Eurographics Symposium on Rendering (2007)
Jan Kautz and Sumanta Pattanaik (Editors)

Compressed Random-Access Trees for Spatially Coherent Data

Sylvain Lefebvre[1] and Hugues Hoppe[2]

[1] REVES-INRIA, Sophia-Antipolis, France
[2] Microsoft Research, Redmond, WA, USA

Abstract

Adaptive multiresolution hierarchies are highly efficient at representing spatially coherent graphics data. We introduce a framework for compressing such adaptive hierarchies using a compact randomly-accessible tree structure. Prior schemes have explored compressed trees, but nearly all involve entropy coding of a sequential traversal, thus preventing fine-grain random queries required by rendering algorithms. Instead, we use fixed-rate encoding for both the tree topology and its data. Key elements include the replacement of pointers by local offsets, a forested mipmap structure, vector quantization of inter-level residuals, and efficient coding of partially defined data. Both the offsets and codebook indices are stored as byte records for easy parsing by either CPU or GPU shaders. We show that continuous mipmapping over an adaptive tree is more efficient using primal subdivision than traditional dual subdivision. Finally, we demonstrate efficient compression of many data types including light maps, alpha mattes, distance fields, and HDR images.

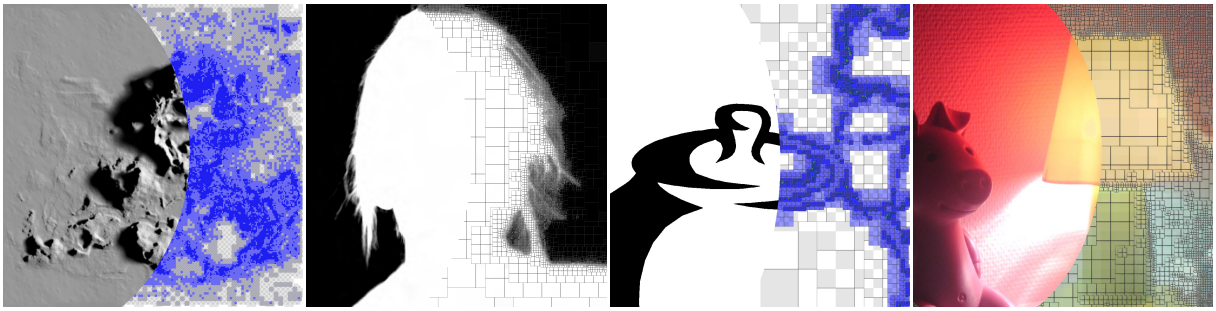


Figure 1: *Coherent data stored in a compact randomly accessible adaptive hierarchy with efficient mipmap filtering.*

1. Introduction

Spatial data in computer graphics is often very coherent. For example, distance fields are continuous, light maps are smooth except at shadow boundaries, alpha mattes are constant except near silhouettes, and high-dynamic-range (HDR) images have broad regions of similar luminance. Effective data compression permits more image content to be stored within a given memory budget.

There is a large body of work on compressing coherent data, particularly in the context of images. However, most compression schemes such as JPEG2000 involve sequential traversal of the data for entropy coding, and therefore lack efficient fine-grain random access. A fundamental challenge in rendering is that, while some input can be streamed sequentially (e.g. geometric primitives), the remaining data accesses are often random (e.g. projected texture maps, parallax occlusion maps, shadow maps).

Compression techniques that retain random access are more rare. A common approach is fixed-rate compression of image blocks, such as vector quantization [YFT80; NH92;BAC96] and the S3TC/DXT scheme widely available in hardware [MB98].

For the case of coherent data, traditional block-based compression has two drawbacks. First, blocks encode data at a single scale and therefore lack a good prediction model for low-frequency variations (e.g. linear ramps) that span across many blocks. Second, most schemes allocate a uniform bit-rate across the image, and thus lose information in high-detail regions, while over-allocating bits in low-detail regions.

Adaptive hierarchies such as wavelets and quadtrees offer both multiresolution prediction and spatial adaptivity [e.g. Sha93;Sam06]. Such tree structures have been applied to light maps [Hec90], distance fields [FPRJ00], point sets [RL01], octree textures [BD02;DGPR02], and irradiance [CB04]. Several techniques compress trees as reviewed in Section 2, but these techniques generally require sequential traversal and therefore give up efficient random access.

Contributions. We design a compressed tree representation that preserves fine-grain random access by using fixed-rate encoding for both the tree topology and its data. As shown in Figure 1, our scheme efficiently compresses coherent spatial data that is typically difficult for traditional block-based approaches: distance fields, light maps, alpha mattes and HDR images. To our knowledge this is the first

scheme to offer random access to a compressed adaptive hierarchy. Novel elements include:

- Use of a primal-subdivision tree structure for efficient mipmap filtering over an adaptive hierarchy.
- Replacement of node pointers by local offsets, and optimization of tree layout for concise offset encoding.
- A forested mipmap structure formed by replacing the coarsest levels by a mipmap and indirection table.
- Lossy compression of inter-level residuals using vector quantization (VQ) over broods in the tree, and extension to sparsely defined data.

The end result is an extremely simple data structure in which both topology and data are encoded in 8-bit fields. Because the codebook indices refer to data blocks rather than individual samples, the tree is one level shallower than the finest resolution data.

Our scheme is not intended for detailed color images, which result in dense trees and are thus better handled by block-based schemes. Nonetheless we show that these encode reasonably well.

Evaluation cost. One concern with tree structures is that they may increase memory bandwidth since each query involves several memory references. However, hierarchical access patterns are very coherent in practice [CB04], so most memory reads can be intercepted by memory caches. In fact, with sufficient query locality, the tree data only needs to be read from memory once (per pass), so the compactness of the tree is a bandwidth win, as analyzed in more detail in Section 8.

To mitigate the computational cost of traversing the tree, we collapse the coarsest levels to form a "forested mipmap" as described in Section 4.3.

Our compressed adaptive tree is appropriate for both CPU and GPU evaluation. We demonstrate trilinear mipmap evaluation within a pixel shader, and achieve real-time performance without specialized hardware.

2. Related work

Sequential traversals. Most schemes for compressing tree data consider a linear ordering of the tree nodes and encode a sequence of traversal codes and/or data residuals [e.g. Kno80;Woo84;Sam85;GW91;TS00]. Using a good prediction model and entropy coding, such pointerless representations achieve excellent compression, and have been applied to image wavelets [Sha93], isosurfaces [SK01], and point-based surfaces [BWK02]. However, these linear representations do not allow efficient random access to the spatial data, as they require decompression of the whole tree.

Location codes. An alternative is to store a spatially ordered list of the locations of leaf nodes [e.g. Gar82], but such a list does not permit hierarchical compression of the node data.

Random-access trees. Rusinkiewicz and Levoy [RL01] introduce a point-cluster hierarchy that supports view-dependent traversal. Their structure could be adapted to random point queries, but this would require decoding of many sibling nodes at each tree level.

Hierarchical vector quantization. Several schemes use VQ in a hierarchical setting. Gersho and Shoham [GS84] apply VQ to coarse-level amplitudes, and use these quantized values to guide the selection of codebooks at finer levels. Vaisey and Gersho [VG88] adaptively subdivide image blocks based on their variance, apply frequency transforms to the blocks, and use different VQ codebooks for different-sized blocks. The multistage hierarchical vector quantization (MSHVQ) of Ho and Gersho [HG88] is closest to our approach in that its stages perform downsampling, block-based VQ, and subtraction of the linearly interpolated reconstructions. It differs in that the VQ blocks do not form a regular tree structure, and adaptivity is only considered in the last stage with highest resolution. Tree-structured VQ [GG92] uses a decision tree to accelerate VQ encoding; another acceleration technique is hierarchical table lookup [CCG96].

Block-based schemes. Block-based data compression has been a very active area of research [e.g. SW03;SA05]. The latest graphics hardware supports several new block-based schemes [Bly06]. The most relevant to us is DXGI_FORMAT_BC4_UNORM (BC4U), a scheme for lossy compression of single-channel (grayscale) images; it is 4 bits/pixel like the original DXT1 scheme.

Adaptive random-access schemes. Other adaptive representations include indirection tables [KE02], page tables [LKS*06], quadtrees of images [FFBG01], B-trees of losslessly compressed blocks [IM06], and spatial hashing [LH06]. These schemes are able to adapt the spatial distribution of data samples, but do not focus on hierarchical compression of the data itself.

Unlike previous random-access compression schemes, our hierarchy exploits data prediction across resolutions, which is key to concisely encoding smoothly varying data.

3. Primal subdivision for efficient interpolation

3.1 Traditional dual subdivision

In a traditional region quadtree (dimension $d = 2$), nodes correspond to spatial *cells* that are properly nested across resolution levels, forming a dual-subdivision structure [ZS01]. An important drawback of a dual tree is that mipmap filtering becomes expensive when the tree is adaptive. Indeed, Benson and Davis [BD02] explain that mipmap interpolation requires a total of 3^d lookups per level. The problem is that pruned tree nodes must be interpolated from the next-coarser level (as shown by the red arrows in Figure 2a), and this interpolation requires a large support (e.g. point D requires values from nodes A, B, and C). Also, in pruned areas of the tree, the successive interpolations of the 3^d local nodes is equivalent to a multiquadratic B-spline, which is nicely smooth but expensive to evaluate.

3.2 Our primal subdivision approach

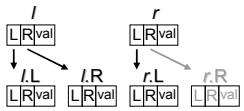
We instead associate tree nodes with the cell *corners* (e.g. as in [FPRJ00] and [LKS*06]), so that finer nodes have locations that are a superset of coarser nodes (though their values may differ). This corresponds to a primal-subdivision structure, and allows continuous interpolation

over an adaptive tree using only 2^d lookups per level (e.g. nodes B,C in Figure 2b). Moreover, refinement can terminate with simple multilinear interpolation when all 2^d local nodes are pruned. To our knowledge these advantages of primal trees have not been explained previously.

To represent a primal tree, we "slant" the tree structure as shown in Figure 2b. Thus, in 2D, the children of a node at location (x, y) have locations (x, y), $(x + 2^{-l}, y)$, $(x, y + 2^{-l})$, and $(x + 2^{-l}, y + 2^{-l})$ in level l. This slanting causes some subtrees to fall outside the input domain, as illustrated by the dangling links on the right boundary. (Dual trees have similar undefined subtrees if the domain size is not a power of two.) Fortunately, our data compression scheme (Section 5) is able to efficiently ignore such undefined data.

For completeness, we show pseudocode for continuous mipmap interpolation over a primal 1D tree, first on a complete tree:

```
class Tree {
  Tree L, R;
  float val;
}
```




```
float evaluate1D(Tree root, float x, float level) {
  return evaluate1DRec(x, level, root.L, root.R);
}

float evaluate1DRec(float x, float level, Tree l, Tree r) {
  float vC = interp(l.val, r.val, x);    // value at current level
  if (level<=0.0) return vC;
  float vF;                              // value at finer level
  if (x<0.5) {                           // select left/right subtree
    vF = evaluate1DRec((x-0.0)*2, level-1.0, l.L, l.R);
  } else {
    vF = evaluate1DRec((x-0.5)*2, level-1.0, l.R, r.L);
  }
  if (level>=1.0) return vF;
  return interp(vC, vF, level);          // blend two levels
}
```

Next, we eliminate the recursion and generalize the evaluation to an adaptive tree:

```
float evaluate1D(Tree root, float x, float level) {
  Tree l = root.l, r = root.r;
  float vl = l.val, vr = r.val;
  float vC = interp(vl, vr, x);          // value at current level
  for (;;) {
    if (!l && !r) return vC;             // early exit if pruned
    vm = interp(vl, vr, 0.5);            // default midpoint value
    if (x<0.5) {                         // select left/right subtree
      x = (x-0.0)*2; l = (l ? l.L : 0); r = (l ? l.R : 0); vr = vm;
    } else {
      x = (x-0.5)*2; l = (l ? l.R : 0); r = (r ? r.L : 0); vl = vm;
    }
    if (l) vl=l.val;                     // set values if not pruned
    if (r) vr=r.val;
    float vF = interp(vl, vr, x);        // value at current level
    if (level<=1.0) return interp(vC, vF, level);
    vC = vF;  level = level-1.0;
  }
}
```

The generalization of the evaluation procedure to higher dimensions is straightforward.

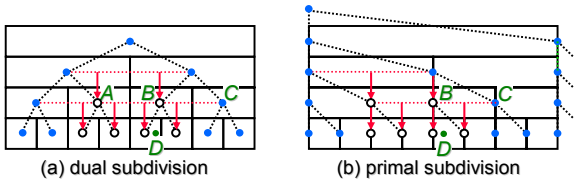


Figure 2: *Dual-subdivision and primal-subd. trees in 1D. In an adaptive tree, pruned nodes (hollow circles) must be interpolated from coarser values, and this is more complicated in dual subdivision.*

4. Compressed tree topology

We now describe our scheme for compressing the topology of the tree; Section 5 will address the compression of its associated data.

Terminology. A tree node with at least one child is an internal node; otherwise it is a leaf. The depth of a node is the length of the path to the root, so the root node has depth zero. Level l of the tree refers to all nodes at depth l. The tree height L is the maximum level. A *complete tree* has all its leaves at the same depth, and hence a total of 2^{dL} leaves. In an *arbitrary tree*, each node may have any number (0 to 2^d) of children. We focus on *full trees*, in which all internal nodes have a full set of (2^d) children. Note that a complete tree is always full, but not conversely.

4.1 Traditional tree data structures

We begin by reviewing structures for tree topology. In the simplest case, each node contains a data record and 2^d pointers to child nodes, any of which can be NULL:

```
struct Node {
  Data data;
  Node* children[2^d];   // NULL if the child is pruned
};
```

Assuming 32-bit pointers, the tree topology requires $4 \cdot 2^d$ bytes per node, with much wasted space at the leaf nodes.

Sibling tree. An improvement for full trees is to allocate sibling nodes contiguously (forming a *brood*), and to store a single pointer from the parent to the brood [HW91], thus reducing topology encoding to 4 bytes/node:

```
struct Node {
  Data data;
  Brood* brood;      // pointer to first child, or NULL
};

struct Brood {       // children nodes allocated consecutively
  Node nodes[2^d];
};
```

Autumnal tree. If pointers and data records have the same size, an even better scheme is to raise the data from leaf nodes into their parents, to form an autumnal tree [FM86]. Hence the Node structures are only allocated for internal nodes. A single bit identifies if a child is a leaf, and is often hidden within the pointer/data field. Tree topology is reduced to $4 \cdot 2^{-d} + 1/8$ bytes/node. For a quadtree, this is 1.125 bytes/node, much less than the sibling tree.

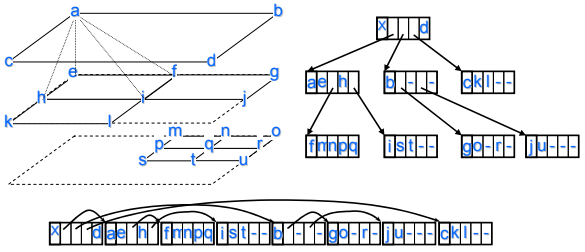


Figure 3: *Example of a primal autumnal tree and its memory packing. Dashes denote undefined data values.*

```
struct PointerOrData {
  bit leafchild;
  union {
    Node* pointer;              // if not leafchild
    Data data;                  // if leafchild
  };
};

struct Node {                   // only for internal nodes
  Data data;
  PointerOrData children[2^d];
};
```

4.2 Encoded local offsets

Even in an autumnal tree, the pointers remain the limiting factor for memory size. Our contribution is to replace such pointers by local offsets. Hunter and Willis [HW91] consider replacing absolute pointers by offsets, but define offsets from the start of the tree data structure. Instead, we define offsets locally, such that an offset of zero refers to memory just after the current node.

Starting with an autumnal tree, we replace each absolute 32-bit pointer by a local scaled offset encoded into 7 bits. The tree data values will also be encoded into 7 bits, so that the PointerOrData structure fits nicely in one byte.

We pack the nodes in memory in preorder as shown in Figure 3. At fine levels, parent nodes are close to all of their children. At coarser levels, the children become separated by their own subtrees, so the offset from the parent to its last child grows. Our idea is to encode each offset y into a 7-bit code $x \in [0,127]$ as $y = s_l x$ where s_l is a per-level scaling parameter. At finer levels where offsets are small, this encoding is wasteless with $s_l = 1$. At coarser levels where $s_l > 1$, if the desired offsets cannot be encoded exactly (i.e. are not a multiple of s_l), we leave some padding space between the subtrees.

We perform the packing in a fine-to-coarse order. For each level, having already packed the finer subtrees into memory blocks, we iteratively concatenate these subtree blocks after their respective parents such that they are addressable as encoded offsets from the parents — leaving padding space as needed. We exhaustively search for the integer scaling factor $s_l \in \{\lceil y_{\max}/127 \rceil \ldots y_{\max}\}$ that gives the best packing (where $y_{\max}$ is the largest offset). Table 1 shows some example results.

Another strategy would be to pack nodes in level-order (equivalent to breadth-first search). However, such ordering would give offsets that are larger and less predictable.

Level l	Num. nodes	Scaling s_l	Padding (bytes)
0	1	500	58
1	4	316	354
2	9	135	897
3	25	61	960
4	76	26	1016
5	202	10	555
6	486	5	0
7	1228	1	0
8	3218	1	0
9	*8322*	*1*	*0*

Table 1: *Result of offset encoding for the data in Figure 5. A forested mipmap replaces tree levels 0-4 (Section 4.3).*

4.3 Forested mipmap

Maintaining the coarsest levels as a tree structure has a number of drawbacks: (1) These levels are usually dense, so adaptivity is unnecessary; (2) The traversal of these coarse levels adds runtime cost; (3) Much of the padding space introduced by our offset encoding occurs there; (4) VQ compression is ineffective due to the small number of data nodes. For these reasons, we collapse the coarsest tree levels 0..4 to form a (non-adaptive) mipmap pyramid. At the finest of these pyramid levels, we also store a 17^2 indirection table with pointers to the resulting clipped subtrees. We call this overall structure a forested mipmap.

For the same example in Table 1, the forested mipmap results in a decrease of 1847 bytes. Overall the tree topology requires 0.36 bytes/node for this quadtree.

5. Compressed tree data

A benefit of a tree structure T is that data at finer levels can be predicted from coarser levels [e.g. BA83], in our case by simple multilinear interpolation. In the case of spatially coherent data, the residual differences tend to be small. Indeed, pruning of subtrees with near-zero residuals already offers significant data compression. In this section, we examine how to further compress the data residuals themselves.

5.1 Brood-based vector quantization

To support efficient random access, we compress the inter-level residuals using vector quantization. VQ is an approach that approximates a set of vectors by a small codebook, replacing each vector by an index into the codebook [GG92].

Specifically, we apply VQ to the blocks of data residuals associated with the *broods* of the tree. Recall that a brood is the set of 2^d children of a parent node. Each codebook index encodes the data residuals for these 2^d data samples. We use a codebook of 128 elements: each codebook index is 7 bits. For a complete quadtree, these indices correspond to storage of 1.75 bits per data at the finest level, or a total of 2.33 bits per data when accounting for all levels.

Thus, we create a new tree T' (which we call a VQ tree) in which each node data is a 7-bit codebook index. Because each index encodes a *block* of residual data, the VQ tree T' is one level shorter than the original data tree T. That is, each leaf node in T' stores data for 2^d samples in leaves of

T. Our association of a data block per tree node is similar to brick maps [CB04], although we store residuals and encode the blocks.

For a single-channel image, each residual vector contains 4 pixel residuals, and is therefore 4-dimensional. The 128 codevectors are a sparse sampling of this 4D space. (For a color image, the codevectors are an even sparser sampling of a 12D space.) Fortunately, there is significant clustering. In addition, the VQ tree is able to correct coarse-level errors in the finer levels. Because the tree encodes a cascade of small residuals, tree-based VQ yields significantly less error than VQ applied to a uniform grid of blocks [BAC96], as seen in Figure 13.

In 2D, we visualize the adaptive primal tree T' by outlining each node's Voronoi region, and showing its VQ data as 4 sub-squares. This nicely reveals both the tree structure and the data resolution.

Implementation. We compute a separate codebook at each tree level, using k-means clustering [Llo82], which converges to a local minimum of the summed intra-cluster variance. To help jump out of local minima, we use cluster teleportation as in [CAD04].

To reduce codebook sizes, we further quantize the codevector themselves to 8 bits per coordinate, and this is achieved as an easy extension to the k-means clustering algorithm.

Lastly, at the finest level L of the tree, we know that all nodes must be leaves. Therefore we omit the leafchild bit and instead use an 8-bit codebook index, together with a 256-entry codebook. This helps to improve compression accuracy at the finest level.

5.2 Extension of VQ to undefined data

As discussed in Section 3.2, subtrees sometimes extend beyond the defined domain, so there exist residual blocks for which some data values are undefined. We exploit the fact that we don't care about these residuals to reduce the error of the vector quantizer, as follows.

A not so well known property of k-means clustering is that it can be extended to partially defined data and still preserve its convergence properties [LFWV03]. The standard k-means clustering algorithm iterates between (1) assigning each vector to the closest cluster point, and (2) updating each cluster point as the mean of the vectors assigned to it. The generalization for partially defined data is to modify these steps to just ignore the undefined components of the input vectors, both when computing distances in step 1 and the centroid points in step 2.

5.3 Construction of adaptive VQ tree

We seek to construct a simplified VQ tree while bounding the L_∞ approximation error at all levels to a given threshold value τ. Because VQ compression is lossy, even the complete VQ tree may not satisfy the tolerance τ, and in that case our goal is to avoid introducing any further such errors.

At a high-level, the construction of the adaptive VQ tree T' involves three steps:

(1) Create a complete mipmap tree T of desired data values.

(2) Apply brood-based vector quantization to form a complete VQ tree T' of compressed residuals.

(3) Adaptively prune the tree T' subject to satisfying τ.

A useful extension is to reach a desired compression rate (e.g. 1.5 bits/pixel) by repeating step (3) using a binary search over τ.

A limitation of this algorithm is that it computes the per-level VQ codebooks using all block residuals in the complete tree, even though the final simplified tree will only contain a subset of these blocks. However, the effect should be minor since the pruned blocks have near-zero residuals. There is actually a complicated inter-dependence between the tree structure and the per-level codebooks. In particular, it is not a good idea to recompute new codebooks on the final simplified tree because this could result in approximation errors that exceed the tolerance τ.

We next discuss the 3 steps in more detail.

Mipmap construction. In fine-to-coarse order $l = L\text{-}1 \ldots 0$, we compute the desired values d_l at nodes of level l from those at level $l+1$ as a least-squares optimization $\min_{d_l} \left\| P_{l,l+1} d_l - d_{l+1} \right\|^2$ where the rows of matrix $P_{l,l+1}$ contain the multilinear interpolation weights (i.e. 0, 1, or powers of $\frac{1}{2}$, for our primal subdivision).

VQ compression of the complete tree. We process each level $l = 1 \ldots L$ of the tree in coarse-to-fine order as follows. We compute the predicted values $p_l = P_{l-1,l} a_{l-1}$ by multilinear interpolation of the approximated values a_{l-1} at the next-coarser level (with $a_0 = 0$). The residuals $r_l = d_l - p_l$ are compressed using brood-based VQ, resulting in compressed residuals $\tilde{r}_l$. Thus, the approximated values are $a_l = p_l + \tilde{r}_l$, and we clamp these to the signal range which is typically [0,1]. We also compute the signed approximation errors $e_l = a_l - d_l$.

Adaptive tree pruning. We process each level $l = L\text{-}1 \ldots 0$ of the VQ tree T' in fine-to-coarse order, looking to prune its leaves. The basic idea is to allow simplification as long as the accumulated approximation errors at all affected nodes in the original tree T do not exceed the tolerance, i.e. $\forall l, \|e_l\|_\infty \leq \tau$.

Because autumnal trees are full, the atomic simplification operation on T' is the removal of all 2^d leaf nodes in a brood. Thus, we need only consider a brood if all its subtrees have been pruned. Since each child in the brood (assumed at level l) contains a codebook index encoding a 2^d block of data, the simplification operation effectively removes a 2^d block of residual values $\tilde{r}^B \subset \tilde{r}_{l+1}$ in level $l+1$. We allow the brood to be removed if the subtraction of these residuals does not increase the approximation error (at any node in the original tree T) beyond the tolerance τ. Specifically, we compute the updated approximation errors e'_l by interpolating the subtracted residuals to each finer level $l' \geq l+1$ as $e'_{l'} = e_{l'} - P_{l',l+1} \tilde{r}^B$ and check if $\left\| e'_{l'} \right\| \leq \tau$.

Even within a level, the affected subtrees of residual blocks $\tilde{r}^B$ for different broods do overlap at their boundaries, so we visit the candidate broods in order of increasing residual norm $\|\tilde{r}^B\|$ to hopefully remove more smaller residuals than fewer larger ones.

5.4 Codebook sharing

Although codebooks are relatively compact (3 KB for a single-channel image), they need to be stored along with each image. On smooth data such as distance fields and light maps, we find that data residuals are extremely auto-similar across levels, so a *shared codebook* can be reused by all tree levels subject to an appropriate scaling factor, thus requiring only 1 KB. Specifically, we construct the shared codebook r_S at the finest level. Then for each coarser level we compute the scaling factor $\|r\|/\|r_S\|$ of the image residuals relative to the shared codebook, and apply this scaling to the codevectors.

In addition, for a class of images with similar content, we can design a *universal codebook* using a training image (Figure 9). Section 6.3 presents results using a universal codebook on distance fields. However we find that such a universal codebook does not extend well to dissimilar light maps or color images.

6. Applications and compression results

We demonstrate the efficiency of tree-based compression on several data types, including light maps, alpha mattes, distance fields, and high-dynamic-range images. Table 2 summarizes the results. All examples use forested mip-maps. Compression times range from 2 to 10 minutes, most of which is spent in VQ optimization. We manually selected target bit-rates; it would be desirable to automate this rate selection based on image content.

We compare memory sizes with BC4U and DXT1 (for grayscale and color images respectively) which are both 4 bits/pixel, as these are widely available representatives of block-based compression. The reported memory sizes include both the tree and codebook. We also compare with the block-based VQ scheme of [BAC96]. Please refer to our supplemental results for additional examples.

Note that many block-based schemes like BC4U and DXT1 require storage of separate (compressed) mipmap levels, which effectively raises storage cost to 5.33 bits/pixel for a full pyramid. In contrast, our tree representation directly includes all mipmap levels.

Another benefit of trees, which makes direct comparisons challenging, is that while the inter-level residuals are quantized (to 8 bits), the reconstructed signal is floating-point and attains greater accuracy at each finer level, as demonstrated with the distance function in Section 6.3.

6.1 Light maps

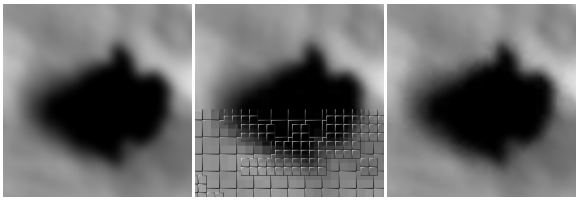
Our approach is especially well suited to light maps, as they contain both smoothly varying regions and sharp shadow boundaries. Figure 4 compares our method to BC4U compression which is 4 bits/pixel and has a PSNR of 48.8 dB. As shown in the graph of Figure 6, we reach this numerical accuracy at 2.2 bits/pixel. Moreover, Figure 4 shows that even at 1 bit/pixel (44.2 dB), our reconstruction is visually more faithful, with less noise and fewer dithering artifacts.

Dataset	Input		Compressed tree			BC4U/ DXT1	Beers [BAC96]
	Dim.	Size (KB)	Size (KB)	Bits/ pixel	PSNR (dB)	PSNR (dB)	PSNR (dB)
Land (lightmap)	1025^2	1051	135	1.03	44.2	48.8	40.6
Lady (matte)	1025^2	1051	139	1.06	52.8	53.0	44.6
Teapot (dist)	1025^2	131	8.7	0.07	-	-	-
Piggy (HDR)	513^2	3158	165	5.00	-	-	-
Monkey (matte)	1025^2	1050	95	0.72	51.2	51.5	43.8
Bull (dist)	1025^2	131	7.8	0.06	-	-	-
Desk (HDR)	644x874	6754	349	4.96	-	-	-
Atlas (lightmap)	1025^2	3151	269	2.05	49.6	52.5	41.4
Nefertiti (RGB)	513^2	790	65	1.97	37.8	36.3	33.9
Flowers (RGB)	513^2	790	116	3.52	31.0	29.6	28.2

Table 2: *Quantitative results including comparison with 4 bit/pixel BC4U or DXT1 compression and 2 bit/pixel VQ scheme of [BAC96].*

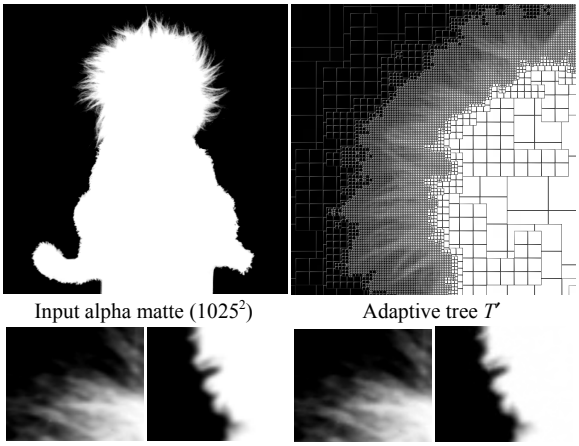
6.2 Alpha mattes

Alpha mattes often have only a small fraction of pixels with fractional alpha values. Our adaptive tree nicely skips all the solid regions of an alpha map, while precisely reproducing the smooth transitions between opaque and transparent areas. The alpha matte of Figure 5 is compressed by BC4U at 4 bits/pixel with an accuracy of 51.5 dB. We achieve a similar result at only 0.7 bits/pixel (see Figure 6).



Close-up of original (8 bpp) | Tree-compressed (1 bpp) | BC4U-compressed (4 bpp)

Figure 4: *Close-up on the light map of Figure 1.*



Input alpha matte (1025^2) | Adaptive tree T'

Close-ups of input | Close-ups of compressed

Figure 5:*Compression of alpha matte (0.7 bpp;51.2 dB)*

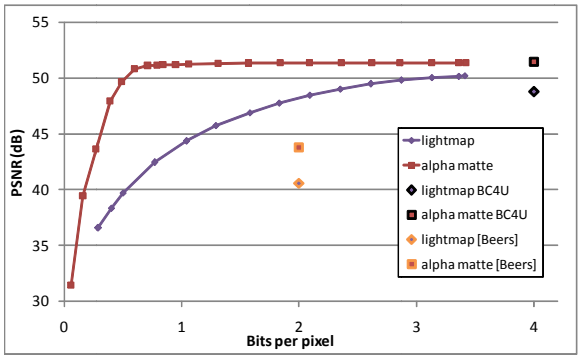


Figure 6: Rate-distortion curve for the light map and alpha matte examples. Isolated dots represent compression results with DXT1, BC4U, and [BAC96].

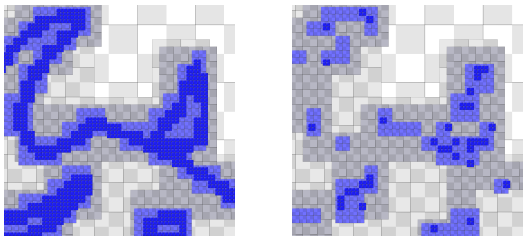
6.3 Adaptively sampled distance fields

Adaptively sampled distance fields are an elegant representation for vector outlines and 3D shapes [FPRJ00]. When applying tree compression to such a distance field, we are mainly interested in the shape of its zero set, so we modify the criterion used in the adaptive tree simplification. We let the tolerance τ be larger, but restrict the simplification to preserve the sign of the approximated data a_l everywhere.

Our scheme precisely and compactly encodes complex vector outlines. In the example of Figure 8, the compressed tree is 7.8 KB while the original parametric vector representation (with quadratic Bezier segments) is 3.2 KB. And, this result is obtained using a universal codebook trained on the image in Figure 9.

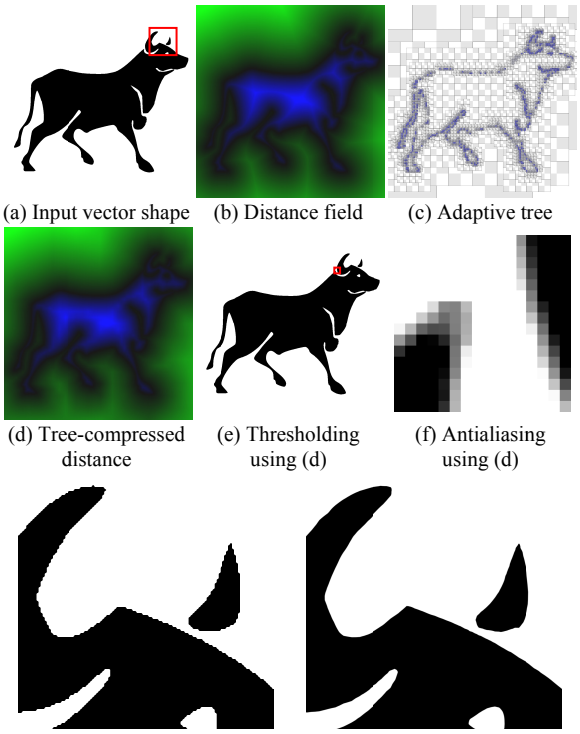
To measure the accuracy of our representation, we extract the zero isocurve of the compressed distance field, and measure the RMS distance between points on this curve and the original curve. The geometric PSNR is a remarkably high 82 dB, i.e. the error is not visible if the shape is rasterized at a resolution less than $10K^2$ pixels. Moreover, the distance representation permits high-quality antialiasing and magnification (Figure 8f-g), which would not be possible using a traditional binary image.

Figure 7 shows that a traditional binary-valued tree is much larger as it cannot exploit the smoothness of the vector outline, and hence requires more refinement.



Traditional binary-valued tree | Tree-compressed distance field

Figure 7: *A traditional quadtree on the binary image is much more refined than our tree compression of the distance field. (Both perfectly reproduce Figure 8a rasterized at 1025^2 resolution.)*



(a) Input vector shape | (b) Distance field | (c) Adaptive tree

(d) Tree-compressed distance | (e) Thresholding using (d) | (f) Antialiasing using (d)

(g) Thresholding on magnification of binary image and (d) resp.

Figure 8: *Representation of a vector shape (3.2KB) as a signed-distance field at 1025^2 resolution using a randomly accessible compressed tree (7.8KB), and its benefits for resolution-independent antialiasing and magnification. A binary image would require 131KB and would not magnify as a smooth shape outline as shown in (g).*

Training image | Using specialized codeb. (24.4KB, PSNR 79.2dB) | Using universal codeb. (21.2KB, PSNR 79.2dB)

Figure 9: *Training data used for universal codebook on distance fields, with negligible deterioration in resulting coding quality. PSNR numbers measure the geometric accuracy of the outline curves.*

6.4 High-dynamic-range images

Munkberg et al [MCHA06] and Roimela et al [RAI06] present DXT-like compression schemes for HDR images, using a luminance-hue factorization. Our idea is to capture the high-dynamic range variations using an aggressively compressed tree and to rely on an ordinary low-dynamic image to encode the remaining detail.

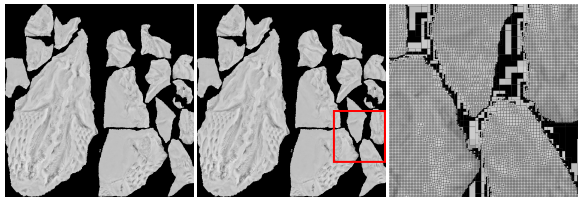
Specifically, we apply tree compression to the log(RGB) image to capture the HDR variation at only 1 bit/pixel. The benefit of encoding all 3 color channels rather than just luminance is that we reduce subsequent hue quantization artifacts. Then we subtract the compressed log(RGB) signal from the original log(RGB) image to create a low-dynamic-

range detail image, and quantize its channels (separately) to 8 bits. We compress this quantized detail image using ordinary DXT1 compression in 4 bits/pixel (Figure 12e). The overall representation uses 5 bits/pixel and compares favorably with the earlier result of [MCHA06] at 8 bits/pixel. We report rms errors in $\log_2$(RGB) space as in [XPH05]. Our tree-compressed result has few color quantization artifacts, even at extreme exposure levels.

6.5 Texture atlases

Texture atlases often contain charts separated by unused space (Figure 10). Our compressed tree ignores these undefined regions in two ways. First, the tree structure is adaptively pruned. Second, thanks to our sparse VQ approach (Section 5.2), the codebook quality is not impacted by the boundaries between the defined and undefined areas.

We modify the compression algorithm as follows. First, we extrapolate data outside the chart boundaries with a pull-push step [SSGH01]. We use this new image to compute the mipmap of desired values. Second, we remove from the mipmap tree T all sub-trees covering empty regions; the tree is no longer complete, and some residual blocks now contain undefined data values. This is handled by our modified VQ as described Section 5.2.



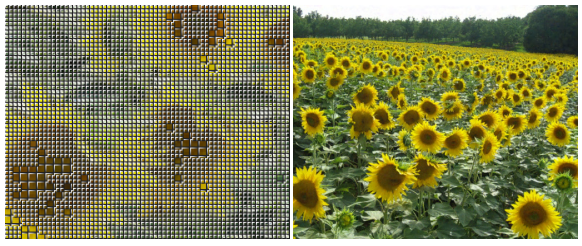
Original atlas (black is unused) | Tree-compressed (2.05 bpp; 49.6 dB) | Adaptive tree T' (close-up)

Figure 10: *Multi-chart texture atlas compression. Unused regions are omitted from the tree and ignored by VQ.*

6.6 Limitation: color images

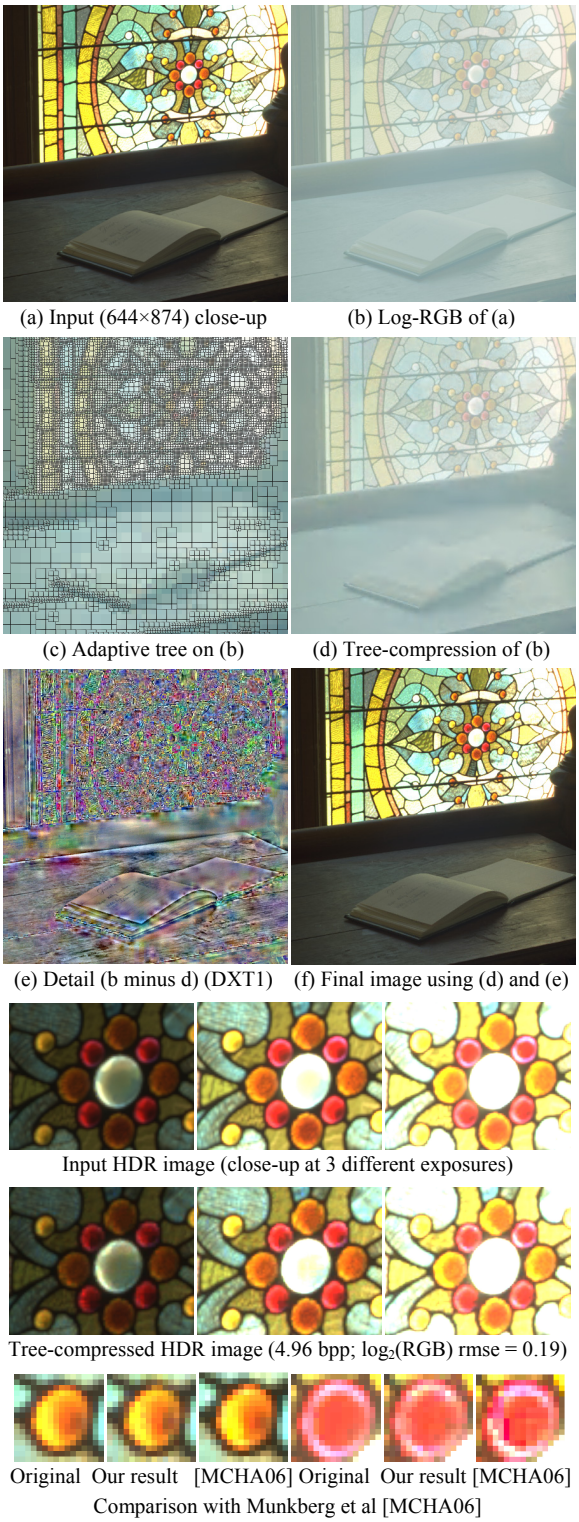
Tree compression can also be applied to color images. It is most effective on images with large *smooth* areas, such as in Figure 13 where we obtain a 2X memory savings compared to DXT1 compression, with slightly higher accuracy.

However, on more common images with uniform high-frequency detail, the resulting tree becomes too dense to be a significant benefit over traditional block-based approaches, as shown in Figure 11.



Adaptive tree T' (close-up) | Tree-compr. (3.52 bpp; 31 dB)

Figure 11: *Uniformly distributed detail creates a near-complete tree, which is not our desired scenario.*



(a) Input (644×874) close-up | (b) Log-RGB of (a)

(c) Adaptive tree on (b) | (d) Tree-compression of (b)

(e) Detail (b minus d) (DXT1) | (f) Final image using (d) and (e)

Input HDR image (close-up at 3 different exposures)

Tree-compressed HDR image (4.96 bpp; $\log_2$(RGB) rmse = 0.19)

Original | Our result | [MCHA06] | Original | Our result | [MCHA06]

Comparison with Munkberg et al [MCHA06]

Figure 12: *For an HDR image, aggressive tree compression in log(RGB) space (at 1 bit/pixel), with remaining detail represented as a low-dynamic-range DXT1 image (4 bits/pixel). In comparison, Munkberg et al [MCHA06] report rmse=0.25 at 8 bits/pixel.*

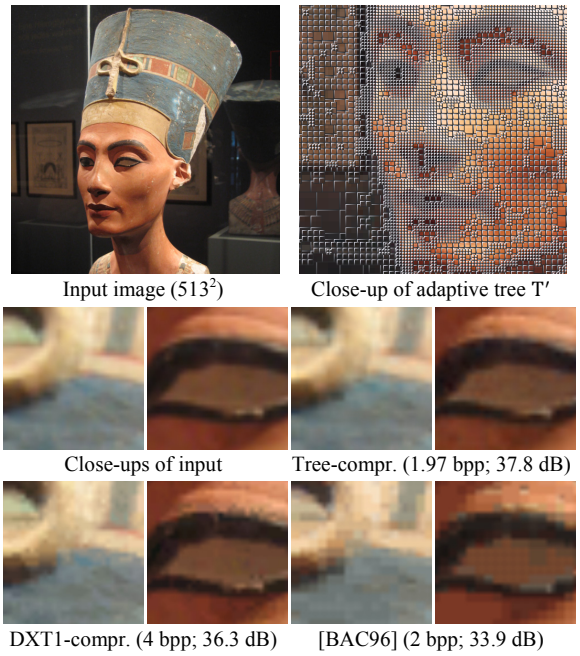


Figure 13: *Compression of a relatively smooth color image, compared with DXT1 compression and with uniform 2×2 block VQ using a 256-entry codebook.*

7. Tree evaluation

Our decompression scheme is easy to implement on a CPU. The following is pseudocode for trilinear evaluation at a point x and mipmap level l:

```
value Evaluate(point x, float l) {
  If l ≤ 4,
    return trilinearly filtered value from mipmap.
  Identify the square cell containing x at level 4
    (i.e. the starting level for trees in the forested mipmap).
  For each of the four corners of this cell:
    retrieve the root node address and value.
  Re-express point x in the cell's local coordinates.
  Loop:
    At point x, bilinearly interpolate the four node values.
      If --l < 1 or all four node addresses are NULL,
      return value at x lerp'ed with that in prior level using l.
      Set the new cell as the quadrant containing point x.
    For each of the four new cell corners:
      Predict the new node value using bilinear interpolation.
      If the node parent address is non-NULL,
        Access the VQ codebook to add the residual value.
        Update the node address to the appropriate child.
    Re-express point x in the new cell's coordinates.
}
```

Due to the tree adaptivity and the collapse of coarsest levels into a forested mipmap, the number of tree levels traversed in the loop is relatively low on average, as shown in the rightmost column of Table 4.

We have also implemented the evaluation procedure within a GPU pixel program. DirectX 10 enables unfiltered access to 1D memory buffers with a maximum size of 128 MB. This linear memory layout enables better caching behavior than the complex addressing resulting from unfolding the tree in a 2D texture. Integer arithmetic lets us decode the data structure efficiently. The image decompression shader compiles to 298 instructions. On a GeForce 8800 GTX, we render the images at their original resolutions with full filtering enabled. The decompression rates, shown in Table 3, are about 20X slower than the DXT1/BC4U schemes. But of course, these block-based decompression schemes benefit from specialized hardware in the GPU, and the texture caching and filtering system have been optimized for their use. We analyze possible caching strategies in the next section. Even without assistance from specialized hardware, our scheme allows real-time rendering when decompressing a screen-sized texture.

Dataset	Frames/sec	Dataset	Frames/sec
Land (lightmap)	48	Bull (dist)	122
Lady (matte)	60	Ennis (HDR)	34
Teapot (dist)	115	Atlas (lightmap)	47
Desk (HDR)	64	Nefertiti (RGB)	207
Monkey (matte)	82	Flowers (RGB)	189

Table 3: *Current rendering performance on the GPU.*

8. Analysis and discussion

Benefits of tree structure. Data coherence generally permits a very adaptive hierarchy. In particular, note the representation of the signed distance function in Figure 8, where the adaptive tree is able to represent the smooth function at a coarse resolution, yet still capture its localized fine detail (such as sharp corners) at fine resolution. Also, our scheme supports floating point signals at no additional cost, as exploited in the HDR application.

Bandwidth analysis. An important consideration in any compression scheme is the memory bandwidth necessary to decode samples under typical texture access patterns. Indeed, as processors continue to integrate more computational cores, bandwidth becomes the likely bottleneck. Although our hierarchical compression involves several memory accesses (up to 8 at each resolution level in the worst case), most of these accesses are temporally coherent and can therefore be intercepted on-chip. In this section we explore two such bandwidth reduction strategies, which can be used separately or together:

- Cache of multiresolution nodes. We introduce a cache indexed by the parent address and child index (0..3), which returns the child node address and its float value. (Addresses refer to locations within the memory buffer.) We assume a fully associative cache with LRU replacement as in [IM06]. We find that a cache of 256 entries is already very effective. Each entry requires 12 bytes for grayscale signals, so the cache occupies only 3KB.
- Buffering of the last query. We store the multiresolution samples used by the last sample evaluation, i.e. a stack of cells, each holding an (x, y) location, 4 data values, and 4 memory buffer addresses. For a grayscale image, a 6-level stack needs 216 bytes. Given a query point, we iterate through the stack levels fine-to-coarse until the point lies within the buffered cell, and then begin the coarse-to-fine tree evaluation algorithm as before. Consequently we avoid traversing the tree from its root if intermediate resolutions are already buffered, and thereby reduce computation in addition to bandwidth.

We have performed a set of simulations using these two bandwidth reduction strategies. Using the 1025^2 light map of Figure 1, we simulate a Morton (Z-order) texture-space traversal, as would be typical in a rasterization pass, as well as scanline traversal. We also simulate texture mapping the atlas of Figure 10 onto the mesh in Figure 15, with Morton order in screen-space. In both cases, the 256-entry codebook is small (1 KB) and we assume that it is loaded into an on-chip buffer.

Table 4 and Table 5 summarize the simulation results. For the light map, the compressed data size is 135.3 KB, or 10 times smaller than the uncompressed mipmap pyramid. Accessing this compressed data without any caching results in a memory bandwidth of 25807 KB, which is significantly larger than even the original uncompressed data (1052 KB). For the Morton ordering, introducing the 3KB node cache and the last-query buffer reduces bandwidth to 147 KB, which is only 1.1 times the compressed memory representation. Figure 14 graphs bandwidth as a function of total cache size for this Morton traversal. With a sufficiently large node cache, the last-query buffer does not affect bandwidth, but does significantly reduce computation. For the atlas access in Table 5, the bit rate is less than the compressed representation due to mipmapping.

Large datasets. Our current tree construction procedure (Section 5.3) creates a complete tree before adaptively pruning it, and thus does not scale well to large images. However, it should be possible as future work to alter the algorithm to more concisely compute accumulated errors. The runtime representation should scale to larger textures. Of course, a practical alternative is a tiling structure.

Scheme	Bits/pixel	Average number levels traversed
Uncompressed image	8	
Image with its mipmap pyramid	10.7	
Compressed representation	1.03	
Morton order: tree evaluation	190.6	4.8
with multiresolution node cache	1.1	4.8
with buffering of last query	3.7	0.4
with both cache and buffering	1.1	0.4
Scanline order: tree evaluation	190.6	4.8
with multiresolution node cache	9.6	4.8
with buffering of last query	19.2	0.98
with both cache and buffering	9.8	0.98

Table 4: *Analysis of memory bandwidth cost to evaluate the tree-compressed 1025^2 light map of Figure 1, without and with our two bandwidth reduction strategies.*

Scheme	Bits/pixel	Average number levels traversed
Uncompressed image	8	
Image with its mipmap pyramid	10.7	
Compressed representation	2.05	
Atlas access: tree evaluation	270.7	4.8
with multiresolution node cache	1.8	4.8
with buffering of last query	44.7	0.92
with both cache and buffering	1.8	0.92

Table 5: *Memory bandwidth for texturing the mesh of Figure 15 with the atlas of Figure 10.*

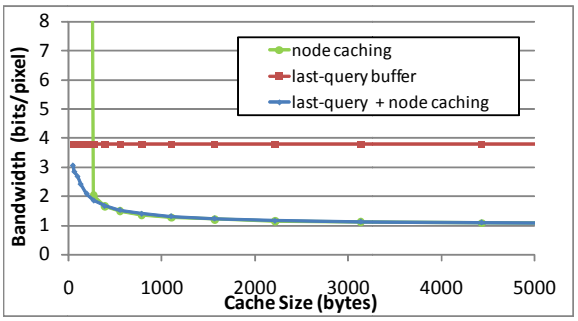


Figure 14: Bandwidth as function of total cache size (including node caching and/or last-query buffer).

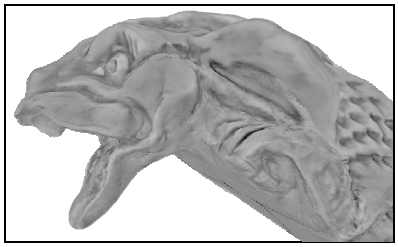


Figure 15: *Viewpoint used for bandwidth measurements on the atlas of Figure 10.*

9. Summary and future work

We have introduced a framework for compressing adaptive hierarchies using a compact *randomly-accessible* tree structure. Such a tree provides a natural continuous mipmap interpolation structure, and we have a shown that this interpolation is achieved most efficiently using a primal subdivision structure.

Some avenues for future work include:

- Dynamic loading and unloading of subtrees for large data, exploiting local offsets to allow data relocation.
- Use of the quadtree construction of Ziegler et al [ZDTS07] for dynamic compression on the GPU.
- Application of the tree structure to octree textures, where sparse VQ will be especially advantageous.
- Use of tree-compressed 3D distance fields for real-time collision detection.
- Generalization of the tree structure to a directed acyclic graph, for representation of tiled texture patterns.
- Architectural designs for hardware implementation.
- Runtime tree updates for incremental data changes.
- Improved tree compression using perceptual metrics.

10. Acknowledgments

We thank Hanan Samet for pointing us in the direction of autumnal trees, Nick Apostoloff and Jue Wang for the alpha-matte data. The textured model used in Figure 10 and Figure 15 is from the MIT CSAIL database. The HDR image of Figure 1 is courtesy of Roimela et al [RAI06]. The HDR image of Figure 12 is from OpenEXR.

11. References

[BAC96] BEERS A., AGRAWALA M., CHADDHA N. 1996. Rendering from compressed textures. *ACM SIGGRAPH*.

[BD02] BENSON D., DAVIS J. 2002. Octree textures. *ACM SIGGRAPH*, 785-790.

[Bly06] BLYTHE D. 2006. The Direct3D 10 system. *ACM SIGGRAPH*, 724-734.

[BWK02] BOTSCH M., WIRATANAYA A., KOBBELT L. 2002. Efficient high quality rendering of point sampled geometry. *Eurographics Workshop on Rendering*, 53-64.

[BA83] BURT P., ADELSON E. 1983. The Laplacian pyramid as a compact image code. *IEEE Trans. on Comm.* 31(4), 532-540.

[CCG96] CHADDHA N., CHOU P., GRAY R. 1996. Constrained and recursive hierarchical table-lookup vector quantization. *IEEE Data Compression Conference*.

[CB04] CHRISTENSEN P., BATALI D. 2004. An irradiance atlas for global illumination in complex production scenes. *Eurographics Symposium on Rendering*.

[CAD04] COHEN-STEINER D., ALLIEZ P., DESBRUN M. 2004. Variational shape approximation. *ACM SIGGRAPH*, 905-914.

[DGPR02] DEBRY D., GIBBS J., PETTY D., ROBINS N. 2002. Painting and rendering on unparameterized models. *ACM SIGGRAPH*, 763-768.

[FM86] FABBRINI F., MONTANI C. 1986. Autumnal quadtrees. *The Computer Journal*, 29(5), 472-474.

[FFBG01] FERNANDO R., FERNANDEZ S., BALA K., GREENBERG D. 2001. Adaptive shadow maps. *ACM SIGGRAPH*, 387-390.

[FPRJ00] FRISKEN S., PERRY R., ROCKWOOD A., JONES T. 2000. Adaptively sampled distance fields: A general representation of shape for computer graphics. *ACM SIGGRAPH*, 249-254.

[Gar82] GARGANTINI I. 1982. An effective way to represent quadtrees. *Communications of the ACM*, 25(12), 905-910.

[GG92] GERSHO A., GRAY R. 1992. *Vector quantization and signal compression*. Kluwer Academic Publishers, Boston.

[GS84] GERSHO A., SHOHAM Y. 1984. Hierarchical vector quantization of speech with dynamic codebook allocation. *ICASSP*, 9(1), 416-419.

[GW91] GOLDBERG M., WANG L. 1991. Comparative performance of pyramid data structures for progressive image transmission. *IEEE Trans. on Comm.* 39(4).

[Hec90] HECKBERT P. 1990. Adaptive radiosity textures for bidirectional ray tracing. *ACM SIGGRAPH*, 145-154.

[HG88] HO Y-S., GERSHO A. 1988. Variable-rate multi-stage vector quantization for image coding. *IEEE ICASSP*, 1156-1159.

[HW91] HUNTER A., WILLIS P. 1991. Classification of quad-encoding techniques. *Eurographics Conference*.

[IM06] INADA T., MCCOOL M. 2006. Compressed lossless texture representation and caching. *Graphics Hardware*, 111-120.

[Kno80] KNOWLTON K. 1980. Progressive transmission of grey-scale and binary pictures by simple, efficient, and lossless encoding schemes. *Proceedings of IEEE*.

[KE02] KRAUS M., ERTL T. 2002. Adaptive texture maps. *Graphics Hardware*, 7-15.

[LH06] LEFEBVRE S., HOPPE H. 2006. Perfect spatial hashing. *ACM SIGGRAPH*, 579-588.

[LKS*06] LEFOHN A., KNISS J., STRZODKA R., SENGUPTA S., OWENS J. 2006. Glift: Generic, efficient, random-access GPU data structures. *ACM TOG*, 25(1).

[LFWV03] LENDASSE A., FRANCOIS D., WERTZ V., VERLEYSEN M. 2003. Nonlinear time series prediction by weighted vector quantization. *ICCS*, 417-426.

[Llo82] LLOYD S. 1982. Least squares quantization in PCM. *IEEE Transactions on Information Theory* 28(2).

[MB98] MCCABE D., BROTHERS J. 1998. DirectX 6 texture map compression. *Game Developer*, 42-46.

[MCHA06] MUNKBERG J., CLARBERG P., HASSELGREN J., AKENINE-MÖLLER T. 2006. High dynamic range texture compression for graphics hardware. *ACM SIGGRAPH*.

[NH92] NING P., HESSELINK L. 1992. Vector quantization for volume rendering. *Workshop on Volume Visualization*, 69-74.

[RAI06] ROIMELA K., AARNIO T., ITÄRANTA J. 2006. High dynamic range texture compression. *ACM SIGGRAPH*.

[RL01] RUSINKIEWICZ S., LEVOY M. 2001. QSplat: A multiresolution point rendering system for large meshes. *ACM SIGGRAPH,* 343-352.

[Sam85] SAMET H. 1985. Data structures for quadtree approximation and compression. *CACM* 28(9), 973-993.

[Sam06] SAMET H. 2006. *Foundations of multidimensional and metric data structures*. Morgan Kaufman.

[SSGH01] SANDER P., SNYDER J., GORTLER S., HOPPE H. 2001. Texture mapping progressive meshes. *ACM SIGGRAPH*, 409-416.

[SK01] SAUPE D., KUSKA J.-P. 2001. Compression of isosurfaces for structured volumes. *VMV*, 471-476.

[SW03] SCHNEIDER J., WESTERMANN R. 2003. Compression domain volume rendering. *IEEE Visualization*, 39.

[Sha93] SHAPIRO J. 1993. Embedded image coding using zerotrees of wavelet coefficients. *IEEE Trans. on Signal Processing*, 41(12), 3445-3462.

[SA05] STRÖM J., AKENINE-MÖLLER T. 2005. iPACKMAN: High-quality, low-complexity texture compression for mobile phones. *ACM Graphics Hardware*, 63-70.

[TS00] TZOVARAS D., STRINTZIS M. 2000. Optimal construction of reduced pyramids for lossless and progressive image coding. *IEEE TCS*, 47(4), 332-348.

[VG88] VAISEY J., GERSHO A. 1988. Variable rate image coding using quad-trees and vector quantization. *EURASIP*.

[Woo84] WOODWARK J. 1984. Compressed quad trees. *The Computer Journal*, 27(3), 225-229.

[XPH05] XU R., PATTANAIK S., HUGHES C. 2005. High-dynamic-range still-image encoding in JPEG 2000. *IEEE CG&A* 25(6), 57-64.

[YFT80] YAMADA Y., FUJITA K., TAZAKI S. 1980. Vector quantization of video signals. *Proceedings of IECE*.

[ZDTS07] ZIEGLER G., DIMITROV R., THEOBALT C., SEIDEL H.P. 2007. Real-time Quadtree Analysis using HistoPyramids. *IS&T and SPIE Conference on Electronic Imaging*.

[ZS01] ZORIN D., SCHRÖDER P. 2001. A unified framework for primal/dual quadrilateral subdivision schemes. *CAGD*, 18(5), 429-454.