A Runtime Cache for Interactive Procedural Modeling

Tim Reiner [1] Sylvain Lefebvre [2] Lorenz Diener [1] Ismael García [3] Bruno Jobard [2] Carsten Dachsbacher [1]

[1] *Computer Graphics Group*
Karlsruhe Institute of Technology

[2] *ALICE Team*
INRIA Nancy Grand-Est

[3] *Geometry and Graphics Group*
Universitat de Girona

Abstract

We present an efficient runtime cache to accelerate the display of procedurally displaced and textured implicit surfaces, exploiting spatio-temporal coherence between consecutive frames. We cache evaluations of implicit textures covering a conceptually infinite space. Rotating objects, zooming onto surfaces, and locally deforming shapes now requires minor cache updates per frame and benefits from mostly cached values, avoiding expensive re-evaluations. A novel parallel hashing scheme supports arbitrarily large data records and allows for an automated deletion policy: new information may evict information no longer required from the cache, resulting in an efficient usage. This sets our solution apart from previous caching techniques, which do not dynamically adapt to view changes and interactive shape modifications. We provide a thorough analysis on cache behavior for different procedural noise functions to displace implicit base shapes, during typical modeling operations.

Keywords: interactive shape modeling, implicit surface rendering, procedural textures, runtime cache, parallel hashing

1. Introduction

One of the core qualities of procedural modeling is to enable programmers and artists to pass the modeling task of creating rich *visual complexity* on to the computer [7]. Ideally, this results in realistic scenes, naturally unfolding up to unlimited detail, starting from very small descriptions.

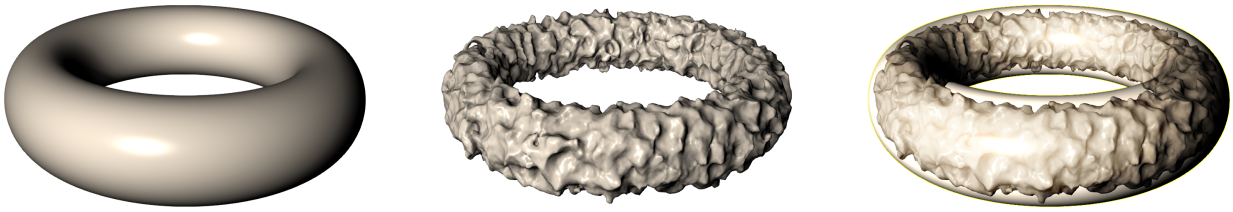
This paper deals with both implicit shape and texture descriptions, which are of particular interest: unlike explicit descriptions, they can be evaluated *on demand*, and are naturally amenable to modeling techniques, such as constructive solid geometry and blending, suited for both handcrafted, man-made objects, and organic, soft objects.

However, implicit surface descriptions have two main drawbacks: first, the outcome of a shape is not as easy to control as, for instance, it is for explicit meshes; yet there is a large body of previous research, proposing manifold modeling environments to regain control, as the next section shows. Second, it is more difficult to render implicitly described scenes in real-time than it is for explicit representations.

With the advent of parallel processing, implicit surfaces become viable for direct interactive rendering. In this paper, we use *sphere tracing* [10] to render shapes described by *signed distance functions* (SDF) efficiently. Figure 1(a) shows a torus, which, together with other implicit base shapes, can be easily described with an SDF and directly rendered on the GPU with sphere tracing. To move on to more complex, more interesting shapes, we apply procedural displacement textures (see Figure 1(b)), as visual complexity requires these fine-scale details. Most recent and powerful procedural texturing methods are based on spot noises, obtained by summing a large number of kernels positioned randomly in space [12]. Alas, evaluating them is very expensive, and may not happen in real-time anymore.

For this reason, we propose a technique for caching evaluations of implicit solid textures. Instead of relying on a fixed, spatially limited grid, our cache adapts to the view conditions and uses available memory where it is most useful. Since simple base shapes are sufficiently fast to evaluate, we cache only the texture part within a coarse shell, as Figure 1(c) shows. The whole process happens lazily and seamlessly during rendering, which couples smoothly into the on-demand evaluation of implicit procedural descriptions. This lets the user model shapes without invalidating the cache, while the texture details are efficiently rendered.

Contributions. We propose a novel cache mechanism equivalent to a sparse grid spanning the entire, infinite 3D texture space, in order to model complex procedu-



(a) Implicit base shape, (b) with textured surface, (c) base shape acting as a coarse shell.

Figure 1: Our notion of *texture* throughout the paper. We assume that distance functions describing implicit base shapes are fast to evaluate. Only the texture part, by contrast, is expensive, and will be cached. The base shape further acts as a coarse shell, and texture is cached only within the shell. We do not narrow down to color textures; more importantly, we grasp *surface texture* in its literal sense, and think of *texture* as mainly procedural displacements.

ral shapes interactively. The cache is inspired by recent real-time hashing schemes which we revisit to implement a cache mechanism. Key-data pairs are inserted in the hash, new data evicting old data, without having to explicitly implement a least-recently-used policy. Our hashing scheme efficiently allocates blocks of data during insertions to exploit the native tri-linear interpolation available on graphics hardware. Our scheme is very easy to use and implement: it does not require complex communication between CPU and GPU and fits entirely within OpenGL 4.2 shaders. The technical implementation has no impact on the procedural functions, i.e., no modifications to implicit descriptions are required. We also provide a thorough analysis of the cache behavior for a variety of modeling examples.

2. Previous work

Implicit surfaces are amenable to ray tracing, but in the recent past—and without programmable graphics hardware—this was far apart from being feasible in real-time. Instead, one had to resort to indirect visualization techniques for rendering, such as variants of Marching Cubes [15], with intermediate, explicit representations; interesting level-of-detail (LOD) techniques can be found in [11]. Specialized techniques have also been proposed: Loop and Blinn [14] achieved real-time rendering on the GPU by assembling fourth-order piecewise algebraic surfaces, introduced by Sederberg [20].

Hart presented sphere tracing [10] to render distance functions efficiently. *Distance fields* can be used to store explicit distance values, which are then simply queried instead of evaluated. However, they consume a large amount of memory; they can be stored hierarchically using *adaptive distance fields* (ADF) [8], significantly reducing memory consumption. Rendering ADFs can be performed efficiently on the GPU, as proposed by Bastos and Celes [4]: the field is stored into a perfect spatial hash [13], a static data structure efficiently accessed by the GPU (see also Section 3). However, and contrary to our approach, their data structure is fixed and suits only for interactive rendering, not modeling, as it cannot adapt to interactive surface deformations.

Schmidt et al. presented ShapeShop [19], an interactive, sketch-based modeling environment for implicit surfaces. They achieve interactivity by caching the scalar fields in dense grids [18]: *caching nodes* are manually placed into the well-known BlobTree [21]. This significantly increases rendering performance (by an order of magnitude at that time) at the expense of a large memory consumption. Our schemes address this issue—while providing similar functionality—further exploiting that a given viewpoint uses only a small fraction of the total data.

Reiner et al. [17] recently presented a direct visualization approach for an interactive implicit modeling environment based on signed distance functions. As they rely on purely analytical forms, they support high-frequency detail, but complex textures can no longer be evaluated interactively. Our cache provides significant improvements to this approach.

Finally, the Gigavoxel framework [6] could be used for caching implicit distance fields. Gigavoxel is a hierarchical data structure which can be efficiently traversed by rays. Rays stop when encountering missing data, which is produced before tracing resumes. In contrast, our scheme exploits the fact that the data is implicit, seamlessly balancing between computation and memory storage. In particular we can render frames which would not fit entirely within the cache. Contrary to Gigavoxel, our scheme does not require maintenance

of a pointer-based hierarchy, resulting in a simpler implementation. Our scheme also exploits the hash eviction mechanism (see Section 3) to avoid implementing an explicit least-recently-used policy, removing entirely the need for tracking data usage at every frame.

3. Background on parallel spatial hashing

Our cache scheme is inspired by recent work on parallel spatial hashing. Here, we present the basis of these approaches, which is a core element of our algorithm.

Spatial hashing schemes work with key-data pairs. A key is a unique identifier for the data, typically the x, y, z coordinates. Keys are used to insert and retrieve data from the *hash table*, which stores all key-data pairs. The hash algorithm computes the location within the hash table from the key.

Lefebvre and Hoppe [13] proposed a hashing scheme for computer graphics for storing texture data around a surface. While access to the data was efficient—requiring two memory accesses and one addition—the construction process was sequential and much slower, requiring in the order of minutes.

Alcantara et al. [2] proposed a parallel hash construction inspired by Cuckoo hashing. Keys can be inserted in a fixed number of locations (typically, four) within the table. Each thread inserts a key, evicting the previously stored key. The thread is then responsible for inserting the evicted key to its next location. This process continues until the last evicted key finds an empty location. All threads insert in parallel and compete for empty slots. Using four locations, the process terminates with high probability if the table is less than 90% full. While construction is much faster, access now requires to test all possible locations for the key. This however remains very fast on modern hardware.

García et al. [9] improved the construction and access performance by relying on a Robin Hood strategy. Keys now have a full sequence of possible insertion locations. The *age* of a key is the position along the sequence which was used to insert the key. During insertion, keys can only be evicted by older keys: young keys make room for older ones. This allows to fill the table at a higher density, without hindering access performance. In addition, the authors propose an insertion sequence which preserves memory coherence, greatly improving performance for coherent access patterns.

Our scheme is based on both the latest work of Alcantara et al. [2, 3] and García et al. [9], but we modify these in several ways: first, our cache allows for keys to stream in and out of the hash at every frame during rendering, keys with new data evicting keys with older data. Second, contrary to these schemes, our universe is unbounded—coordinate records are not limited to 32 or 64 bits. Finally, we introduce an allocation/indirection scheme to avoid having to move data around the table when keys are evicted. This reduces the required bandwidth significantly when large data records are used.

4. Overview

Our implicit surfaces are defined as the zero-crossing of real-valued scalar fields. We render the surfaces using sphere tracing [10], and assume that the scalar fields are obtained from functions which are Lipschitz continuous: there exists a constant Λ so that $|x - y| < \delta \Rightarrow |f(x) - f(y)| < \Lambda\delta$.

We note f the function defining the surface, and t the function defining the texture. The final shape s is obtained as $s(\mathbf{p}) = f(\mathbf{p}) + \alpha\, t(\mathrm{T}\,\mathbf{p})$, with $\mathbf{p}$ a point in space, α a weight, and T a transformation from world to texture space. The texture function has a bounded influence: There exists M so that for all $\mathbf{p}, |t(\mathbf{p})| \leq M$. Therefore we do not evaluate t when sufficiently far away from the surface since it has no influence. Throughout this paper, we assume that t is much more expensive to compute than f.

The sphere tracing pseudo code for a single pixel is:

```
1   step = Infinity
2   p    = Eye
3   l    = 0
4   while (step > Epsilon) {
5     if ( outside_coarse_shell ) {
6       eval = f(p) // evaluate distance to coarse shell
7     } else {
8       // inside coarse shell, evaluate also texture
9       eval = f(p) + α * cached_t(T * p)
10    }
11    // march evaluated distance along ray
12    step = eval * StepSize
13    p    = p + step * RayDirection
14    l    = l + step
15    if (l > FarClip) {
16      pixelColor = BkgColor
17    }
18  }
19  pixelColor = shade(p)
```

In this algorithm, `Eye` is the current eye position, `RayDirection` is the direction of the ray from the eye through the pixel center, `Epsilon` controls the accuracy of the ray-surface intersection, `StepSize` avoids over-stepping and depends on the Lipschitz property of the function, `FarClip` is the maximum distance after which we stop marching and `BkgColor` is the background screen color. Finally, `cached_t` is the cached texture function. If the cache has sufficient resolution,

this is equivalent to calling `t(p)` directly, but results in better performance through reuse of previous computations.

Our goal is to cache the expensive evaluation of t so that the overall rendering time is decreased. Throughout modeling the user may modify f, or may change α and T: these operations do not invalidate the cache. If the user changes the overall procedural description of t, however, the cache has to be discarded.

Each time `cached_t` is called, our cache is queried for data. The cache is equivalent to a sparse grid covering the entire texture space, storing information in its cells. This produces a discretized, tri-linearly interpolated version of t (Section 6.1). The cache cells have to be small enough to properly capture the details of t. In our implementation, the user has to specify the cell size manually; we use $(1/192)^3$ in all our examples.

For each query, we check whether data is available in the cache. If data is available, we return it directly. If not, we *compute* the new data and insert it in the cache. In our implementation, we insert only the last cache miss encountered along a ray; this limits the number of insertions to one per pixel at most. At the next frame, this data will be available with high probability and the cache miss is unlikely to occur again.

Depending upon the current viewing condition, the inserted data point may evict older data, or fail to insert. Note that even if a data point fails to insert, it will have another chance at the next frame if still visible. We exploit this observation to obtain a simpler, more efficient algorithm.

From the rendering point of view, this entire process is transparent. Thanks to latest advances in graphics hardware, we can implement the entire process in OpenGL 4.2 shaders, avoiding slow CPU/GPU communication.

5. Cache mechanism

Rendering at each frame is performed in three steps:

1. **Raymarch:** All rays are traced. For each pixel the intersection (if any) is computed, as well as the *last* cache miss position (if any). During ray marching the cache is read only.
2. **Insert:** A new slot is reserved in the hash table for each pixel which has encountered a cache miss.
3. **Produce:** The data for each newly inserted key is evaluated and stored in the cache.

Our cache data structure is organized around a hash. The hash stores information about each key: the age of the key, the key's 3D coordinates, and the data associated with the key. In practice, this information is stored in different tables: `C` stores the spatial coordinates of the keys, `D` stores the data records, and `H` stores the age of the keys as well as the location of their data in the other two tables. All tables have the same size and are 3D arrays. The cumulative size of the tables correspond to the amount of memory allocated for caching. This is independent from the virtual resolution of the sparse cache grid in space.

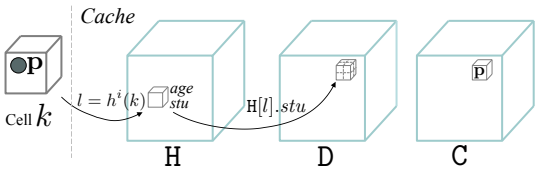


Figure 2: Our cache data structure. A point **p** is enclosed within cell k in texture space. The cell is stored at age i within `H`, its data and coordinates respectively at `D`[`H`[l].*stu*] and `C`[`H`[l].*stu*] with location $l = h^i(k)$.

Given a key k at age i, the hash computes a location $l = h^i(k)$. The key information can be found at `H`[l]. The key spatial coordinates are at `C`[`H`[l].*stu*] and its data at `D`[`H`[l].*stu*], with `stu` the location of the records for the key in `C` and `D`. This indirection lets us move keys in `H` without having to update `C` and `D`, which is a unique advantage of our hashing scheme. This is summarized in Figure 2.

The pseudo code for `cached_t` is given below:

```
1  cached_t( p, H, C, D ) {
2    k = texture2cell ( p )
3    d = access( p, H,C,D )
4    if ( d == null ) {
5      // cache fault, record for insertion
6      LastCacheMiss = p
7      // compute the value instead
8      return t( p )
9    } else {
10     // data is in cache!
11     return d
12   }
13 }
```

The function `texture2cell` computes which cell of the cache grid encloses **p**. It translates the continuous positions into coordinates within the cache grid. This function defines the cache resolution—in fact *resolution* in this context refers to the size of a cache cell in space since the cache's spatial extent is unbounded.

`LastCacheMiss` records the position of the last cache miss. It is later used to insert data into the hash.

5.1. Accessing keys

Our hash follows the open addressing scheme of García et al.: each key k is associated with a sequence of possible insertion locations noted $h^i(k)$. The insertion algorithm (discussed Section 5.2) ensures that the maximum insertion age is bounded: if a key is present in the table, then it is at location $h^i(k)$ with $i < A$. A is called the *maximum age* for the table. In our implementation, it is arbitrarily fixed to a value of 4.

Retrieving a key simply amounts to walking along the sequence $h^i(k)$, until either the key is found, or the maximum age is reached. The pseudo code for retrieving a key is given below:

```
1  access( p, H, C, D ) {
2    k = texture2cell ( p )
3    for (int i = 0 ; i < A ; i ++) {
4      p_stored = C[ h(k,i) ]
5      if ( texture2cell ( p_stored ) == k ) {
6        return D[ H[h(k,i)]. stu ]
7      }
8    }
9    // key not in hash
10   return null ;
11 }
```

The hash function h is the coherent hash function of García et al., which is a random translation added to k and determined by i:

$$h^i(k) = (k + O[i]) \mod N,$$

where O is a precomputed table of random offsets and N the size of the hash table. All these quantities are 3D vectors, and vector operations are performed independently on x, y, z.

The coherent hash function significantly improves access performance over randomizing functions [9]. Note also that keys with a younger age are faster to retrieve.

5.2. Inserting keys

The heart of our method is the insertion algorithm. Our goal is quite different from a typical hashing scheme, and the following paragraphs enumerate these differences.

Cache policy. While the hash table is empty at the start of the application, it quickly fills up with new data at each frame. Once full, new data should still be accepted but will have to erase some older data (see Figure 3). In addition, we would like the histogram of ages to be favorably biased toward recently inserted data: it is more likely to be useful for the next viewpoints, and younger ages imply faster access.

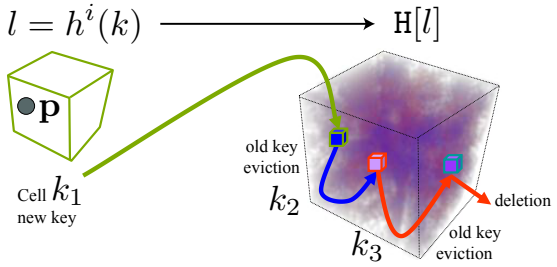


Figure 3: Cache insertion: point **p** generated a cache miss. It is enclosed within cell k_1. The cell is inserted at age 1 within H. The slot is already occupied by k_2, which is evicted. This generates a second eviction, and the last key k_3 cannot be reinserted: it is hence deleted from the cache.

Existing spatial hashing schemes are meant to build a hash table from a predetermined set of keys, seeking to fill the hash table entirely (García et al. report fill rates of up to 99%). Once the table is full, *all* further insertions would fail. This implies that our cache would never be updated beyond this point. A possible approach would be to implement an explicit least-recently-used strategy, keeping track of which keys are accessed every frame. However, this would require a lot of bandwidth during rendering: each thread should write in the hash at every access. Instead, we would like to have the access to be very fast, so that the cache is overall beneficial.

Our first intuition was to exploit a modified Robin Hood policy, comparing the ages of keys. However, a policy inspired by Cuckoo hashing provides a simpler and better answer. Similarly to Cuckoo hashing, we blindly evict keys. A new key therefore always evicts a key already in the hash. The evicted key is tested for its next location, but is only allowed to evict a key which was also already in the hash (i.e., not inserted within the current frame). We limit the number of iterations, so that only a few successful evictions can occur before the key falls out of the cache. This process is illustrated in Figure 3. In addition, any key reaching the maximum age automatically falls out of the cache. The result is an increased probability of falling out of the cache for older keys, as revealed by the histograms in Figure 4.

Data storage. Another important technical issue is that existing schemes store the data along the keys. This is convenient, since keys and data can be stored simultaneously in 64 bits words: A *single* atomic operation *both* evicts and inserts the key-data records (`atomicExchange` for Alcanatara et al., `atomicMax` for García et al.). These algorithms are designed around

this idea: without atomic operations, a value should first be read, tested, and only then would the hash table be written. In-between the read and write, another thread could change the table, producing inconsistencies. Atomic operations avoid this issue by performing the entire process within a single atomic call.

As we shall see in Section 6.1, efficient tri-linear interpolation requires large data records. This data is too large for being stored and moved along with the key, and must be kept in a separate table. Keeping both tables synchronized is similarly difficult: in-between writes to both tables, another thread may change the first table, again generating inconsistencies.

We introduce a new hash algorithm which avoids having to move data records around, while ensuring that the data structure is consistent at all times. In our scheme, the data of a key k remains stored at the same location in table `D`, as long as the key is in the hash and regardless of the number of times the key moves within `H` due to evictions. This location, noted `stu`, is stored along the age of the key in `H`.

An additional benefit is that our keys are no longer limited to coordinates fitting within 32 bits: we can store the coordinates in table `C`, again at the `stu` location. We thus can rely on 32 bits coordinate *triples* (96 bits): our cache covers the full unbounded 3D space.

Duplicated keys. Neighboring pixels are very likely to generate similar cache misses, and will therefore try to insert the same keys simultaneously into the hash. Our cache insertion is robust to these cases and ensures that keys stored in the hash are unique. This mechanism is discussed in more detail throughout the next sections.

Algorithm. Next, we give the entire pseudo code of our algorithm for inserting a single key, and provide a line-by-line walkthrough.

The algorithm takes the following parameters: `p` is the coordinates of the insertion point, `d` the data to be stored, `H` is the hash table, `C` stores the x, y, z coordinates of the inserted keys, and `D` stores the data. `H` is a table of 32 bits storing the age (4 bits) and the data coordinates `stu` (24 bits). In our implementation the remaining 4 bits are unused.

All new keys detected within the frame are inserted in parallel, each thread running the insertion algorithm. New keys in the table are flagged by having their `stu` coordinate set to null. After insertion, the algorithm returns the location of the key in `H` along with the `stu` coordinates for its data in `C` and `D`. Note that the `stu` record of the key in `H` is *not* updated immediately. All insertions for the frame must terminate first. This synchronization ensures that if other insertion threads are still running, they still detect the key as new (`stu == null`).

```
1  insert (p,d,H,C,D) {
2    inserted_l      = null
3    inserted_stu    = null
4    k               = texture2cell ( p )
5    key_info.age    = 1
6    key_info.stu    = null
7    iter            = 0
8    while( iter++ < A ) {
9      l     = h( k, key_info.age )
10     prev  = H[ l ]
11     if ( prev == empty || prev.stu != null ) {
12       // evict the key
13       read = atomicCAS( & H[ l ] , prev, key_info );
14       if ( read == prev ) {
15         // atomic insertion succeeded
16         if ( inserted_l == null) {
17           inserted_l = l;
18         }
19         if ( read == empty ) {
20           // this was an empty slot
21           inserted_stu = l;
22           // we found an empty slot, return
23           break
24         } else {
25           // a key was evicted, try to re-insert
26           k = texture2cell ( C[ prev.stu ] )
27           key_info.age    = prev.age
28           key_info.stu    = prev.stu
29           inserted_stu    = prev.stu
30         }
31       } else {
32         // conflict during write
33         if (key_info.stu == null) {
34           break // exit if inserting a new key
35         }
36       }
37     } else {
38       // a new key concurrently inserted is encountered
39       if (key_info.stu == null) {
40         break // exit if inserting a new key
41       }
42     }
43     // try next location
44     key_info.age ++
45     if (key_info.age >= A){
46       break
47     }
48   }
49   // done, return insertion location and stu
50   return ( inserted_l , inserted_stu )
51 }
```

We now describe each step of the algorithm. Please keep in mind that any key which fails to insert, or falls out of the cache, will have another chance at the next frame if it is still used. Therefore, one of our design principles is to reject keys aggressively to keep the algorithm simple.

Line 2–3: `inserted_l` records where the new key will be inserted in `H`. `inserted_stu` records which slot will be allocated for the data of the new key in `C` and `D`. It will be known only after the algorithm unfolds.

Line 4–6: `k` is the key: the coordinates of the cache cell enclosing **p**. Since the insertion request was generated on a cache miss, `k` is *new*: it is not in the hash when the insertions start. `key_info` is a 32 bits bit-field storing the age of the key being inserted and its `stu` coordinates. `stu` is null since it is unknown for the new key.

Line 7–8: The insertion algorithm will run for only a maximum number of iterations. We set this number of iterations to the maximum age A. Any key still not inserted at the end falls out of the cache (deletion). This always happens once the cache is full: an old key must be deleted to make room for a new key.

Line 9: An insertion location is computed from the key coordinates and its age. The age is incremented every iteration, line 44.

Line 10: The hash table is read at location `l`, and the result is stored in `prev`.

Line 11: We test whether the key can be inserted at the current location. It is the case if the current slot is empty, or if it is occupied by a key that can be evicted. We do *not* allow the eviction of empty keys (`stu == null`). The main reason is that unless an empty slot is found, we will give the `stu` of the last evicted key to the new key. Therefore, only keys with a valid `stu` can be safely evicted. Section 5.3 discusses this in more details.

Line 13–14: The key is tentatively written in `H` line 13, using an atomic operation. The atomic compare and swap (CAS) only writes `key_info` if the value in `H` is still `prev`. The value read by `atomicCAS` in `H` is returned and tested line 14. If the value is still the same, the write succeeded: the key already there has been evicted and we try to reinsert it in lines 15–30. If the value has changed, no write occurred: another thread wrote in `H` since our first read in line 10. We then check whether we must stop in lines 33–35. This is important to avoid duplication, as explained in Section 5.3.

Line 16–18: We record the location where the first (new) key is inserted. We will return this value to the caller at the end of the eviction sequence.

Line 19: In case of a successful insertion, two cases can occur. In the first case we inserted the key on an empty location (lines 20–23) and the insertion terminates. We record the location of the empty slot in `inserted_stu` as the location for the data of the new key. In the second case we inserted the key by evicting another (lines 25–29). We have to re-insert the previous key, and therefore re-initialize insertion. Note that we read the coordinates of the evicted key in `C` (line 26). This is correct since we never evict new keys for which an entry in `C` is not yet available (see line 11). We keep track of the `stu` record of the evicted key in `inserted_stu` (line 29). If the key cannot be reinserted, it will disappear from the cache. We will then reuse its `stu` location for the new key we initially inserted.

Line 32–35: The write in `H` failed due to a concurrent write. We must stop if we are currently inserting the new key. This avoids duplicated insertions as will be explained in Section 5.3.

Line 38–41: The current slot in `H` is occupied by a new key inserted in parallel by another thread. We must stop if we are currently inserting the new key. This avoids duplicated insertions (Section 5.3).

Line 44: The age of the current key is increased. The algorithm will try to insert it at its next location at the next iteration.

Line 45–47: The maximum allowed age is reached for a key: the algorithm stops and the key falls out of the hash table (deletion).

Note that this entire process runs for a limited number of iterations. Many keys will fall out of the hash. However, if the cache is large enough and if the viewpoint stops moving, the process stabilizes after a few frames.

5.3. Correctness

The algorithm has to ensure two important properties: 1) no key appears more than once, despite several threads trying to insert the same key; 2) no `stu` coordinate is used twice.

Avoiding duplicates. Duplicated keys are detected during insertion. If a new key is encountered while the thread is also trying to insert a new key (line 39), this could be a case of duplication. Since the table `C` is not yet updated for new keys, we cannot check the coordinates. We therefore blindly stop in this case (line 40).

If a write conflict occurs while inserting a new key (line 33), we also have to stop. If the conflict is with a new key, it could be a duplication. If the conflict is with an older key evicted by another thread, we again must stop. Indeed, let us assume that we anyway insert the new key at its next location: if another thread is trying to insert the same key later, it will evict the older key which created the conflict *before* detecting the duplication.

This guarantees that inserted keys are unique. Note that these tests are only necessary for new keys, since keys being evicted are always unique. In any case, the new key which failed to insert will get another chance at the next frame.

Allocation of data slots. The `stu` locations are allocated to new keys based on the following observation: At the end of a full insertion sequence we either 1) have found an empty slot, or 2) removed a key from the cache. Let us assume the new key was successfully inserted (otherwise we had no data to store). In both cases, we are given a slot for storing the data of the new key:

1. If an empty slot is found, we simply set `stu` to the location of the empty slot (line 21).
2. If a key falls out of the hash, we erase its data by reusing its `stu` data location for the data of the new key (line 29).

Note that by construction we are guaranteed that empty slots in `H` correspond to empty slots in `D` and `C`: A key reaching an empty slot in `H` fills the corresponding locations in `H`, `C`, and `D`. Slots in `H` are never emptied once occupied. Therefore, slots in `C` and `D` cannot be occupied while empty in `H`. This ensures that we never erase data which is in use.

5.4. Producing data

The last step of the algorithm, after ray marching and insertion, is to produce the data associated with the keys. All threads are synchronized before producing the data. This ensures that all insertions terminated. The `stu` coordinates of successfully inserted keys are then written at the key locations in `H`. Next, the key coordinates are written in `C`, and their data records are produced in `D` at their allocated `stu` coordinates. We discuss the content of each data record in Section 6.1.

6. Implementation

We implemented our scheme using OpenGL 4.2. Each of the three steps *Raymarch*, *Insert* and *Produce* corresponds to one shader. We perform the three steps in sequence. Temporary results are stored in OpenGL render buffers: `LastCacheMiss` after the *Raymarch* step, and `inserted_l`, `inserted_stu` after *Insert*. The three passes are necessary to synchronize all threads.

We exploit the `shader_image_load_store` extension to write directly into textures when updating the hash and producing data. This extension also provides the `atomicCAS` operation required by our approach.

6.1. Efficient tri-linear interpolation

Implementing tri-linear interpolation explicitly in a shader requires eight accesses to our data structure. In our case this implies eight possible cache misses and eight possible insertions, at *each step along the ray*. While this is possible, our tests resulted in very poor performance, in fact much slower than eight times the cost of a single access.

Lefebvre and Hoppe [13] faced a similar problem for texture interpolation. Their solution was simple and elegant, despite a significant increase in memory usage. Rather than storing single data points, they proposed to store interpolation cells made of 2^3 samples. Since these cells are stored in a volume texture, native hardware interpolation can be used *within a cell*. However, neighboring cells are not stored together, implying a significant overhead since each data point now potentially appears 8 times in different interpolation cells. This overhead can be reduced by relying on larger blocks.

We follow the same approach. In our case the overhead is however limited: our cache adapts to the view. We therefore designed our cache algorithm for this blocking scheme, which is the reason for the support for large data records. Each data record is in fact a full 2^3 floating point precision block.

During the *Produce* step of our algorithm, we compute the 8 values at the corners of the cache cell enclosing the inserted point. This implies that each data point is computed up to eight times more often (once per interpolation cell). One possibility would be to maintain a separate cache for single data points. Using larger blocks would also reduce this overhead since only boundary samples would be duplicated.

7. Results

We first analyze the behavior of our cache, and then present results of interactive rendering and modeling sessions. Please also refer to our accompanying video sequences for illustrations of these modeling sessions.

7.1. Setup

We test our hash mainly by displaying a textured icosahedron (Fig. 5), filling the viewport, at varying rotation speeds. Implicit base shapes, such as an icosahedron, can be easily obtained using *generalized distance functions* [1]. Because the hash cannot fit the entire data, keys will get inserted and deleted from the cache at every frame. Depending on the rotation speed, more or less cache misses will occur in-between frames. In all technical tests we use the same spot noise: spherical kernels, carving out from the surface, are randomly placed in space. The viewport resolution is 800×800. Unless otherwise specified, the maximum age is $A = 4$.

All our results were obtained on an Nvidia GTX 580 graphics card, running with driver version 295.51.

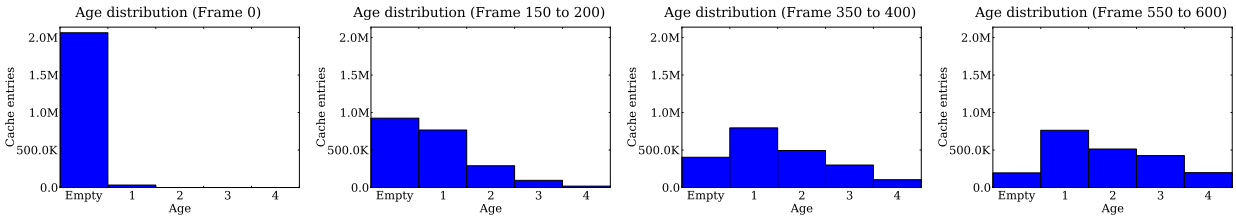


Figure 4: Histograms showing the distribution of the ages of cache entries, after 0, 200, 400, and 600 frames. The histogram stabilizes after this point. Note the bias toward younger keys.

7.2. Histogram of ages

As explained in Section 5.2, one of our objectives is to bias the histogram of keys toward younger keys. Figure 4 shows the age distribution of cache entries obtained for a rotation speed of 2 radians per second. The maximum age is discussed next.

7.3. Maximum age

We investigate the effect of changing the maximum age A. The tradeoff is that a higher maximum age leads to a more efficient insertion—and hence less deletions from the cache—but slower access. Figure 6 shows the behavior of the cache for different values of A. The shape is rotating on screen so that a similar amount of new data appears at every frame.

Note that the number of deletions is very high for $A = 2$: the cache deletes keys aggressively, among which many are in use for the current view. The overall rendering time suffers from this missing data. $A = 4$ behaves similarly to $A = 8$, $A = 12$ after 40 seconds. Before that point it is more difficult to insert keys and it needs more time until stabilization.

The rendering time for $A = 8$ and $A = 12$ is acceptable after 20 seconds, but then slows down after 40s: the cache is able to keep more keys in the hash, but has to push them further away along their sequences, reducing insertion and access performance for little benefit.

We experimentally selected $A = 4$ as a good tradeoff.

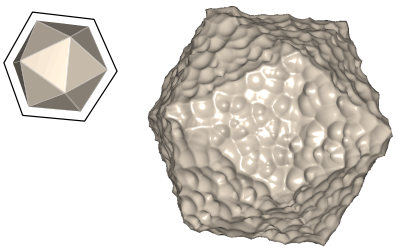


Figure 5: Textured icosahedron and implicit base shape. This object is used for the histograms in Figure 4, and throughout the analysis spanning Figures 6, 7, 8, and 9.

7.4. Hash table size

For a fixed cache cell size $(1/192)^3$ in texture space, we analyze the impact of a decreasing hash table size. The rotation speed is fixed to 2 rad/s. The number of cache misses and the render time are given in Figure 8.

Clearly, a larger hash table size directly benefits rendering by strongly reducing the number of cache misses. The constraint here is essentially how much memory can be allocated by the host. In our implementation we use a hash table size of 128^3, which requires a total size of 80 MB for all tables. Several such caches could therefore easily fit on a GPU, caching different procedural textures.

7.5. Cache misses and render time

We now analyze how the number of cache misses and the rendering time evolves for varying rotation speeds. For a given rotation speed, the values are averaged over 400 frames to allow for the cache to stabilize. Results are given in Figure 7. Slower rotations are faster to render, but note how the curves flatten as rotation speed increases: faster rotations do not require to refill the cache entirely, since many keys remain shared between consecutive frames. This is a good property for modeling, where rotating objects is a very common operation.

7.6. Procedure complexity and cache benefit

We analyze the benefit of using the cache for different procedural complexities. To this end, we significantly increase the number of impulses of our spot noise. We set the texture weight low so that the shape geometry is not significantly modified. The resulting performance ratio between using the cache or not is shown in Figure 9 for a rotation speed of 1 rad/s.

This shows that increased complexity leads to a larger performance ratio. However, this difference depends on the rotation speed. It is maximal for a fixed viewpoint—once everything is in the cache the rendering time is the same—and the difference decreases for larger rotation speeds. If many new keys appear, a complex procedure

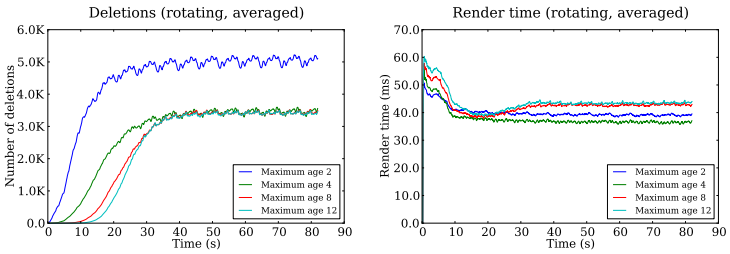


Figure 6: Number of cache deletions *(left)* and rendering time *(right)* for varying maximum ages A. (Icosahedron, rotation speed 2 rad/s.)

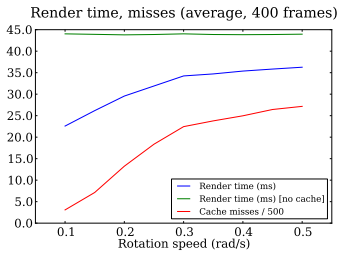


Figure 7: Cache misses and rendering time as a function of rotation speed.

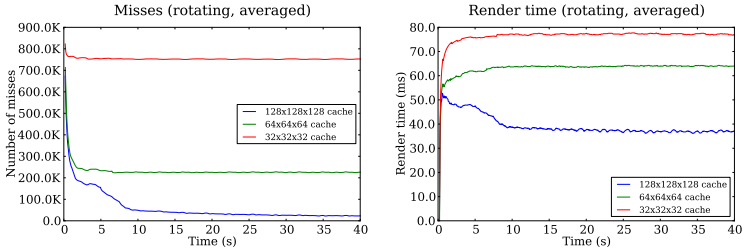


Figure 8: Number of cache misses *(left)* and rendering time *(right)* for a varying hash table size. (Icosahedron, rotation speed 2 rad/s.)

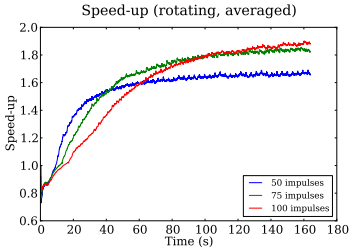


Figure 9: Cache speed-up ratios for different complexities of noise. (1 rad/s.)

will again provide less benefit that a simple one due to the large production of data.

In the first few frames the speed-up ratio is below one, since the cache has to be filled with a large number of missing data. Please also note that the speed-up is relatively low due to the rotation—it can be seen in Figure 7 that the speed-up is much larger at low rotation speeds.

7.7. Precision and shading

The cache stores a discretized version of the texture, which is tri-linearly interpolated. This results in a slightly different surface when enabling the cache, as shown in Figure 10. Our examples use a cell size of $(1/192)^3$, with the object fitting exactly into the unit cube. In practice we did not find the differences visually disturbing. Visual popping, however, may occur when keys are deleted or inserted. This could be entirely avoided by artificially limiting the resolution of the analytical function, evaluating it on a lattice with interpolation.

Figure 10 visualizes the loss of precision both in screen space and on the object's surface. *Center:* subtle shading differences *(blue)* and silhouette mismatches *(red)* appear in screen space. At the green curves, the deviation is 15 times a cell length along the view ray. *Right:* enabling the cache yields a low-distortion surface. We numerically computed the Hausdorff distance by sampling one surface and performing a gradient descent onto the other (and vice versa). For all our examples and for a variety of cell sizes the Hausdorff distance levels out at roughly 65% of a cell length. Red areas indicate that the Hausdorff distance is about to get reached. Of course, this is assuming that the cell size

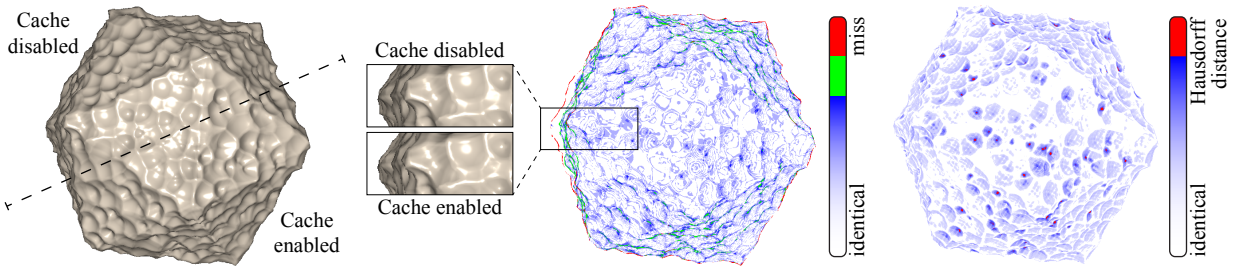


Figure 10: *Left:* surfaces compared with cache enabled/disabled. *Center:* for each pixel on the image plane, the Euclidean distance between these surfaces along the view ray is visualized; silhouette mismatches are red. *Right:* on the uncached surface, the distance to the cached version is mapped; in red regions the Hausdorff distance is reached.

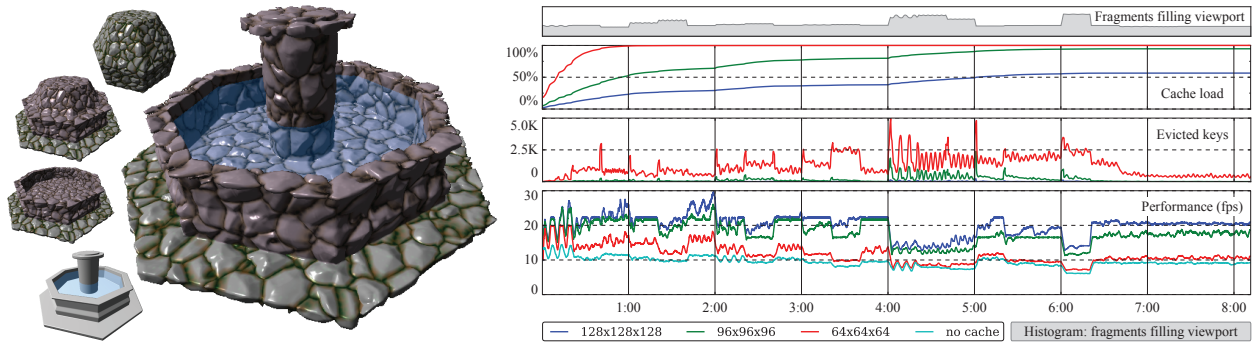


Figure 11: Modeling a more complex shape during a longer session: a fountain is constructed step by step. To the right, an extensive graph gives an insight into cache behavior throughout the eight minutes long session.

specified by the user is small enough to capture high frequencies. If it is not the case, the Hausdorff distance can be much higher as thin features may be entirely missed.

Computing the surface normals has a significant impact on performance. Using central differences requires six additional texture queries after surface intersection. Using forward differences halves the amount of queries, but results in less smooth, artifact-prone shading (see Figure 12). However, performance then almost doubles from 30 to 58 fps for the the icosahedron with increased spot noise impulses, and with caching enabled. Without caching, performance increases from 12 to only 14 fps.

Still, we opted for higher quality central differences for all examples and during performance measurements.

7.8. Long-term cache behavior

An important question is whether the cache behavior remains consistent in time. Figure 11 presents a complex modeling session, with a step-by-step construction of a fountain. It is assembled using two intersected generalized distance functions for the core base shape, followed by two pairs of intersected boxes to form the hexagonal border. Another two pairs of intersected boxes model the ground. The basin of the fountain is carved out by a difference operation. Two cylinders, another box, and another difference operation form the central column. Two more intersected boxes are used for the water. This sums up to 15 shapes and 8 CSG operations. A complex procedural stone texture,

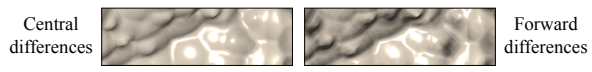


Figure 12: Using forward differences causes inferior shading, but improves cache performance significantly.

which features both color and displacement, is applied to the base shape during the entire session.

To analyze the cache behavior over a longer period of time, we scripted a modeling session during which the fountain is assembled. This script allows us to play back the same session repeatedly, profiling the cache with different choices of parameters.

Every 20 seconds, a new shape is added or a CSG operation is performed. Each new shape is inserted off-center and then gradually translated into place to emulate a typical modeling operation. Three of the boxes appear only for associated, following CSG operations and thereafter disappear. These boxes are large in spatial extent and thus have a heavy impact on the cache while they are visible on the screen. During the session, the camera constantly rotates at 0.5 rad/sec.

Please consult Figure 11 for our following analysis of three different cache sizes during the eight minutes long session. On the very top a histogram shows the proportion of fragments visible in the viewport. This directly influences performance, as ray marching is an image order rendering technique. Note the three plateaus; this is when extensive CSG proxy geometry is visible. The next plot reveals that the smallest cache (size 64^3, red curve) fills up within the first minute and then suffers clearly from insertions and involved evictions. The largest cache (size 128^3, blue curve), by contrast, has a generous amount of memory at it's disposal and never fills up entirely during the session; consequently, little evictions occur inside the largest cache, but almost half of its allocated memory remains unused. In between, a cache with size 96^3 (green curve) becomes entirely used, does not waste memory, yet generates few evictions: this mid-sized cache offers a good tradeoff and is most efficient. The cache size is the decisive factor for the amount of evicted keys, which is clearly reflected in

the center plot. This in turn impacts the performance, as the bottom plot reveals. The cache remains beneficial throughout the entire session: note how the green curve gradually catches up with the blue curve, again and again after each change in the scene. Large proxy geometry used for CSG is visible between minutes 4:00 and 5:00. Hereafter, the mid-size cache, even though entirely full now, still provides a large benefit, despite the evictions required to fit in newly inserted data; in addition, it does not waste memory as the largest cache does. Even if full, the caches quickly evict old data to adapt to abrupt large changes automatically, thus preserving performance.

In the end performance is slightly lower than initially, but note how the "no cache" curve also goes down: this highlights that there is no inherent cache degeneration, instead, the model just gets more and more complex.

7.9. Interactive modeling sessions

Figure 13 shows the evolution of cache misses when displaying a stone arch. Without the cache the arch is rendered at 11 fps, against 36 fps with the cache (fixed view, 800×800 resolution). Next, the same arch is being interactively deformed: the arch is parametric and the user changes its thickness and the height of its base. Thanks to the cache, only the modified parts require further computations. During this interaction performance ranged from 15 to 27 fps, depending on the magnitude of the changes. Performance is at its maximum when the user performs small, precise adjustments: this is an important property, because then a higher frame rate is most needed for fast visual feedback.

This also suits well for sketch-based applications; in Figure 14, to the left, the user sketches "SMI", which is extruded while sketching, and simultaneously inserted into the cache. On the right, an example is given where

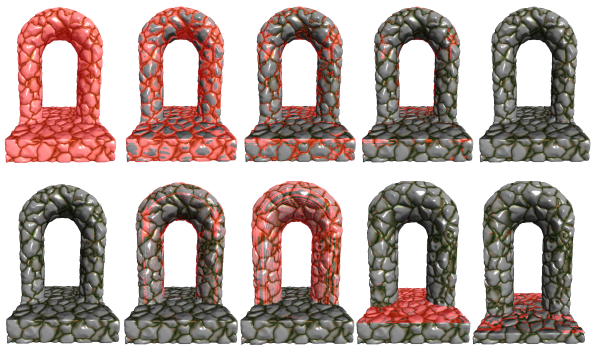


Figure 13: *Top row:* the first five frames after clearing the cache. Cache misses appear in red. *Bottom:* the user increases the radius of the arch and then lowers the base.

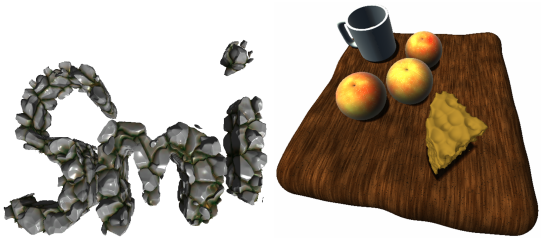


Figure 14: More examples: a sketching application and a scene with three independently cached textures.

we independently cache three different noise textures for multiple objects. This allows the user, for instance, to adjust the placement and shapes of apples, without influencing other, unassociated caches.

In Figure 15, an expensive tree bark texture is applied to a cylinder. After enabling the cache, performance raises from barely 6 to almost 40 fps. The user can adjust a variety of parameters, controlling the final appearance. Changing these does not invalidate the cache, which stores the base noise used to generate the bark appearance instead of the final result. Additionally, interaction remains smooth while the user can stretch, bend, and rotate cracks: we access already cached values for different positions in screen space with a simple transformation of the texture domain.

8. Limitations and future work

There are several limitations in our current approach. First, the spatial resolution of the cache is homogeneous in space: on large extent scenes the cache quickly saturates due to distant surfaces. A typical solution to this issue is to rely on a multi-resolution cache [4, 5]. This would also accommodate for cases where the data exhibits a non-uniform resolution across space.

A second limitation is that large disocclusions create peaks in misses, generating a large number of insertions. This may reduce performance for a few frames below the performance obtained without the cache. We believe this problem could be alleviated by further limiting the maximum number of insertions per frame. Optimizing insertions further would also reduce this issue.

Section 7.7 revealed that shading has a significant impact on performance. It should be investigated whether efficiently caching texture derivatives can help to accelerate surface normal approximation.

There is, of course, a more fundamental limit to the benefit of a cache: as it exploits spatio-temporal coherence, any change in the definition of the texture triggers a cache-wide refresh, and all prior values have to be discarded. There is little to be done about this issue.

Figure 15: Modeling with a complex tree bark texture; interaction remains smooth while adjusting parameters.

9. Conclusions

Our runtime cache is a first step toward significantly accelerating the rendering and modeling of textured implicit surfaces. It integrates seamlessly into the on-demand evaluation of implicit procedural descriptions; with lazy evaluation, we seek for the best balance between computation and memory by exploiting spatio-temporal coherence between consecutive frames.

Performance is currently limited by technical factors; OpenGL 4.2 is at an early stage of development, and we believe many optimizations will be available soon.

Our algorithm is very general and could be used to cache many other quantities in space, such as lighting, opacity, or eventually full hypertextures [16]. Further procedural modeling techniques even extend to the highly (or fully) automated creation of landscapes, whole cities, and diversified natural scenes; it would be interesting to adapt our work to accelerate key parts of these techniques as well. We believe this work will find many applications for rendering and modeling high-quality, highly detailed surfaces.

Acknowledgments. This work was partially funded by the Agence Nationale de la Recherche (SIMILAR-CITIES 2008-COORD-021-01) and an INRIA Nancy Grand-Est collaboration grant. Tim Reiner is supported by the Intel Visual Computing Institute, Saarbrücken. Ismael García is supported by the Ministerio de Ciencia e Innovación, Spain (TIN2010-20590-C02-02). Iñigo Quilez kindly allowed us to use his procedural shaders for the apple scene in Fig. 14. We thank the reviewers for their helpful suggestions for improvements.

References

[1] Akleman, E., Chen, J., 1999. Generalized distance functions. In: Shape Modeling International 1999. IEEE, pp. 72–79.

[2] Alcantara, D., Sharf, A., Abbasinejad, F., Sengupta, S., Mitzenmacher, M., Owens, J., Amenta, N., Dec. 2009. Real-time parallel hashing on the GPU. ACM Transaction on Graphics (SIGGRAPH Asia 2009) 28 (5), 154:1–154:9.

[3] Alcantara, D., Volkov, V., Sengputa, S., Mitzenmacher, M., Owens, J., Amenta, N., 2011. Building an efficient hash table on the GPU. GPU Computing Gems, Jade Edition.

[4] Bastos, T., Celes, W., 2008. GPU-accelerated adaptively sampled distance fields. In: Shape Modeling International 2008. IEEE, pp. 171–178.

[5] Castro, R., Lewiner, T., Lopes, H., Tavares, G., Bordignon, A., 2008. Statistical optimization of octree searches. Computer Graphics Forum 27 (6), 1557–1566.

[6] Crassin, C., Neyret, F., Lefebvre, S., Eisemann, E., 2009. Gigavoxels: Ray-guided streaming for efficient and detailed voxel rendering. In: ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games (I3D 2009). ACM, pp. 15–22.

[7] Ebert, D. S., Musgrave, F. K., Peachey, D., Perlin, K., Worley, S., 2002. Texturing and Modeling: A Procedural Approach, 3rd Edition. Morgan Kaufmann Publishers Inc., San Francisco, CA.

[8] Frisken, S., Perry, R., Rockwood, A., Jones, T., 2000. Adaptively sampled distance fields: a general representation of shape for computer graphics. In: SIGGRAPH 2000. ACM, pp. 249–254.

[9] García, I., Lefebvre, S., Hornus, S., Lasram, A., 2011. Coherent parallel hashing. ACM Transactions on Graphics (SIGGRAPH Asia 2011) 30 (6), 161:1–161:8.

[10] Hart, J. C., 1996. Sphere tracing: A geometric method for the antialiased ray tracing of implicit surfaces. The Visual Computer 12, 527–545.

[11] Hornus, S., Angelidis, A., Cani, M.-P., 2003. Implicit modeling using subdivision curves. The Visual Computer 19, 94–104.

[12] Lagae, A., Lefebvre, S., Cook, R., DeRose, T., Drettakis, G., Ebert, D., Lewis, J., Perlin, K., Zwicker, M., 2010. A survey of procedural noise functions. Computer Graphics Forum 29 (8), 2579–2600.

[13] Lefebvre, S., Hoppe, H., 2006. Perfect spatial hashing. ACM Transaction on Graphics (SIGGRAPH 2006) 25 (3), 579–588.

[14] Loop, C., Blinn, J., 2006. Real-time GPU rendering of piecewise algebraic surfaces. ACM Transactions on Graphics (SIGGRAPH 2006) 25, 664–670.

[15] Lorensen, W., Cline, H., 1987. Marching cubes: A high resolution 3D surface construction algorithm. Computer Graphics (SIGGRAPH 1987) 21 (4), 163–169.

[16] Perlin, K., Hoffert, E., 1989. Hypertexture. Computer Graphics (SIGGRAPH 1989) 23 (3), 253–262.

[17] Reiner, T., Mückl, G., Dachsbacher, C., 2011. Interactive modeling of implicit surfaces using a direct visualization approach with signed distance functions. Computers & Graphics (Shape Modeling International 2011) 35 (3), 596–603.

[18] Schmidt, R., Wyvill, B., Galin, E., 2005. Interactive implicit modeling with hierarchical spatial caching. In: Shape Modeling International 2005. IEEE, pp. 104–113.

[19] Schmidt, R., Wyvill, B., Sousa, M. C., Jorge, J. A., 2005. Shapeshop: Sketch-based solid modeling with blobtrees. In: SBIM 2005. pp. 53–62.

[20] Sederberg, T. W., 1985. Piecewise algebraic surface patches. Computer Aided Geometric Design 2 (1–3), 53–59.

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