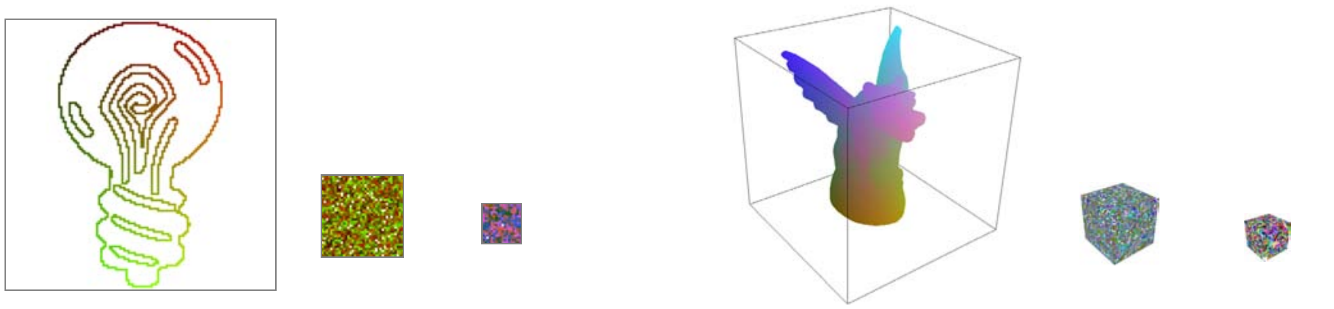
Perfect Spatial Hashing

Sylvain Lefebvre Hugues Hoppe
Microsoft Research



Sparse 2D data n=1381 pixels in 128^2 image | Hash table H $m=38^2$ (=1444) | Offset table Φ $r=18^2$ | Sparse 3D data n=41,127 voxels in 128^3 volume | Hash table H $m=35^3$ (=42,875) | Offset table Φ $r=19^3$

Figure 1: Representation of sparse spatial data using nearly minimal perfect hashes, illustrated on coarse 2D and 3D examples.

Abstract

We explore using hashing to pack sparse data into a compact table while retaining efficient random access. Specifically, we design a perfect multidimensional hash function – one that is precomputed on static data to have no hash collisions. Because our hash function makes a single reference to a small offset table, queries always involve exactly two memory accesses and are thus ideally suited for parallel SIMD evaluation on graphics hardware. Whereas prior hashing work strives for pseudorandom mappings, we instead design the hash function to preserve spatial coherence and thereby improve runtime locality of reference. We demonstrate numerous graphics applications including vector images, texture sprites, alpha channel compression, 3D-parameterized textures, 3D painting, simulation, and collision detection.

Keywords: minimal perfect hash, multidimensional hashing, sparse data, adaptive textures, vector images, 3D-parameterized textures.

1. Introduction

Many graphics applications involve sparsely defined spatial data. For example, image discontinuities like sharp vector silhouettes are generally present at only a small fraction of pixels. Texture sprites overlay high-resolution features at sparse locations. Image attributes like alpha masks are mainly binary, requiring additional precision at only a small subset of pixels. Surface texture or geometry can be represented as sparse 3D data.

Compressing such sparse data while retaining efficient random-access is a challenging problem. Current solutions incur significant memory overhead:

- Data quantization is lossy, and uses memory at all pixels even though the vast majority may not have defined data.
- Block-based indirection tables typically have many unused entries in both the indirection table and the blocks.
- Intra-block data compression (including vector quantization) uses fixed-length encodings for fast random access.
- Quadtree/octree structures contain unused entries throughout their hierarchies, and moreover require a costly sequence of pointer indirections.

Hashing. We instead propose to losslessly pack sparse data into a dense table using a hash function $h(p)$ on position p. Applying traditional hashing algorithms in the context of current graphics architecture presents several challenges:

(1) Iterated probing: To cope with collisions, hashing algorithms typically perform a sequence of probes into the hash table, where the number of probes varies per query. This probing strategy is inefficient in a GPU, because SIMD parallelism makes all pixels wait for the worst-case number of probes. While GPUs now have dynamic branching, it is only effective if all pixels in a region follow the same branching path, which is unlikely for hash tests.

(2) Data coherence: Avoiding excessive hash collisions and clustering generally requires a hash function that distributes data seemingly at random throughout the table. Consequently, hash tables often exhibit poor locality of reference, resulting in frequent cache misses and high-latency memory accesses [Ho 1994].

Perfect hashing. To make hashing more compatible with GPU parallelism, we explore using a *perfect* hash function – one that is precomputed for a static set of elements to have no collisions. Moreover, we seek a *minimal* perfect hash function – one in which the hash table contains no unused entries.

There has been significant research on perfect hashing for more than a decade, with both theoretical and practical contributions as reviewed in Section 2. An important theoretical result is that perfect hash functions are rare in the space of all possible functions. In fact, the *description* of a minimal perfect hash function is expected to require a number of bits proportional to the number of data entries. Thus one cannot hope to construct a perfect hash using an expression with a small number of machine-precision parameters. Instead, one must resort to storing additional data into auxiliary lookup tables.

In this paper, we define a perfect multidimensional hash function of the form

$$h(p) = h_0(p) + \Phi\left[h_1(p)\right],$$

which combines two imperfect hash functions h_0,h_1 with an *offset table* Φ. Intuitively, the role of the offset table is to “jitter” the imperfect hash function h_0 into a perfect one. Although the offset table uses additional memory, it can fortunately be made significantly smaller than the data itself – typically it has only 15-25% as many entries, and each entry has just 8 bits per coordinate.

Spatial coherence. Prior work on perfect hashing has focused on external storage of data records indexed by character strings or sparse integers. To our knowledge, no work has considered multidimensional data and its unique opportunities. Indeed, in computer graphics, 2D and 3D texture data is often accessed coherently by the parallel GPU, and is therefore swizzled, tiled, and cached. Ideally, hashed textures should similarly be designed to exploit access coherence.

Our solution has two parts. Whereas prior work seeks to make intermediate hash functions like h_0,h_1 as random as possible, we instead design them to be spatially coherent, resulting in efficient access into the offset table Φ. Remarkably, we define h_0,h_1 simply as modulo (wraparound) addressing over the tables. Second, we optimize the offset values in Φ to maximize coherence of *h* itself. Creating a perfect hash is already a difficult combinatorial problem. Nonetheless, we find that there remain enough degrees of freedom to improve coherence and thereby increase runtime hashing performance.

Implemented on the GPU, our perfect hash allows data access using just one additional texture access, plus only about 4-6 more shader instructions depending on the application scenario.

Sparsity encoding. In addition to sparse data compaction, we also describe several schemes for encoding the spatial positions of these sparse samples. Specifically, we introduce domain bits, position tags, and parameterized position hashes.

Filtering and blocking. For applications that require continuous local interpolation of the sparse data, we consider two approaches. The first is to allow native filtering in the dedicated GPU hardware by grouping the data into sample-bordered blocks [Kraus and Ertl 2002]. In this setting, our contribution is to replace the traditional block indirection table by a compact spatial hash over the sparsely defined blocks. The limiting factor in using blocks is that data must be duplicated along block boundaries, thus discouraging small blocks sizes and leading to memory bloat.

Our second solution attains a more compact representation by forgoing blocking and instead performing filtering explicitly as general-purpose computation. At present this incurs an appreciable loss in performance, but can reduce memory by a factor 3 over blocking schemes.

Examples. To provide context to the discussion, we first illustrate our scheme using two coarse examples in Figure 1. The domain data values are taken from simple linear color ramps, and the offset table vectors are visualized as colors.

In the 2D example, the 128^2 image contains a set of 1,381 pixels (8.4%) with supplemental information, e.g. vector silhouette data. This sparse pixel data is packed into a hash table of size 38^2=1,444, which is much smaller than the original image. The perfect hash function is defined using an offset table of size 18^2.

In the 3D example, a triangle mesh is colored by accessing a 3D texture of size 128^3. Only 41,127 voxels (2.0%) are accessed when rendering the surface using nearest-filtering. These sparse voxels are packed into a 3D table of size 35^3=42,875 using a 19^3 offset table.

2. Related work on hashing

Perfect hashing. We can provide here only a brief summary of the literature. For an extensive survey, refer to [Czech et al 1997].

The probability that randomly assigning *n* elements in a table of size *m* results in a perfect hash is

$$\Pr_{\mathrm{PH}}(n,m) = (1)\cdot\left(1-\tfrac{1}{m}\right)\cdot\left(1-\tfrac{2}{m}\right)\cdots\left(1-\tfrac{n-1}{m}\right).$$

When the table is large (i.e. $m \gg n$), we can use the approximation $e^x \approx 1+x$ for small *x* to obtain:

$$\Pr_{\mathrm{PH}}(n,m) \approx 1\cdot e^{-1/m}\cdot e^{-2/m}\cdots e^{-(n-1)/m}$$
$$= e^{-(1+2+\ldots+(n-1))/m} = e^{-(n(n-1)/2m)} \approx e^{-n^2/2m}.$$

Thus, the presence of a hash collision is highly likely when the table size *m* is much less than n^2. This is an instance of the well known “birthday paradox” – a group of only 23 people have more than 50% chance of having at least one shared birthday.

The probability of finding a *minimal* perfect hash (where *n*=*m*) is:

$$\Pr_{\mathrm{PH}}(n) = \left(\tfrac{n}{n}\right)\cdot\left(\tfrac{n-1}{n}\right)\cdot\left(\tfrac{n-2}{n}\right)\cdots\left(\tfrac{1}{n}\right)$$
$$= \tfrac{n!}{n^n} = e^{(\log n! - n\log n)} \approx e^{((n\log n - n) - n\log n)} = e^{-n}$$

which uses Stirling’s approximation $\log n! \approx n\log n - n$. Therefore, the expected number of bits needed to describe these rare minimal perfect hash functions is intuitively

$$\log_2 \tfrac{1}{\Pr_{\mathrm{PH}}(n)} \approx \log_2 e^n = (\log_2 e)n \approx (1.443)n,$$

as reported by Fox et al [1992] and based on earlier analysis by Mehlhorn [1982].

Several number-theoretical methods construct perfect hash functions by exploiting the Chinese remainder theorem [e.g. Winters 1990]. However, even for sets of a few dozen elements, these functions involve integer coefficients with hundreds of digits.

A more computer-amenable approach is to define the hash using one or more auxiliary tables. Fredman et al [1984] use three such tables and two nested hash functions to hash a sparse set of *n* integers taken from $\mathbb{Z}_u = \{0,\ldots,u-1\}$. Their scheme takes constant time and $3n\log n$ bits of memory. The hash is constructed with a deterministic algorithm that takes $O(nu)$ time. Schmidt and Siegel [1990] reduce space complexity to the theoretically optimal $\Theta(n)$ bits, but the constant is large and the algorithm difficult.

Some schemes [e.g. Brain and Tharp 1990] treat perfect hashing as an instance of sparse matrix compression. They map a bounded range of integers to a 2D matrix, and compact the defined entries into a 1D array by translating the matrix rows. Sparse matrix compression is known to be NP-complete.

The most practical schemes achieve compact representations and scale to larger datasets by giving up guarantees of optimality. These probabilistic constructions may iterate over several random parameters until finding a solution. For example, Sager [1985] defines a hash $h(k) = h_0(k) + g_1[h_1(k)] + g_2[h_2(k)] \bmod m$, where functions h_0,h_1,h_2 map string keys *k* to $\mathbb{Z}_m, \mathbb{Z}_r, \mathbb{Z}_r$ respectively, and g_1,g_2 are two tables of size *r*. However, this algorithm takes expected time $O(r^4)$, and is practical only up to *n*=512 elements.

Fox et al [1992] adapt Sager’s approach to create the first scheme with good average-case performance (~11*n* bits) on large datasets. Their insight is to assign values of auxiliary tables g_1,g_2 in decreasing order of number of dependencies. They also describe a second scheme that uses quadratic hashing and adds branching based on a table of binary values; this second scheme achieves ~4*n* bits for datasets of size *n*~10^6.

Previous work considered either perfect hashing of strings, or perfect hashing of integers in a theoretical setting. Our contribution is to extend the basic framework of [Sager 1985] to 2D and 3D spatial input domains and hash tables. We discover that multidimensional tables allow effective hashing using a single auxiliary table together with extremely simple functions h_0,h_1. Whereas prior work strives to make these intermediate hashes as random as possible [Östlin and Pagh 2003], we instead design them to preserve coherence. We adapt the heuristic ordering strategy of [Fox et al 1992] and extend the construction algorithm to optimize data coherence.

Spatial hashing. Hashing is commonly used for point and region queries in multidimensional databases [Gaede and Günther 1998]. Spatial hashing is also used in graphics for efficient collision detection among moving or deforming objects [e.g. Mirtich 1996; Teschner et al 2003]. However, all these techniques employ imperfect hashing – traditional multi-probe CPU hash tables.

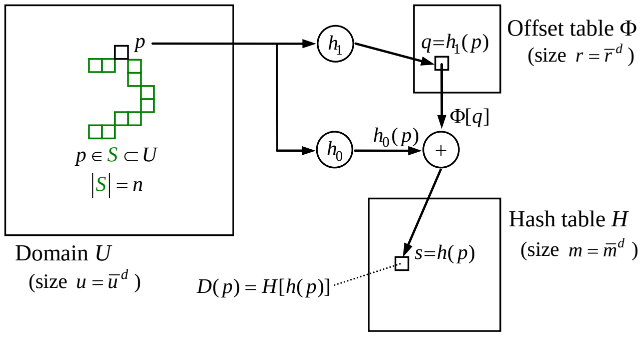


Figure 2: Illustration of our hash function definition. The "hash" functions h_0,h_1 are in fact simple modulo addressing.

3. Our perfect hashing scheme

3.1 Overview and terminology

We assume that the spatial domain U is a d-dimensional grid with $u = \bar{u}^d$ positions, denoted by $\mathbb{Z}_{\bar{u}}^d = [0 \ldots (\bar{u}{-}1)]^d$. In this paper we present results for both 2D and 3D domains.

The sparse data consists of a subset $S \subset U$ of n grid positions, where each position $p \in S$ has an associated data record $D(p)$. Thus the data density is the fraction $\rho = n/u$. For datasets of codimension 1, such as curves in 2D or surfaces in 3D, we typically find that $\rho \sim 1/\bar{u}$.

Our goal is to replace the sparsely defined data $D(p)$ by a densely packed hashed texture $H\left[h(p)\right]$ where:

- the *hash table* H is a d-dimensional array of size $m = \bar{m}^d \geq n$ containing the data records $D(p), p \in S$;
- the *perfect hash function* $h(p) : U \to H$ is an injective map when restricted to S, mapping each position $p \in S$ to a unique *slot* $s = h(p)$ in the hash table.

As illustrated in Figure 2, we form this perfect hash function as

$$h(p) = h_0(p) + \Phi\left[h_1(p)\right] \bmod \bar{m}\text{, where:}$$

- the *offset table* Φ is a d-dimensional array of size $r = \bar{r}^d = \sigma n$ containing d-dimensional vectors;
- the map $h_0 : p \to M_0 p \bmod \bar{m}$ from domain U onto hash table H is a simple linear transform with a $d{\times}d$ matrix M_0, modulo the table size;
- the map $h_1 : p \to M_1 p \bmod \bar{r}$ from domain U onto the offset table Φ is similarly defined.

As we shall see in Section 4.2, we let M_0,M_1 be identity matrices, so h_0,h_1 correspond simply to modulo addressing. (All modulo operations are performed per-coordinate on vectors.)

3.2 GPU implementation

For simplicity, we describe the scheme mathematically using arrays with integer coordinates (e.g. $0,1,\ldots,\bar{m}{-}1$) and integer values. These are presently implemented as 2D/3D textures with normalized coordinates (e.g. $\frac{0.5}{\bar{m}},\frac{1.5}{\bar{m}},\ldots,\frac{\bar{m}-0.5}{\bar{m}}$) and normalized colors (e.g. $\frac{0}{255},\frac{1}{255},\ldots,\frac{255}{255}$). The necessary scales and offsets are a slight complication, and should be obviated by integer support in upcoming GPUs [Blythe 2006]. All modulo operations in the hash definition are obtained for free in the texture sampler by setting the addressing mode to wrap. Textures with arbitrary (non-power-of-two) sizes are supported in current hardware.

We find that quantizing the offset vectors stored in Φ to 8 bits per coordinate provides enough flexibility for hashing even when the hash table size $\bar{m}$ exceeds 2^8=256. Therefore we let Φ be a d-channel 8-bit texture. However, to avoid bad clustering during hash construction it is important to allow the offsets to span the full hash table, so we scale the stored integers $\mathbb{Z}_{256}^d$ by $\lceil \bar{m}/255 \rceil$.

The following is HLSL pseudocode for the hashing function:

```
static const int d=2;   // spatial dimensions (2 or 3)
typedef vector<float,d> point;
#define tex(s,p) (d==2 ? tex2D(s,p) : tex3D(s,p))

sampler SOffset, SHData; // tables Φ and H.
matrix<float,d,d> M[2]; // M₀,M₁ prescaled by 1/m̄ ,1/r̄ .

point ComputeHash(point p) { // evaluates h(p) → [0,1]^d
  point h0 = mul(M[0],p);
  point h1 = mul(M[1],p);
  point offset = tex(SOffset, h1) * oscale; // (*⌈m̄/255⌉)
  return h0 + offset;
}

float4 HashedTexture(point pf) : COLOR {
  // pf is prescaled into range [0, ū ] of space U
  point h = ComputeHash(floor(pf));
  return tex(SHData, h);
}
```

While the above code is already simple, several easy optimizations are possible. For 2D domains, the two matrix multiplications `mul(M[i],p)` are done in parallel within a float4 tuple. As we shall see in the next section, the matrices M_0,M_1 are in fact scaled identity matrices, so the matrix multiplications reduce to a single multiply instruction.

In most applications, the input coordinates are continuous and must be discretized prior to the hash using a floor operation. We sought to remove this floor operation by using "nearest" sampling access to textures Φ and H, but unfortunately current hardware seems to lack the necessary precision. Otherwise, the coordinate scalings could also be moved to the vertex shader. For now, the `HashedTexture` function above reduces to 6 instructions in 2D.

4. Hash construction

We seek to assign table sizes $\bar{m},\bar{r}$, hash coefficients M_0,M_1, and offset values in table Φ such that

(1) $h(p)$ is a perfect hash function;

(2) Φ and H are as compact as possible;

(3) the accesses to the tables, $\Phi\left[h_1(p)\right]$ and $H\left[h(p)\right]$, have good coherence with respect to p.

Our strategy is to let the hash table be as compact as possible to contain the given data, and then to construct the offset table to be as small as possible while still allowing a perfect hash function. More precisely, we introduce a greedy algorithm for filling the offset table (Section 4.3), and iteratively call this algorithm with different offset table sizes (Section 4.1). For a large enough offset table, success is guaranteed, so our iterative process always terminates with a perfect hash.

4.1 Selection of table sizes

For simplicity, we assume that the hash table is a square or cube. We let its size $\bar{m}$ be the smallest value such that $m = \bar{m}^d \geq n$. Because we quantize offset values to 8 bits, not all offset values are representable if $\bar{m} > 256$, in which case we slightly increase the table size to $m = \bar{m}^d \geq (1.01)n$ to give enough leeway for a perfect hash. Consequently, the perfect hash function is not strictly minimal, but the number $m - n$ of unused entries is small.

Next, we seek to assign the offset table size $\bar{r}$ to be as small as possible while still permitting a perfect hash. We have explored two different strategies for selecting $\bar{r}$, depending on whether the speed of hash construction is important or not.

For fast construction, we initially set the offset table size $\bar{r}$ to the smallest integer such that $r = \bar{r}^d \geq \sigma n$ with the factor σ=1/(2d). Because the offset table contains d channels of 8 bits each, this size corresponds to 4 bits per data entry, and allows a perfect hash in many cases. If the hash construction fails, we increase $\bar{r}$ in a geometric progression until construction succeeds.

For compact construction, we perform a binary search over $\bar{r}$. Because our construction is greedy and probabilistic, finding a perfect hash for a given table size $\bar{r}$ may require several attempts, particularly if $\bar{r}$ is close to optimal. We make up to 5 such attempts using different random seeds. Hash construction is a preprocess, so investing this extra time may be reasonable.

We find empirically that hash construction is less effective if $\bar{r}$ has a common factor with $\bar{m}$ or if $\bar{m} \bmod \bar{r} \in \{1, \bar{r}-1\}$. Thus for the fast construction we automatically skip these unpromising sizes, and for the compact construction we adapt binary search to avoid testing them unless at the lower-end of the range.

4.2 Selection of hash coefficients

Our initial approach was to fill the matrices M_0,M_1 (defining the intermediate hash functions h_0,h_1) with random prime coefficients, in the same spirit as prior hashing work. To improve hash coherence we then sought to make these matrices more regular.

To our great amazement, we found that letting M_0,M_1 just be *identity matrices* does not significantly hinder the construction of a perfect hash. The functions h_0,h_1 then simply wrap the spatial domain multiple times over the offset and hash tables, moving over domain points and table entries in lockstep. Thus the offset table access $\Phi[h_1(p)]$ is perfectly coherent. Although $h_0(p)$ is also coherent, the hash table access $H[h(p)]$ is generally not because it is jittered by the offsets. However, if adjacent offset values in Φ are the same, i.e. if the offset table is locally constant, then h itself will also be coherent. This property is considered further in Section 4.4.

One necessary condition on h_0,h_1 is that they must map the defined data to distinct pairs. That is, $p \in S \rightarrow (h_0(p), h_1(p))$ must be injective. Indeed, if there were two points p_1,$p_2 \in S$ with $h_0(p_1)$=$h_0(p_2)$ and $h_1(p_1)$=$h_1(p_2)$, then these points would always hash to the same slot $h(p_1)$=$h(p_2)$ regardless of the offset stored in $\Phi[h_1(p_1)]$, making a perfect hash impossible.

The condition for injectivity is similar to a perfect hash of n elements into a table of size $|H| \times |\Phi| = mr$. Applying the same formula as in Section 2, we derive a probability of success of

$$\Pr_{PH} \approx e^{-n^2/(2mr)} \approx e^{-n/2r},$$

which seems ominously low – only 12% for $\sigma = r/n = 0.25$. We believe that this is the main reason that previous perfect hashing schemes resorted to additional tables and hash functions.

However, unlike prior work we do not select h_0,h_1 to be random functions. Because our functions h_0,h_1 have periodicities $\bar{m}$ and $\bar{r}$ respectively, if these periodicities are coprime then we are guaranteed injectivity when the domain size $\bar{u} \leq \bar{m}\bar{r}$, or equivalently (since $m \approx n$) when the data density $\rho = n/u \geq 1/r$. In practice, r is typically large enough that this is always true, and thus we need not test for injectivity explicitly.

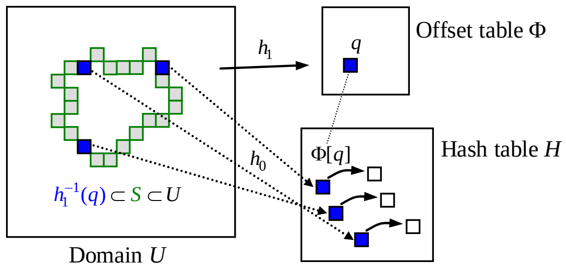


Figure 3: The assignment of offset vector $\Phi[q]$ corresponds to a uniform translation of the points $h_1^{-1}(q)$ within the hash table.

4.3 Creation of offset table

As shown in Figure 3, on average each entry q of the offset table is the image through h_1 of $\sigma^{-1} = n/r \approx 4$ data points – namely the set $h_1^{-1}(q) \subset S$. The assignment of the offset vector $\Phi[q]$ determines a uniform translation of these points within the hash table. The goal is to find an assignment that does not collide with other points hashed in the table.

A naïve algorithm for creating a perfect hash would be to initialize the offset table Φ to zero or random values, look for collisions in $h(p)$, and try to "untangle" these by perturbing the offset values, for instance using random descent or simulated annealing. However, this naïve approach fails because it quickly settles into imperfect minima, from which it would take a long sequence of offset reassignments to find a lower total number of collisions.

A key insight from [Fox et al 1992] is that the entries $\Phi[q]$ with the largest sets $h_1^{-1}(q)$ of dependent data points are the most challenging to assign, and therefore should be processed first. Our algorithm assigns offset values greedily according to this heuristic order (computed efficiently using a bucket sort). For each entry q, we search for an offset value $\Phi[q]$ such that the data entries $h_1^{-1}(q)$ do not collide with any data previously assigned in the hash table, i.e.,

$$\forall p \in h_1^{-1}(q),\; H[h_0(p) + \Phi[q]] = undef\ .$$

The space of 8-bit-quantized offset values is $\mathbb{Z}^d_{\min(\bar{m},256)} \lceil \bar{m}/255 \rceil$. In practice it is important to start the search at a random location in this space, or else bad clustering may result. If no valid offset $\Phi[q]$ is found, one could try backtracking, but we found results to be satisfactory without this added complexity. Note that towards the end of construction, the offset entries considered are those with exactly one dependent point, i.e. $|h_1^{-1}(q)|$=1. These are easy cases, as the algorithm simply finds offset values that direct these sole points to open hash slots.

4.4 Optimization of hash coherence

Because we assign M_1 to be the identity matrix, accesses to the offset table Φ are always coherent. We next describe how hash construction is modified to increase coherence of access to *H*.

First, we consider the case that hash queries are constrained to the set of defined entries $S \subset U$. Let $N_S(p_1, p_2)$ be 1 if two defined points $p_1, p_2 \in S$ are spatially adjacent in the domain (i.e. $\|p_1 - p_2\| = 1$), or 0 otherwise. And, let $N_H(s_1, s_2)$ be similarly defined for slots in tables *H*. We seek to maximize

$$\begin{aligned}\mathcal{N}_H &= \sum\nolimits_{p_1, p_2 \,|\, N_S(p_1,p_2)=1} N_H\left(h(p_1), h(p_2)\right) \\ &= \sum\nolimits_{p_1, p_2 \,|\, N_H(h(p_1),h(p_2))=1} N_S\left(p_1, p_2\right).\end{aligned}$$

It is this latter expression that we measure during construction. When assigning an offset value $\Phi[q]$, rather than selecting any value that is valid, we seek one that maximizes coherence. Specifically, we examine the slots of *H* into which the points $h_1^{-1}(q)$ map, and count how many neighbors in *H* are also neighbors in the spatial domain:

$$\max_{\Phi[q]} \mathcal{C}\left(\Phi[q]\right), \quad \mathcal{C}\left(\Phi[q]\right) = \sum_{p \in h_1^{-1}(q),\, \|\Delta\|=1} N_S\left(h^{-1}\left(h_0(p) + \Phi[q] + \Delta\right), p\right).$$

As it is too slow to try all possible values for $\Phi[q]$, we consider the following heuristic candidates:

(1) We try setting the offset value equal to one stored in a neighboring entry of the offset table, because the hash function is coherent if the offset table is locally constant:

$$\Phi[q] \in \left\{\Phi[q'] \;\middle|\; \|q - q'\| < 2\right\}.$$

(2) For each point $p \in h_1^{-1}(q)$ associated with *q*, we examine its domain-neighboring entries $p' \in S$. If a neighbor p' is already assigned in the table *H*, we see if any neighboring slot in *H* is free, and if so, try the offset that would place *p* in that free slot:

$$\begin{aligned}\Phi[q] \in \Big\{ h(p') + \Delta - h_0(p) \;\Big|\; & p \in h_1^{-1}(q),\; p' \in S,\; N_S\left(p, p'\right) = 1, \\ & h(p') \neq undef,\; \|\Delta\| = 1,\; H\left[h(p') + \Delta\right] = undef \Big\}.\end{aligned}$$

In the case that hash queries can span the full domain *U*, then local constancy of Φ is most important, and we give preference to the candidates found in (1) above.

As a postprocess, any undefined offset entries (i.e. for which $h_1^{-1}(q) = \varnothing$) are assigned values coherent with their neighbors. In Table 2 we show obtained values for the normalized coherence metric $\tilde{\mathcal{N}}_H = \mathcal{N}_H / \sum_{p_1, p_2} N_S(p_1, p_2)$.

5. Sparsity encoding

The hash table stores data associated with a sparse subset of the domain. Depending on the application, it may be necessary to determine if an arbitrary query point lies in this defined subset.

Constrained access. Some scenarios such as 3D-parameterized surface textures guarantee that only the defined subset of the domain will ever be accessed. This is by far the simplest case, as the hash table need not store anything except the data itself, and no runtime test is necessary.

Domain bit. For scenarios involving unconstrained access, one practical approach is to store a *binary image* over the domain, where each pixel (bit) indicates the presence of data (or blocks of data) in the hashed texture. One benefit is that a dynamic branch can be performed in the shader based on the stored bit, to completely bypass the hash function evaluations (h_0,h_1) and texture reads (Φ,*H*) on the undefined pixels. Of course, this acceleration is subject to the spatial granularity of GPU dynamic branching, but is often extremely effective.

Current graphics hardware lacks support for single-bit textures, so we pack each 4×2 block of domain bits into one pixel of an 8-bit luminance image. To dereference the bit, we perform a lookup in a 4×2×256 texture; next-generation hardware will have more convenient instructions for such bit manipulations.

If a non-sparse image is already defined over the domain, an alternate strategy is to *hide* the domain bit within this image, for instance in the least-significant bit of a color or alpha channel. Such a hidden bit is convenient to indicate the presence of sparse supplemental data beyond that in the normal image.

Position tag. When the data is very sparse, storing even a single bit per domain point may take significant memory. An alternative is to let each slot of the hash table include a tag identifying the domain position $\hat{p}$ of the stored data. Given a query point, we then simply compare it with the stored tag.

Encoding a position $\hat{p} \in U$ requires a minimum of $\log_2 u$ bits. In practice, we store the position tags in an image with *d* channels of 16 bits, thus allowing a domain grid resolution of $\bar{u}$ =64K. Such position tags are more concise than a domain bit image if the data density $\rho = n/u < 1/(16d)$.

Parameterized position hash. The set $h^{-1}(s) \subset U$ of domain points mapping to a slot *s* of the hash table has average size u/m. Our goal is to encode which one is the defined point $\hat{p} \in S$. Viewed in this context, the position tag seems unnecessarily long because its $\log_2 u$ bits select from among all *u* domain points; ideally we would like to store just $\log_2(u/m)$ bits. This lower bound is likely unreachable due to lack of structure in the hash function, but a terser encoding is possible.

We first experimented with storing a hash value from a third hash function $h_2(p)$. However, if this position hash stores *k* bits and has random distribution, the expected number of false positives over the domain is $2^{-k}u$, which is no better than storing position tags. The problem is that a globally defined hash function does not adapt to the set $h^{-1}(s)$ mapping in each slot. (It would be interesting to constrain the hash function *h* to avoid such false positives, but we could not devise an efficient way to do this during hash construction.)

Our solution is to store in each slot a tuple $\left(k, h_k(\hat{p})\right)$, where the integer $k \in \{1, \ldots, K\}$ locally selects a *parameterized hash* function $h_k(p)$ such that the defined point $\hat{p}$ has a hash value $h_k(\hat{p}) \in \{1, \ldots, R\}$ different from that of all other all domain points mapping to that slot. More precisely,

$$\forall p \in h^{-1}(s) \setminus \hat{p}, \quad h_k(p) \neq h_k(\hat{p}). \qquad (1)$$

The assignment of tuples $\left(k, h_k(\hat{p})\right)$ proceeds after hash construction as follows. We first assign $k = 1$ and compute $h_k(\hat{p})$ at all slots. For the few slots without a defined point $\hat{p}$, we just assign $h_k(\hat{p})$ =1. We then sweep through the full domain *U* to find the undefined points whose parameterized hash values (under *k*=1) conflict with $h_k(\hat{p})$, and mark those slots. We then make a second sweep through the domain to accumulate the sets $h^{-1}(s)$ for those slots with conflicts. Finally, for each such slot, we try all values of *k* to satisfy (1).

Unlike our perfect hash function *h*, we can let $h_k(p)$ be as random as possible since it is not used to dereference memory. For fast evaluation, we use `hk = frac(dot(p, rsqrt(p + k * c1)))`, set *K*=*R*=256, and store $\left(k, h_k(\hat{p})\right)$ as a 2-channel 8-bit image. We show in the appendix that the probability that at all *m* slots will find a parameter *k* satisfying (1) is near unity if the data density is no smaller than $\sigma \approx \frac{m}{u} \sim \frac{1}{800}$.

6. Filtering and blocking

The sparse data $D(p)$ can represent either constant attributes over discrete grid cells (e.g. sprite pointers, line coefficients, or voxel occupancy) or samples of a smooth underlying function (e.g. color or transparency). In this second case we seek to evaluate a continuous reconstruction filter, as discussed next.

With blocking. Our first approach is to enable native hardware bilinear/trilinear filtering by grouping pixels into blocks [Kraus and Ertl 2002]. An original domain of size $w=\overline{w}^d$ is partitioned into a grid of $\overline{u}^d=(\overline{w}/b)^d$ sample-bordered blocks of extent b^d. Each block stores $(b+1)^d$ samples since data is replicated at the block boundaries. If n blocks contain defined data, these pack into a texture of size $n(b+1)^d$. To reference the packed blocks, previous schemes use an indirection table with u pointers, for a total of $16u$-$24u$ bits. All but n of the pointers reference a special empty block. The block size that minimizes memory use is determined for each given dataset through a simple search.

Our contribution is to replace the indirection table by a hash function, which needs only ~4 bits per *defined* block. In addition, for the case of 2D unconstrained access, we must encode the defined blocks using either a domain bit or position hash, for a total of $4n+u$ or $20n$ bits respectively.

Table 1 shows quantitative comparisons of indirection tables and domain-bit hashes for various block sizes. As can be seen from the table, the hash offset table is much more compact than the indirection table and therefore encourages smaller block sizes. One limiting factor is that replication of samples along block boundaries penalizes small blocks. Another limiting factor for the 2D case is that unconstrained access requires encoding the data locations using domain bits (or a position hash). Nonetheless, we see an improvement over indirection tables of 30% in 2D and 42% in 3D. For the 2D case, one practical alternative is to hide the domain bits within the color domain image itself (Figure 7), in which case we require only 27.7KB with block size 3^2, or 41% less than the indirection table scheme.

Here is HLSL pseudocode for block-based spatial hashing:

```
float4 BlockedHashedTexture(point pf) {
  // pf is prescaled into range [0, ū ]
  point fr = frac(pf);
  point p = pf - fr;     // == floor(pf)
  if (use_domain_bit && !DecodeDomainBit(p))
    return undef_color;
  point h = ComputeHash(p);
  if (use_position_hash && !PositionMatchesHash(p,h))
   return undef_color;
  return tex(SHData, h + fr * bscale);
}
```

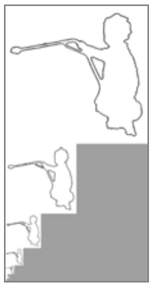
Without blocking. To remove the overhead of sample replication, our second approach performs explicit (non-native) filtering on an unblocked representation. The shader retrieves the nearest 2^d samples from the hashed data and blends them. Such explicit filtering is slower by a factor ~7 in 3D on the NVIDIA 7800 GTX. (The slowdown factor is only ~3 on an ATI X1800, perhaps because its texture caching is more effective on this type of access pattern.) Possibly this penalty could diminish in future architectures if filtering is relegated to general computation.

As shown in the first row of Table 1, the unblocked representation provides only a small improvement in 2D because we still require a position hash to encode data locations. However, in 3D where access is constrained, memory use is reduced by an impressive factor of 3. In fact, the unblocked hashed texture is only 16% larger than the defined data values.

Block size b	Blocks			Memory size (KB)				
	Number n	Density ρ	Size (KB)	Indir. table total	Spatial hash total	offset Φ	dom. bit	pos. hash
unblocked	11,868	3.7%	11.9	-	**30.7**	6.9	-	23.8
1^2	8,891	2.8%	35.6	676.3	120.1	4.4	80.1	-
2^2	2,930	3.7%	26.4	186.5	48.0	1.5	20.2	-
3^2	1,658	4.6%	26.5	98.0	36.7	1.2	9.0	-
4^2	1,107	5.5%	27.7	68.0	33.4	0.6	5.0	-
5^2	817	6.3%	29.4	55.4	**33.2**	0.5	3.2	-
6^2	655	7.3%	32.1	50.1	34.8	0.4	2.3	-
7^2	532	8.1%	34.0	47.2	36.1	0.3	1.7	-
8^2	460	9.1%	37.3	47.3	38.8	0.3	1.3	-
9^2	392	9.9%	39.2	**47.1**	40.5	0.2	1.0	-
10^2	339	10.4%	41.0	47.5	42.1	0.2	0.8	-
unblocked	4,500K	0.4%	13,723	-	**15,698**	1,976	-	-
1^3	2,266K	0.2%	54,396	3,275,622	55,382	986	-	-
2^3	563K	0.4%	45,596	448,249	**45,869**	273	-	-
3^3	250K	0.6%	47,952	167,958	48,104	152	-	-
4^3	140K	0.8%	52,561	102,892	52,650	89	-	-
5^3	89K	1.0%	57,952	83,797	58,005	53	-	-
6^3	62K	1.2%	63,873	**78,874**	63,910	37	-	-
7^3	46K	1.4%	69,955	79,485	69,983	28	-	-

Table 1: Comparison of memory usage for an indirection table and a spatial hash for different block sizes, on the datasets from Figure 7 and Figure 8. Optimal memory sizes are shown in bold.

Mipmapping. Defining a traditional mipmap pyramid over the packed data creates filtering artifacts even in the presence of blocking because the coarser mipmap levels incorrectly blend data across blocks. Our solution is as follows. We compute a correct mipmap over the domain, and arrange all mipmap levels into a flattened, broader domain using a simple function, as shown inset for the 2D case. Then, we construct a spatial hash on this new flattened domain (either with or without blocking). At runtime, we determine the mipmap LOD using one texture lookup [Cantlay 2005], perform separate hash queries on the two nearest mipmap levels, and blend the retrieved colors.



Native hardware mipmap filtering would be possible by assigning two mipmap levels to the packed texture. However, correct filtering would require allocating $(b+3)^d$ samples to each block (where b is odd) so it would incur a significant overhead. For instance, blocks of size b=5 in 2D would need $(5+3)^2+(3+1)^2=80$ samples rather than $(5+1)^2=36$ samples.

7. Applications and results

We next demonstrate several applications of perfect spatial hashing. All results are obtained using Microsoft DirectX 9 and an NVIDIA GeForce 7800 GTX with 256MB, in an 800^2 window.

7.1 2D domains

Vector images. Several schemes embed discontinuities in an image by storing vector information at its pixels [e.g. Sen et al 2003; Ramanarayanan et al 2004; Sen 2004; Tumblin and Choudhury 2004; Ray et al 2005; Tarini and Cignoni 2005; Qin et al 2006]. These schemes allocate vector data at all pixels even though discontinuities are usually sparse. They reduce memory through coarse quantization and somewhat intricate encodings.

Spatial hashing offers a simple, compact solution useful in conjunction with any such scheme – whether vector data is implicit or parametric, linear or higher-order, and with or without corners.

To demonstrate the feasibility and performance of the hash approach, we implement a representation of binary images with piecewise linear boundaries. For each square cell of the domain image, we store two bits b_1,b_2. Bit b_1 is the primary color of the cell, and bit b_2 indicates if any boundary lines pass through the cell. If b_2=1, the shader accesses a hashed texture to retrieve the coefficients $a_i, b_i, c_i,\ i = 1,2$ of two oriented lines passing through the cell, $l_i(x,y) = a_i x + b_i y + c$, where *x*,*y* are cell-local coordinates. The binary color at (x,y) is simply defined as

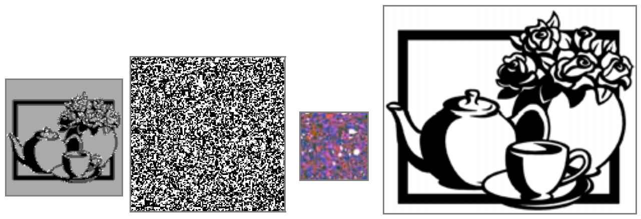
$$b_1 \text{ xor } \left(l_1(x,y) > 0 \ \wedge\ l_2(x,y) > 0\right).$$

We pack the 2 bits per pixel of the domain image as 2×2 blocks into individual pixels of an 8-bit image, and pack the hashed set of line coefficients as two RGB 8-bit images. For the example in Figure 4, given a 256^2 vector data image of 393KB, we create a 128^2 domain image of 16 KB, a hash table of 68KB, and an offset table of 7 KB, for a total of 91KB, or only 11 bits/pixel – quite nice for a resolution-independent representation with such a large number of discontinuities. Spatial hashing of a pinch function [Tarini and Cignoni 2005] could further reduce storage.

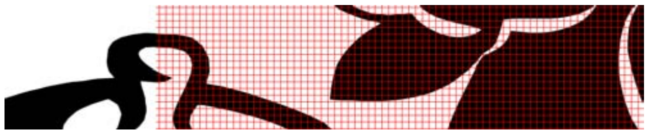
We implement antialiasing as in [Loop and Blinn 2005]. The complete shader, including hashing, takes 40 instructions. One key benefit of our approach is that dynamic branching on the domain bit b_2 lets the shader run extremely quickly on pixels away from the boundaries. For those pixels near discontinuities, the shader makes a total of 5 texture reads: the packed domain bit, an unpacking decode table, the hash offset value, and two triples of line coefficients. The image in Figure 4 renders at 461 Mpix/sec (i.e. 720 frames/sec at 800^2 resolution).

Figure 5 shows two additional examples. The outlines of some 81 characters from the "Curlz" font are converted to vector data in a 1024^2 image. The wide spacing between characters is necessary for the sprite application presented later in Figure 6. Spatial hashing reduces storage from 6.3MB to 495KB (3.8 bits/pixel). The tree example is vectorized in a 512^2 image, and is reduced from 1.6MB to 254KB. Note that these examples have a greater density of discontinuities than would likely be found in many practical applications.

Our prototype can easily be generalized to represent quadratics or cubics instead of lines, or to represent thick vector lines (in which case domain bit b_1 is unnecessary).

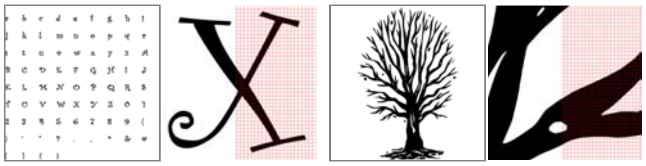


Domain image 128^2×8bits   Hash table *H* 106^2×2×24bits   Offset Φ 59^2×16bits   Rendered image (91KB) (720 fps)



Close-up showing the image cell structure.

Figure 4: Spatial hashing of sparse discontinuity vectors in a 256^2 image. (The hash table stores line coefficients, which we visualize using the discontinuity shader.)

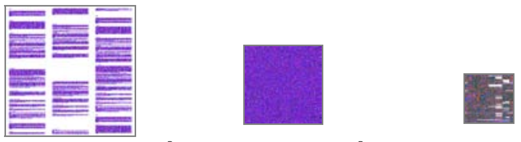


495KB (690 fps)   Close-up (820 fps)   254KB (760 fps)   Close-up (700 fps)

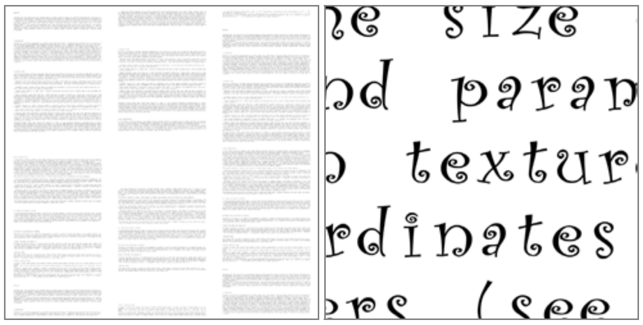
Figure 5: Two additional examples, with close-up views.

Texture sprites. Sprites are high-resolution decals instanced over a domain using texture indirection [Lefebvre and Neyret 2003]. For example they can be used to place character glyphs on a page. We use spatial hashing to compactly store such sprite maps.

In Figure 6 we store pointers to two sprites per domain grid cell. The sprites themselves are represented using hashed discontinuity images as described previously, so this example demonstrates two nested spatial hashes. Even for this relatively dense sprite map, storage is compact and rendering performance is excellent.



Input sprite map (u=512^2) 2097KB   Hash table (m=313^2) 784KB   Offset table (r=200^2) 80KB



Final rendering (225 fps)   Close-up (213 fps)

Figure 6: Spatial hashing of a sprite map. The sprites are the font characters stored in the image of Figure 5, so here we use two nested spatial hashes.

Alpha channel compression. In images with alpha masks, most alpha values are either 0 or 1, and only a small remaining subset is fractional. We pack this sparse subset in a hashed texture, which is blocked to support native bilinear filtering.

In the example of Figure 7, we use an R5G5B5A1 image where the one-bit alpha channel is 1 to indicate full opacity, the color (0,0,0,0) is reserved for full transparency, or else the fractional alpha value is placed in the spatial hash. Storage for the alpha channel is reduced from 8 to 1.7 bits per pixel (including the 1 bit alpha channel). Rendering rate is about 1170 frames/sec. Here is HLSL pseudocode:

```
float4 AlphaCompressedTexture(float2 p) : COLOR {
  float4 pix = tex2D(STexture, p); // R5G5B5A1
  if (pix.a == 1) {        // fully opaque pixel, no-op
  } else if (!any(pix)) { // fully transparent pixel, no-op
  } else {               // fractional alpha in hash table
    pix.a = BlockedHashedTexture(p*scale);
  }
  // optimized: if (dot(1-pix.a,pix.rgb)) pix.a = Blocked...
  return pix;
}
```

For traditional R8G8B8 images, an alternative is to use a coarse 2-bit domain image at the same resolution as the hashed alpha blocks, yielding overall alpha storage of 0.83 bits per pixel and a rendering rate of 830 frames/sec.

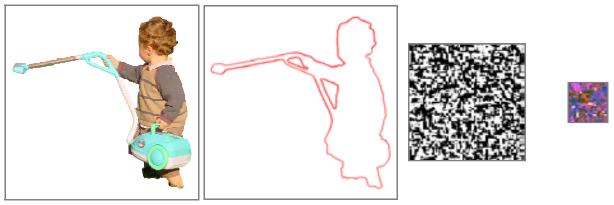


Figure 7: Alpha channel compression using spatially hashed fractional alpha values, with block size b=3.

7.2 3D domains

3D-parameterized surface texture. Octree textures [Benson and Davis 2002; DeBry et al 2002] store surface color as sparse volumetric data parameterized by the intrinsic surface geometry. Such volumetric textures offer a simple solution for seamless texturing with nicely distributed spatial samples. Octree traversal involves a costly chain of texture indirections, although this cost can be mitigated by larger N^3-trees [Lefebvre et al 2005].

Perfect hashing provides an efficient packed representation. We use a block-based hash for native trilinear filtering. As discussed in Section 6, the example of Figure 8 requires a total of 45.9 MB at optimal block size b=2, compared to 78.9 MB for an indirection table at optimal block size b=6. The hashed representation renders at 530 fps, or 300 fps if including mipmapping.

For the same model, an octree constructed for nearest sampling occupies 18.0 MB and achieves only 180 fps. However, for a fair comparison, an unblocked spatial hash also constructed for nearest sampling takes just 7.5 MB and renders at 370 fps. Thus, a spatial hash can be more than twice as compact than an octree, which is in turn generally more compact than N^3-trees with N>2.

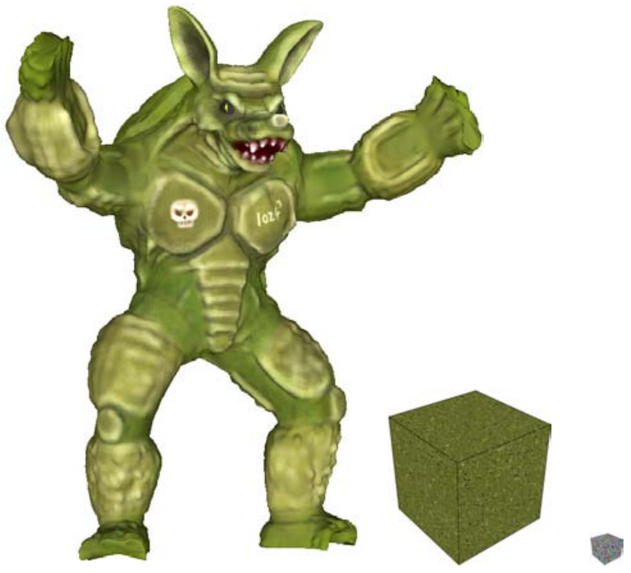


Figure 8: Spatial hashing of 3D surface data. Block size b=2.

3D painting. A 3D hashed texture is well suited for interactive painting, because it is compact enough to uniformly sample a surface at high resolution, yet efficient enough for real-time display and modification. One advantage over adaptive schemes like octrees is that, just as in traditional 2D painting, we need not update any pointer-based structures during interaction.

One complication is that graphics systems do not yet allow efficient rendering into a 3D texture. Thus, to enable fast modification of the hashed data on current systems, we extend our hash function to map 3D domains to 2D textures. This simply involves redefining M_0, M_1 as 2×3 matrices of the form

$$\begin{pmatrix} 1 & 0 & c_1 \\ 0 & 1 & c_2 \end{pmatrix}, \text{ where } c_1, c_2 \text{ are coprime with both } \bar{m} \text{ and } \bar{r}.$$

Note that this corresponds to displacing 2D slices of the volume data, and thus retains coherence along only 2 of the 3 dimensions.

We store position tags along with the hashed data. Then, during painting, we perform rasterization passes over the 2D hashed texture. For each pixel, the shader compares the paintbrush position with the stored position tag and updates the hashed color appropriately. After painting is complete, the hashed 2D data could be transferred to a block-based 3D hash or to a conventional texture atlas.

For the horse example of Figure 9, we hash a 2048^3 volumetric texture into a 2437^2 image. The hashed texture takes 17.8MB, the position tags 35.6MB, and the 1074^2 offset table 2.3MB, for a total of 55.7MB. Painting proceeds at a remarkable rate of 190 frames/sec, as demonstrated in the accompanying video. Note that since we are modifying the full hashed data each frame, the paintbrush can be *any image of arbitrary size* without any loss in performance. The data in Figure 8 was also created this way.

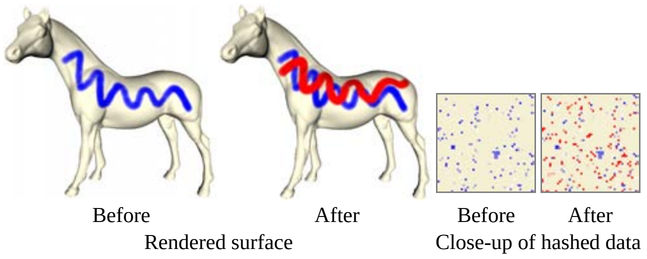


Figure 9: Surface painting with a 3D→2D perfect hash function.

3D simulation. We can also let a finite-element simulation modify the surface data, again as a rasterization pass over the 2D hash table. Here the elements are voxels intersecting the surface. As in [Lefebvre et al 2005], for each element we store 2D pointers to the 3D-adjacent elements (some of which may be undefined). Figure 10 shows two frames from the accompanying video, in which fluid flow is interactively simulated on a 256^3 sparse grid at 103 frames/sec. (Rendering itself proceeds at 1527 frames/sec.)

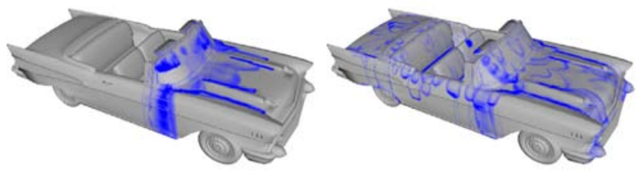


Figure 10: Physical simulation on hashed voxelized surface data.

Application	Dataset	Block size b	Domain grid u	Defined data n	Data density $\rho=n/u$	Hash table m	Offset table r	Offset table bits/n		Construction (sec)		Num. GPU instr.	Runtime (frames/sec) dep. on hash coherence			Opt. coh.
								Theor.	8-bit Φ	Fast	Opt. r		No coh.	No opt.	Opt.	$\tilde{\mathcal{N}}_H$
Vector image	teapot	-	256^2	11,155	17.0%	106^2	59^2	4.37	4.99	0.2	0.9	40	694	725	729	.278
Vector image	font	-	1024^2	33,562	3.2%	184^2	123^2	7.21	7.21	0.8	2.3	40	636	665	689	.216
Vector image	tree	-	512^2	28,462	10.9%	169^2	91^2	4.66	4.66	0.5	2.0	40	675	727	760	.232
Sprites	text	-	512^2	96,174	36.7%	313^2	200^2	6.65	6.65	4.5	7.9	121	208	237	272	.419
Alpha compr.	boy	3	189^2	1,658	4.6%	41^2	24^2	4.17	5.66	0.0	0.0	23	1147	1172	1169	.290
3D texture	armadillo	2	512^3	562,912	0.4%	83^3	45^3	3.40	3.89	7.0	350	9	439	536	540	.113
Painting	horse	-	2048^3	5.9M	0.07%	2437^2	1074^2	3.15	3.15	27.7	1400	10	270	354	403	.055
Simulation	car	-	256^3	142,829	0.9%	381^2	170^2	3.24	3.24	1.7	15	11	1221	1478	1527	.093
Collision det.	gargoyle	6	171^3	94,912	1.9%	46^3	26^3	3.33	4.44	0.6	14	38	127	134	143	.153
-	random	-	2048^2	100,000	2.4%	318^2	136^2	2.96	2.96	0.2	20	-	-	-	-	-
-	random	-	512^3	1.0M	0.7%	101^3	52^3	2.95	3.37	6.0	830	-	-	-	-	-

Table 2: Quantitative results for perfect hashing.

Surface collision detection. Several GPU schemes find possible inter-surface collisions using image-space visibility queries [e.g. Govindaraju et al 2004]. Because the queries test for separability along the view direction, hierarchical decomposition is useful to reduce false positives. And, overcoming image-precision errors involves a Minkowski dilation of each primitive.

A spatial hash enables an efficient *object-space* framework for conservative collision detection – we discretize two surfaces S_A,S_B into voxels and intersect these. Let $Vox_g(S)$ be the sparse voxels of size g that intersect surface S. Rather than directly computing $Vox_g(S_A)\cap Vox_g(S_B)$, we test the voxel centers of one surface against a dilated version of the voxels from the other surface. That is, we compute $Vox_g(S_A+S_e)\cap Centers(Vox_g(S_B))$ where "+" denotes Minkowski sum, S_e is a sphere of voxel circumradius $e = g\sqrt{3}/2$, and *Centers* returns the voxel centers.

We store the sparse voxels $Vox_g(S_A+S_e)$ as a blocked spatial hash, and the points $Centers(Vox_g(S_B))$ as a 2D image. At runtime, given a rigid motion of S_B, we apply a rasterization pass over its stored voxel centers. The shader transforms each center and tests if it lies within a defined voxel of the spatial hash of S_A. The intersecting voxels of S_B provide a tight conservative approximation of the intersection curve. In addition to using a traditional occlusion query, we can render the intersecting voxels by letting the 2D image of S_B be defined as a second (3D→2D) spatial hash.

In Figure 11, the gargoyle surface S_A is voxelized on a 1024^3 grid. Its spatial hash consists of a 46^3 hash table containing blocks of 6^3 bits, a 26^3 offset table, and a 171^3 domain-bit volume, for a total of 3.3MB. (By comparison, an indirection table would require 6.3MB, with optimal block size 13^3.) The voxel center points of the horse surface S_B are stored in a 961^2 image of 5.5MB.

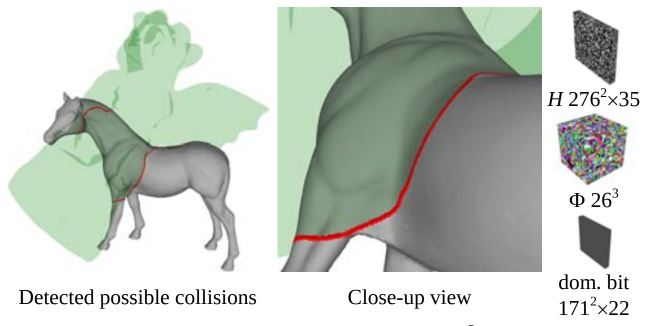


Figure 11: Surface collision detection on 1024^3 grids at 143 fps. Red voxels include possible inter-surface intersections; two tables (H and domain-bit) are compacted by 8X along one axis to create 8-bit textures.

8. Discussion

Table 2 summarizes quantitative results for perfect spatial hashing in the various applications.

Memory size. The offset table sizes correspond to about 3 to 7 bits per data entry, which is good considering the theoretical lower-bound of 1.44 bits. Because the 8-bit offsets are a poor encoding for small datasets where $\bar{m} < 256$, we also include the theoretical bit rate if one were allowed coordinates with $\lceil \log_2 \bar{m} \rceil$ bits. Our practical datasets have sparsity structure in the form of clusters or lower-dimensional manifolds. The two high numbers (for the font and sprite maps) are due to repetitive patterns in this data sparsity. The last two table rows show hashing results on artificial randomly distributed data. Such random data allows a more compact hash structure, with just under 3 bits per data.

Construction cost. The two preprocess times are for fast construction and for binary search optimization over the offset table size. (All other results assume optimized table sizes.)

Runtime coherence. We compare rendering performance using perfect hashes constructed with (1) random matrices M_0,M_1 to simulate pseudorandom noncoherent hash functions, (2) identity matrices M_0,M_1 but no coherence optimization, and finally (3) coherence optimization. Thus, the improvement 1→2 is due to the coherent access to the offset table, and the improvement 2→3 is due to the coherence optimization of Section 4.4. The optimization creates small coherent clusters, as can be visualized in the hash tables (e.g. Figure 1), and as reported by the metric $\tilde{\mathcal{N}}_H$ in the last column. Actual speedup is unfortunately not commensurate. Better knowledge of the inner workings of the GPU texture cache could allow for improved coherence optimization.

Limitations. One main constraint in our current hashing scheme is that the data sparsity structure must remain static, since the perfect hash construction is a nontrivial operation. However, the data values themselves can be modified efficiently as demonstrated in the 3D painting and simulation applications.

Like [Fox et al 1992], our construction algorithm does not provide a guarantee on the size of the offset table necessary to create a perfect hash. In theory the size could exceed $O(n)$. However, in practice we attain perfect hashes whose sizes range from $3n$ to $7n$ bits, where the good case is randomly distributed data and the bad case is data with highly repetitive sparsity structure.

Our current hash structure stores data at a spatially uniform resolution. We could support adaptive resolution by introducing a mipmapped indirection table with sharing of blocks between levels as in [Lefohn et al 2006]. Spatial hashing would then be used to compress this mipmapped indirection table.

Alternatives. Prior GPU-based spatial data structures include indirection tables, N^3 trees, and octrees [Lefohn et al 2006]. Even though the octree is generally the most concise representation in this spectrum, our perfect hash is yet more concise as reported in Section 7.2, so memory savings is a key benefit of spatial hashing. On the other hand, indirection tables and trees currently offer more flexibility for dynamic update and adaptive resolution.

9. Summary and Future work

We have extended perfect hashing to multidimensional domains and hash tables, and shown that the multidimensional setting permits a simple hash function that is extremely harmonious with current GPU architecture. We design and optimize the hash to exploit the access coherence common in graphics processing, and explore several techniques to encode data sparsity. We have shown that perfect hashing is practical in many applications. In particular, we demonstrate compact vector images and sprite maps, real-time unconstrained painting of surface models at 2048^3 resolution, and real-time collision detection at 1024^3 resolution using a compact representation.

Areas for future work include:

- Explore how best to compress hashed data for serialization.
- Further study the use of hashing to encode boundary conditions in simulation.
- Consider higher-dimensional sparse domains such as configuration spaces and time-dependent volumetric data.
- Extend hashing to support efficient dynamic updates and adaptive resolution.

References

BENSON, D., AND DAVIS, J. 2002. Octree textures. *ACM SIGGRAPH*, 785-790.

BLYTHE, D. 2006. The Direct3D 10 system. *ACM SIGGRAPH.*

BRAIN, M., AND THARP, A. 1990. Perfect hashing using sparse matrix packing. *Information Systems*, 15(3), 281-290.

CANTLAY, I. 2005. Mipmap-level measurement. *GPU Gems II*, 437-449.

CZECH, Z., HAVAS, G., AND MAJEWSKI, B. 1997. Perfect hashing. *Theoretical Computer Science* 182, 1-143.

DEBRY, D., GIBBS, J., PETTY, D., AND ROBINS, N. 2002. Painting and rendering on unparameterized models. *ACM SIGGRAPH*, 763-768.

FOX, E., HEATH, L., CHEN, Q., AND DAOUD, A. 1992. Practical minimal perfect hash functions for large databases. *CACM* 33(1), 105-121.

FREDMAN, M., KOMLÓS, J., AND SZEMERÉDI, E. 1984. Storing a sparse table with O(1) worst case access time. *JACM* 31(3), 538-544.

GAEDE, V., AND GÜNTHER, O. 1998. Multidimensional access methods. *ACM Computing Surveys* 30(2), 170-231.

GOVINDARAJU, N., LIN, M., AND MANOCHA, D. 2004. Fast and reliable collision culling using graphics hardware. *Proc. of VRST*, 2-9.

HO, Y. 1994. Application of minimal perfect hashing in main memory indexing. Masters Thesis, MIT.

KRAUS, M., AND ERTL, T. 2002. Adaptive texture maps. *Graphics Hardware*, 7-15.

LEFEBVRE, S., AND NEYRET, F. 2003. Pattern based procedural textures. *Symposium on Interactive 3D Graphics*, 203-212.

LEFEBVRE, S., HORNUS, S., AND NEYRET, F. 2005. Octree textures on the GPU. In *GPU Gems II*, 595-613.

LEFOHN, A., KNISS, J., STRZODKA, R., SENGUPTA, S., AND OWENS, J. 2006. Glift: Generic, efficient, random-access GPU data structures. *ACM TOG* 25(1).

LOOP, C., AND BLINN, J. 2005. Resolution-independent curve rendering using programmable graphics hardware. *SIGGRAPH*, 1000-1009.

MEHLHORN, K. 1982. On the program size of perfect and universal hash functions. *Symposium on Foundations of Computer Science*, 170-175.

MIRTICH, B. 1996. Impulse-based dynamic simulation of rigid body systems. PhD Thesis, UC Berkeley.

ÖSTLIN, A., AND PAGH, R. 2003. Uniform hashing in constant time and linear space. *ACM STOC*, 622-628.

QIN, Z., MCCOOL, M., AND KAPLAN, C. 2006. Real-time texture-mapped textured glyphs. *Symposium on Interactive 3D Graphics and Games*.

RAMANARAYANAN, G., BALA, K., AND WALTER, B. 2004. Feature-based textures. *Eurographics Symposium on Rendering*, 65-73.

RAY, N., CAVIN, X., AND LÉVY, B. 2005. Vector texture maps on the GPU. Technical Report ALICE-TR-05-003.

SAGER, T. 1985. A polynomial time generator for minimal perfect hash functions. *CACM* 28(5), 523-532.

SCHMIDT, J., AND SIEGEL, A. 1990. The spatial complexity of oblivious k-probe hash functions, *SIAM Journal on Computing*, 19(5), 775-786.

SEN, P., CAMMARANO, M., AND HANRAHAN, P. 2003. Shadow silhouette maps. *ACM SIGGRAPH*, 521-526.

SEN, P. 2004. Silhouette maps for improved texture magnification. *Graphics Hardware Symposium*, 65-73.

TARINI, M., AND CIGNONI, P. 2005. Pinchmaps: Textures with customizable discontinuities. *Eurographics Conference*, 557-568.

TESCHNER, M., HEIDELBERGER, B., MÜLLER, M., POMERANETS, D., AND GROSS, M. 2003. Optimized spatial hashing for collision detection of deformable objects. *Proc. VMV*, 47-54.

TUMBLIN, J., AND CHOUDHURY, P. 2004. Bixels: Picture samples with sharp embedded boundaries. *Symposium on Rendering*, 186-194.

WINTERS, V. 1990. Minimal perfect hashing in polynomial time, *BIT* 30(2), 235-244.

Acknowledgments

We thank Dan Baker and Peter-Pike Sloan for their assistance with Microsoft DirectX.

Appendix: Parametrized hash probability

In this section we derive the probability of finding a set of valid parameterized position hashes as described in Section 5.

While the map $S \to H$ is a rare perfect hash, the map $U \to H$ from the full domain has random distribution over the hash table, which we have verified empirically. Thus, the probability distribution for the number X of entries $h^{-1}(s) \subset U$ mapping to a particular table slot is a binomial distribution

$$\Pr\left(X = x\right) = b\left(x;u,m\right) = \binom{u}{x}\left(\tfrac{1}{m}\right)^x (1-\tfrac{1}{m})^{u-x} .$$

Since $u/m \gg 5$, this binomial is well approximated by a normal distribution with mean $\mu = u/m$ and variance $\sigma^2 = u/m$.

Next let us analyze the probability of finding a valid parameterized hash for a slot containing x random entries. For each parameter $k \in \{1,\dots,K\}$, the probability that the hash value $h_k(\hat{p}) \in \{1,\dots,R\}$ at the defined point $\hat{p}$ is different from that of all other x–1 points is $(1–1/R)^{x-1}$. Therefore, the probability that at least one of the K parameters succeeds is

$$\Pr(x;u,m) = 1 - \left(1-\left(1-\tfrac{1}{R}\right)^{x-1}\right)^K .$$

Combining these formulas, the probability that all m slots find valid parameterized hashes is

$$\Pr(u,m) \approx \left(\sum_{x=1\dots u} \tfrac{1}{\sqrt{2\pi}\sigma} e^{-(x-\mu)^2/2\sigma^2}\left(1-\left(1-\left(1-\tfrac{1}{R}\right)^{x-1}\right)^K\right)\right)^m .$$

For a typical table size m~256^2 and our default choice K=R=2^8, this probability is near 100% if the data density is greater than $\rho \approx m/u \sim 1/800$, and drops sharply below this point.