendering time becomes again in favor of *hcsg*: opaque 18.5ms (*array*) vs 14.1ms (*hcsg*), transparent 33.1ms (*array*) vs 18.2ms (*hcsg*).

Rendering resolutions and depth slabs : To reveal the effect of depth slabs, we consider two objects of high depth complexity. We render them using transparency (most relevant for slicing) at increasing resolutions.

The first object (Figure 6, top) is a large union of elongated bars, uniformly distributed in three grids along the main axes. It exhibits a small number of CSG combinations and a high depth complexity. The *array* method uses a single integer to track in/out for the union. The second example is a large cube with many smaller cubes subtracted (10k of them). The first obvious benefit of depth slabs is the ability to reach higher resolutions, which stems from the bounded memory. The second benefit is an increased rendering performance using additional depth slabs, after 512^2

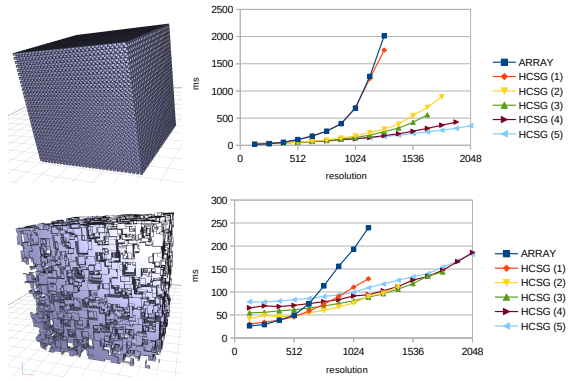
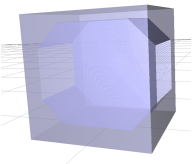


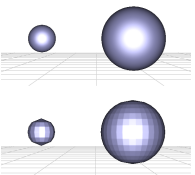
Fig. 6. Rendering time (transparent) as a function of resolution, for *array* and *hcsg* with number of slabs varying from 1 to 5. Missing data is due to the fragments no fitting in memory, see text for details.

(below that the CPU overhead is non negligible). On the second example, after 4 depth slabs the performance decreases again, which is due to CPU setup overheads when starting a depth slab (our current implementation is rather brute force, redoing all the setup work for clipping at each slab).

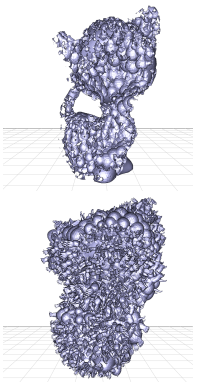
If fragments are clustered within a same depth range, the use of depth slabs has an even larger impact. For instance, in the Figure inset the front/back inner walls each have 150 fragments at the same depth values. The rendering time, dominated by the dexel buffer construction, decreases with the number of depth slab used, from 4.3s for 1 slab, to 0.76s for 3 slabs. Splitting further will not bring any benefit, as our current implementation cannot split sets of fragments having exactly the same depth (see Section 8).



Number of triangles : Many of our examples rely on finely tessellated basic geometric primitives, rendered as triangle meshes. This incurs a non negligible cost through the vertex pipeline, counted in the dexel buffer creation time. For instance, let us consider the cheese example. The number of triangles using the default tesselation (inset, top) is 31337k. If we lower the tessellation (inset, bottom) the number of triangles decreases to 889k. Similarly, the dexel construction time (transparent) decreases from 131ms to 55ms. Of course, this indicates that in this case analytic primitives should be preferred. However, it also reveals that our technique can deal with huge amounts of triangles.



The inset Figure shows 493 instanced individually scaled Armadillo models (345k triangles each) combined to a Kitten model through a difference (top) and a union (bottom), for a total of 170.6 *million* triangles. Transparent rendering takes 284ms per frame while opaque rendering takes 263ms for a 1024x1024 resolution. This shows the benefits in relying on the highly optimized hardware rasterizer for fragment insertions.



This example also reveals the effect of sharing primary value across primitives. Here, the 493 Armadillos are unioned all together and then subtracted from, or unioned with the Kitten. The number of discovered combinations without sharing is 33136, and goes down to respectively 12 and 23. Also, it is worth noting that after all combinations are discovered, the exact same rendering times are obtained: having many more combinations in the table does not impair rendering performance thanks to constant hash lookup times.

Table 2. Timings for transparent rendering (milliseconds) averaged over a full rotation around the object, at 1024x1024. The HCSG approach uses 3 depth slabs. Total time is the rendering time for a frame including synchronization between CPU and GPU. For the *spheres* model some frames fail to render with the ARRAY approach.

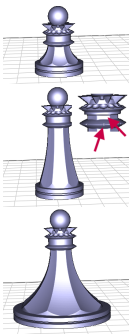
	printer		spider		gears		cardans		cheese		spheres		cubes	
	array	*hcsg*	*array*	*hcsg*	*array*	*hcsg*	*array*	*hcsg*	*array*	*hcsg*	*array*	*hcsg*	*array*	*hcsg*
dexel buffer GPU	4.2	3.9	10.9	7.8	15.1	12.0	27.5	17.0	180.1	110.7	157.2	102.8	132.8	55.2
process lists GPU	5.1	3.7	9.9	3.7	8.7	4.4	363.5	6.7	10.3	9.8	144.2	10.9	120.0	13.8
total	12.6	10.9	24.8	13.9	30.5	18.4	523.2	37.6	194.0	131.5	400.3	116.2	258.9	89.4

Table 3. Timings for opaque rendering (milliseconds) averaged over a full rotation around the object, at 1024x1024. The HCSG approach uses 3 depth slabs. For the *spheres* model some frames fail to render with the ARRAY approach. Total time is the rendering time for a frame including synchronization between CPU and GPU.

	printer		spider		gears		cardans		cheese		spheres		cubes	
	array	*hcsg*	*array*	*hcsg*	*array*	*hcsg*	*array*	*hcsg*	*array*	*hcsg*	*array*	*hcsg*	*array*	*hcsg*
dexel buffer GPU	4.2	3.2	10.9	5.0	15.1	10.2	27.3	9.0	192.7	79.3	153.8	92.3	131.1	40.2
process lists GPU	2.7	2.5	3.4	2.6	6.0	3.8	129.6	3.4	3.0	6.9	98.3	8.2	1.9	6.6
total	7.5	9.1	15.0	10.1	21.8	16.1	158.0	26.3	196.6	98.2	253.1	103.1	137.9	69.5

Number of combinations : The number of combinations discovered by both batch and on-the-fly discovery are provided in Table 1. The time to complete discovery depends on both the number of actual combinations in the model (more combinations require longer exploration), as well as the depth complexity (which impacts dexel buffer construction time). We have seen in Section 5 two ways to reduce the number of discovered combinations. For the models *spider* and *spheres* activating both sharing of values and ROI decreases the number of combinations from, respectively 1000 and 40,9k to 401 and 13.4k (see also Kitten and Armadillo inset above). We discuss in Section 7.3 the tradeoffs when choosing whether to perform batch discovery, on-the-fly discovery or both.

7.2 Parameter exploration



During modeling the user incrementally modifies the CSG tree, e.g. resizing a primitive, adding a new one, switching node operators. When primitive parameters (or types) are changed, it is important for the user to obtain a rapid, accurate feedback. Therefore on-the-fly discovery is activated during parameter editing. For local changes, discovering combinations typically takes only a few frames. For instance, for the chess piece inset, updating required a single frame for the first parameter change (adding 20 combinations) and no update for the second change. In practice this is totally transparent for the user; in fact in most cases no new combinations are discovered. Figure 7 reveals the time required to update the CSGHash table when large changes occurs on a complex model.

Note that after editing the hash may contain combinations that are not present in the scene anymore, but may come back at later stages of editing.

7.3 Balancing batch and on-the-fly discovery

We currently do not have an optimal strategy to balance between batch and on-the-fly discovery. We discuss here some guidelines.

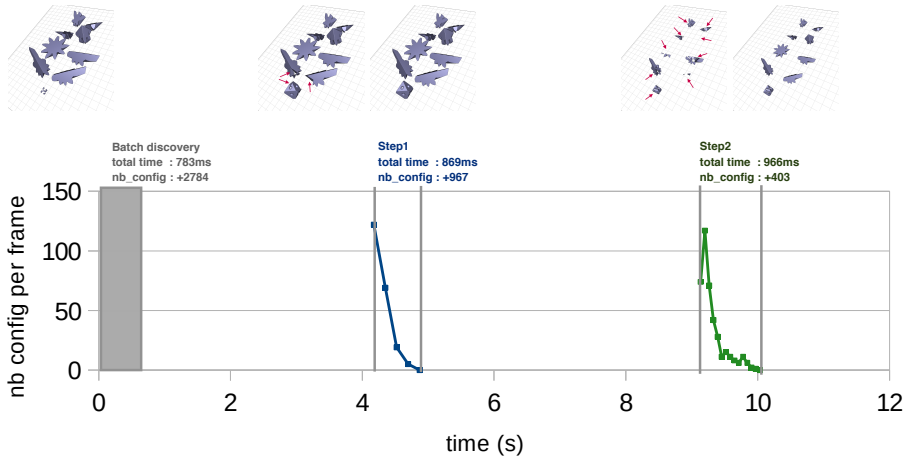
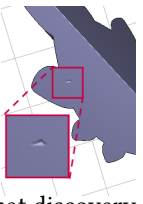


Fig. 7. Editing a parametric model and time required for on-the-fly discovery after user edits. Arrows are pointing visual artifacts due to missing combinations. The correct rendering is recovered in less than a second.

Batch discovery outperforms on-the-fly discovery when the model exhibits many combinations (*cardans*, x2) or has a high depth complexity (*spheres*, x7). Therefore, batch discovery is best used upon loading a new model. Batch discovery has the added advantage of being independent from the viewpoint. Thus, if a sufficient resolution is used, most combinations – if not all – are quickly discovered.



However, we observe that during modeling it is typical for users to zoom in closely to inspect their designs, and it would be difficult to guarantee that no defect remains (while this is possible for slicing, where sampling resolution is known and fixed). Any missed combination may become visible in the rendering: see inset, zoom on *gears* after batch discovery at 128^2, that is $2mm^3$ precision. Such small defects are quickly detected and corrected by on-the-fly discovery. We therefore recommend to run batch discovery once at startup, using a relatively low resolution, and to constantly perform on-the-fly discovery. This can be balanced with GPU load, such that discovery is performed aggressively during idle frames and paused when the application requires precise and responsive interaction, such as painting colors along the model.

8 LIMITATIONS AND FUTURE WORK

Our implementation has a number of limitations. The depth slabs cannot split a set of fragments which all have exactly the same depth. Thus, it is possible to construct pathological cases where memory bounds are not enforced. However, this can be resolved by slightly changing the algorithm to only record the first k fragments, in the manner of a k-buffer [Vasilakis et al. 2015].

The tradeoff between batch / on-the-fly discovery of combinations has to be further explored: the best choice depends on the model itself, so a possible improvement would be to perform batch discovery at very low res, progressively increasing within a given time budget. It may also be possible to perform a conservative analysis of the scene geometry to pre-fill the CSG table, as opposed to always rely on rendering.

Our approach is casting a different view on the problem of CSG expression evaluation during rendering. We hope this will inspire future work to use hashing for rendering and modeling tasks. We are also curious as to whether our technique can benefit off-line ray-tracing, where tradeoffs are somewhat different.

ACKNOWLEDGEMENTS

This work was funded in part by the ERC grant ShapeForge StG-2012-307877.

REFERENCES

Dan A. Alcantara, Andrei Sharf, Fatemeh Abbasinejad, Shubhabrata Sengupta, Michael Mitzenmacher, John D. Owens, and Nina Amenta. 2009. Real-Time Parallel Hashing on the GPU. *ACM Transactions on Graphics (Proceedings of ACM SIGGRAPH Asia 2009)* 28, 5 (2009).

Hichem Barki, Gael Guennebaud, and Sebti Foufou. 2015. Exact, robust, and efficient regularized Booleans on general 3D meshes. *Computers & Mathematics with Applications* 70, 6 (2015), 1235 – 1254. https://doi.org/10.1016/j.camwa.2015.06.016

I. Dolenc, A.and Mäkelä. 1994. Slicing procedures for layered manufacturing techniques. *Computer-Aided Design* 26, 2 (1994), 119 – 126.

Matthijs Douze, Jean-Sébastien Franco, and Bruno Raffin. 2015. *QuickCSG: Arbitrary and Faster Boolean Combinations of N Solids*. Research Report RR-8687. Inria - Research Centre Grenoble – Rhône-Alpes ; INRIA. https://hal.inria.fr/hal-01121419

Elmar Eisemann and Xavier Décoret. 2008. Single-pass GPU Solid Voxelization for Real-time Applications. In *Proceedings of Graphics Interface 2008 (GI '08)*. Canadian Information Processing Society, Toronto, Ont., Canada, Canada, 73–80. http://dl.acm.org/citation.cfm?id=1375714.1375728

Fansen F. Epstein D. and Rossignac J. 1989. *Z-Buffer Rendering from CSG: The Trickle Algorithm*. Research Report RC 15182. IBM.

F. R. Feito, C. J. Ogayar, R. J. Segura, and M. L. Rivero. 2013. Fast and Accurate Evaluation of Regularized Boolean Operations on Triangulated Solids. *Computer Aided Design* 45, 3 (March 2013), 705–716. https://doi.org/10.1016/j.cad.2012.11.004

Ismael García, Sylvain Lefebvre, Samuel Hornus, and Anass Lasram. 2011. Coherent Parallel Hashing. *ACM Transactions on Graphics* 30, 6 (Dec. 2011). https://hal.inria.fr/inria-00624777

Andrew S. Glassner (Ed.). 1989. *An Introduction to Ray Tracing*. Academic Press Ltd., London, UK, UK.

A. S. Glassner, J. Arvo, R. L. Cook, E. Haines, P. Hanrahan, P. Heckbert, and D. B. Kirk. 1989. *Introduction to Ray Tracing*. Academic Press.

Jack Goldfeather, Steven Monar, Greg Turk, and Henry Fuchs. 1989. Near Real-Time CSG Rendering Using Tree Normalization and Geometric Pruning. *IEEE Comput. Graph. Appl.* 9, 3 (May 1989), 20–28. https://doi.org/10.1109/38.28107

John Hable and Jarek Rossignac. 2005. Blister: GPU-based Rendering of Boolean Combinations of Free-form Triangulated Shapes. *ACM Trans. Graph.* 24, 3 (July 2005), 1024–1031. https://doi.org/10.1145/1073204.1073306

Tim Van Hook. 1986. Real-Time Shaded NC Milling Display. In *Proceedings of SIGGRAPH*. 15–20.

D. Kalra and A. H. Barr. 1989. Guaranteed Ray Intersections with Implicit Surfaces. *SIGGRAPH Comput. Graph.* 23, 3 (July 1989), 297–306. https://doi.org/10.1145/74334.74364

A. Kensler. 2006. Ray tracing CSG objects using single hit intersections. (2006). http://xrt.wdfiles.com/local--files/doc%3Acsg/CSG.pdf

Sylvain Lefebvre. 2013. IceSL: A GPU Accelerated CSG Modeler and Slicer. In *AEFA'13, 18th European Forum on Additive Manufacturing*. Paris, France. https://inriav3-preprod.archives-ouvertes.fr/hal-00926861

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

Makerbot. 2013. Customizer. (2013). https://www.thingiverse.com/customizer

M. Maule, J.L.D. Comba, R. Torchelsen, and R. Bastos. 2012. Memory-Efficient Order-Independent Transparency with Dynamic Fragment Buffer. In *Proc. 25th Conference on Graphics, Patterns and Images (SIBGRAPI)*. IEEE, 134–141.

Marilena Maule, João Luiz Dihl Comba, Rafael P. Torchelsen, and Rui Bastos. 2011. A survey of raster-based transparency techniques. *Computers & Graphics* 35, 6 (2011), 1023–1034.

Timothy S. Newman and Hong Yi. 2006. A survey of the marching cubes algorithm. *Computers & Graphics* 30, 5 (2006), 854 – 879. https://doi.org/10.1016/j.cag.2006.07.021

C.J. Ogayar-Anguita, A.L. GarcÃŋa-Fernandez, F.R. Feito-Higueruela, and R.J. Segura-Sanchez. 2015. Deferred boundary evaluation of complex CSG models. *Advances in Engineering Software* 85 (2015), 51 – 60. https://doi.org/10.1016/j.advengsoft.2015.03.003

Arash Partow. 2012. Exprtk. (2012). http://partow.net/programming/exprtk/

Alexander Pasko, Oleg Fryazinov, Turlif Vilbrandt, Pierre-Alain Fayolle, and Valery Adzhiev. 2011. Procedural Function-Based Modelling of Volumetric Microstructures. *Graphical Models* 73, 5 (2011), 165–181. https://doi.org/10.1016/j.gmod.2011.03.001

A. Ricci. 1973. A constructive geometry for computer graphics. *Comput. J.* 16, 2 (1973), 157–160. https://doi.org/10.1093/comjnl/16.2.157

Fabiano Romeiro, Luiz Velho, and Luiz Henrique de Figueiredo. 2008. Scalable GPU rendering of CSG models. *Computers & Graphics* 32, 5 (2008), 526 – 539. https://doi.org/10.1016/j.cag.2008.06.002

Jarek Rossignac. 2011. Ordered Boolean List (OBL): Reducing the Footprint for Evaluating Boolean Expressions. *IEEE Transactions on Visualization and Computer Graphics* 17, 9 (Sept. 2011), 1337–1351. https://doi.org/10.1109/TVCG.2010.232

Jarek R. Rossignac. 1998. Blist: A Boolean list formulation of CSG trees. (1998).

Shane Ryoo, Christopher I. Rodrigues, Sara S. Baghsorkhi, Sam S. Stone, David B. Kirk, and Wen-mei W. Hwu. 2008. Optimization Principles and Application Performance Evaluation of a Multithreaded GPU Using CUDA. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '08)*. ACM, 73–82.

Ryan Schmidt and Tyson Brochu. 2016. Adaptive Mesh Booleans. In *Whitepaper arXiv.org*.

Maria Shugrina, Ariel Shamir, and Wojciech Matusik. 2015. Fab Forms: Customizable Objects for Fabrication with Validity and Geometry Caching. *ACM Transactions on Graphics* 34, 4 (2015).

Nigel Stewart, Geoff Leach, and Sabu John. 1998. An Improved Z-buffer CSG Rendering Algorithm. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Workshop on Graphics Hardware (HWWS '98)*. ACM, New York, NY, USA, 25–30. https://doi.org/10.1145/285305.285308

Nicolas Thibieroz. 2011. Order-Independent Transparency using Per-Pixel Linked Lists. In *GPU Pro 2*, Wolfgang Engel (Ed.). A K Peters, 409–431.

D. Ulyanov, D. Bogolepov, and V. Turlapov. 2017. Interactive vizualization of constructive solid geometry scenes on graphic processors. *Programming and Computer Software* 43, 4 (01 Jul 2017), 258–267.

A. A. Vasilakis, G. Papaioannou, and I. Fudos. 2015. k^+ -buffer: An Efficient, Memory-Friendly and Dynamic k -buffer Framework. *IEEE Transactions on Visualization and Computer Graphics* 21, 6 (June 2015), 688–700.

Kiril Vidimče, Szu-Po Wang, Jonathan Ragan-Kelley, and Wojciech Matusik. 2013. OpenFab: A Programmable Pipeline for Multi-Material Fabrication. *ACM Trans. Graph.* 32, 4 (2013), 1. https://doi.org/10.1145/2461912.2461993

Charlie C.L. Wang. 2011. Approximate Boolean Operations on Large Polyhedral Solids with Partial Mesh Reconstruction. *IEEE Transactions on Visualization and Computer Graphics* 17, 6 (2011), 836–849. https://doi.org/10.1109/TVCG.2010.106

Amy Williams, Steve Barrus, R. Keith Morley, and Peter Shirley. 2005. An Efficient and Robust Ray-box Intersection Algorithm. In *ACM SIGGRAPH 2005 Courses (SIGGRAPH '05)*. ACM, New York, NY, USA, Article 9. https://doi.org/10.1145/1198555.1198748

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

Qingnan Zhou, Eitan Grinspun, Denis Zorin, and Alec Jacobson. 2016. Mesh Arrangements for Solid Geometry. *ACM Trans. Graph.* 35, 4, Article 39 (July 2016), 15 pages. https://doi.org/10.1145/2897824.2925901