Random Access Segmentation Volume Compression for Interactive Volume Rendering – Supplemental

M. Piochowiak, F. Kurpicz and C. Dachsbacher

In this supplemental document we explain how to efficiently answer rank queries on wavelet trees and give additional render timing results of our methods to assess the effect of wavelet trees on performance. This includes a recapitulation of bit vector queries from the main document, followed by detailed explanations of our adapted flat rank [Kur22] structure for constant time rank queries on bit vectors and rank queries on wavelet trees. The last section details the effect of using wavelet trees in our random access CSGV-R segmentation volume compression on render times. The random access and rank support of wavelet trees comes at the cost of tree traversal for all queries. However, we show how the benefits outweigh the costs and render times with our optimized implementations are overall faster compared to an exemplary 4-bit Nibble random access encoding as well as compared to the existing serial CSGV [PD24] compression.

1. Rank Queries on Bit Vectors

A bit vector B is a text over the binary alphabet $\{0,1\}$. The query $rank_\alpha(i) = |\{j < i \colon B[j] = \alpha\}|$ for $\alpha \in \{0,1\}$ returns the number of times α occurred before i. As $rank_0(i)$ can easily be computed as $i - rank_1(i)$, we only discuss $rank_1$ queries. Our bit vectors are stored as arrays of 64 bit words where the first indexed bit in a word is the least significant bit (LSB). On a single word, $rank_1(i)$ can be computed using the `popcount` operator (`bitCount` in GLSL shaders) that retunrs the number of `1` bits set in a word:

```
int rank1Word(uint64 word, int i) {
    return (i > 0u) ? popcount(word << (64u - i))
                    : 0u;
}
```

For $rank_1$ queries on a full bit vector, a naive implementation would have to loop over all words $j < \lfloor i/64 \rfloor$ adding up full word `popcount` results followed by a final `rank1word` on the word at index $\lfloor i/64 \rfloor$. To achieve constant time complexity, the general idea is to split the bit vector into equally sized blocks and store a helper array which stores the number of `1` bits up to the beginning of each block [Jac89]. Storing the bit counts hierarchically reduces memory consumption: The currently most space efficient approach (requiring 3.51% additional space) with reasonable query performance is flat rank [Kur22]. We use an adapted flat rank variant in our method with slightly faster query performance which is designed as follows:

The bit vector is split into blocks of 1280 bits, or 20 words, size. Each entry in our flat rank array covers one of these blocks

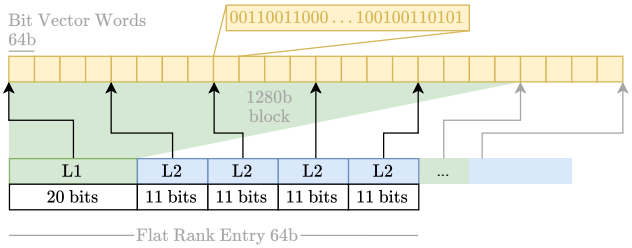

Figure S1 Our adapted flat rank [Kur22] layout for bit vector $rank_1$ queries in constant time at 5% space overhead: Each 64 bit flat rank entry covers a block of 1280 bits in the bit vector (yellow). L1 counts `1` bits in the bit vector before the block (green). L2 entries count `1` bits from the beginning of the block in 4 word intervals (blue).

Algorithm S1 Constant Time Bit Vector $rank_1(i)$

Input bit index i, bit vector B, flat rank array FR
Output number of `1` bits in B before i: $rank_1(i)$

1: $f \leftarrow FR[i/1280]$
2: $count \leftarrow f.\text{getL1}() + f.\text{getL2}(\lfloor (i \bmod 1280)/64^4 \rfloor)$
3: $w \leftarrow \lfloor \lfloor i/64 \rfloor /4 \rfloor \times 4$
4: **for** $(\lfloor i/64 \rfloor \bmod 4)$ times **do** ▷ up to 3 times
5: $count \leftarrow count + \text{popcount}(B.\text{words}[w])$
6: $w \leftarrow w + 1$
7: **return** $count + \text{rank1Word}(B.words[w], i \bmod 64)$

(Figure S1). Our flat rank has a hierarchical depth of two as opposed to the original depth of three: A flat rank array element consists of a single 20 bit L1 entry followed by four 11 bit L2 entries. The L1 entries cover a full 1280 bit block and store the number of `1` bits in the bit vector up to their respective block. L2 entries cover intervals of 4 words each, as opposed to 8 words form the original flat rank, and store the number of `1` bits from the beginning of the block up to their respective interval within the block. Putting it all together, a $rank_1$ query on a bit vector is a sum of the respective L1 and L2 entries, up to three full word (`popcount`) and one partial (`rank1word`) bit counts after the L2 entry start (Algorithm S1).

Compared to the original flat rank [Kur22], removing one hierarchy level and covering half as many words with each L2 entry results in faster querying. We can make this optimization since we have an upper bound for our bit vector length: In our Random Access

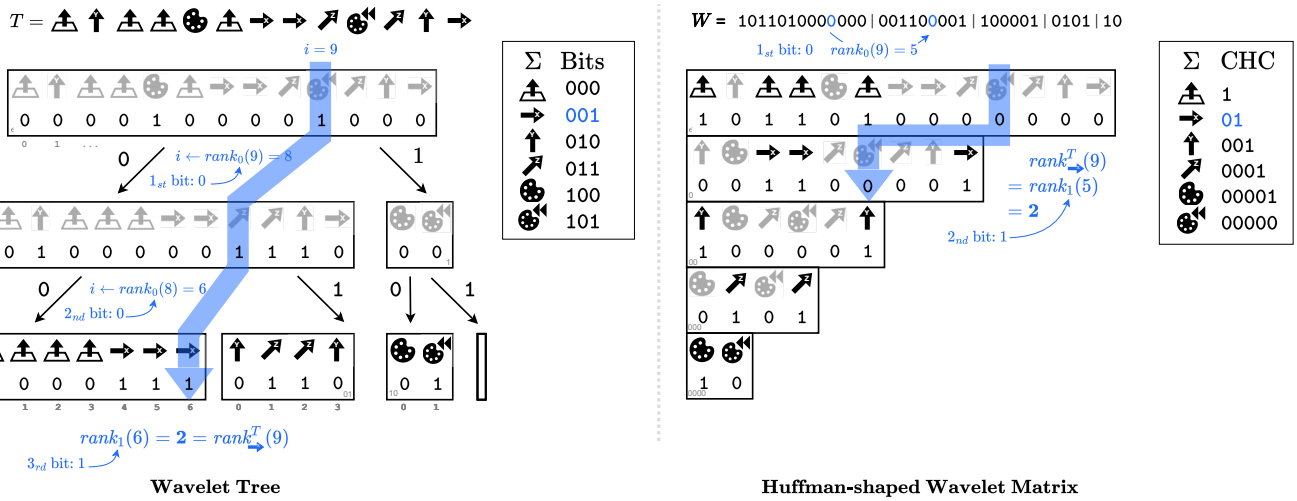

Figure S2 Wavelet tree *rank*. A wavelet tree that encodes a text T of operations in its conceptual form (left), and as stored in memory in form of a compressed Huffman-shaped wavelet matrix (right). **Wavelet tree** traversal for $rank_{➡}(9) = 2$, i.e. the number of times ➡ occurs in T before index 2, is highlighted in blue: Starting at position $i = 9$ in the root node at the top, the traversal bit α in each level $\mathbf{l}$ is the $\mathbf{l}$-th bit of the queried character, in this case `001` for ➡ . We descend to the left node in the next level if α is `0`, and to the right node otherwise. The index i is updated for the next node with a $rank_\alpha(i)$ query on the current node's bit vector. Once the node of the character's last bit is reached, one final $rank_\alpha(i)$ returns the overall requested result. The **Huffman-shaped wavelet matrix** is a pointer-less, compressed representation of a wavelet tree. It stores characters as ther canonical Huffman-codes (CHC) with a variable number of bits. In our implementation, we make use of streamlined Huffman-shaped wavelet matrix that does not have any right child nodes. *rank* queries are computed exactly as before with the only differences that queries terminate on different levels and descend only occurs for `0` bits. Note that characters are only added for readability. In memory, the Huffman-shaped wavelet matrix is only stored as the bit vector W with the start indices of tree levels within.

Algorithm S2 Operation Stream Wavelet Matrix $rank_\omega$, $\omega \in \Sigma$

Input index i, character ω, wavelet matrix bit vector W from brick header: $W_\mathbf{l}$ level starts, $Z_1[\mathbf{l}] = rank_1^W(W_\mathbf{l})$
Output number of ω in T before i: $rank_\omega(i)$

for $\mathbf{l} = 0 \dots (\omega.\text{bitLength} - 1)$ **do**
  $ones_{before} \leftarrow rank_1^W(W_\mathbf{l} + i) - Z_1[\mathbf{l}]$
  $i \leftarrow i - ones_{before}$
if $\omega.\text{lastBit} = 0$ **then**
  return $ones_{before}$ ▷ bit vector $rank_1$ in level
else
  return $i - ones_{before}$ ▷ $rank_0$ in level, only for 🍋

Algorithm S3 Operation Stream Wavelet Matrix $rank_{🍇}$

Input operation stream index i, wavelet matrix bit vector W from brick header: $W_\mathbf{l}$ level starts, $Z_1[\mathbf{l}] = rank_1^W(W_\mathbf{l})$
Output number of 🍇 in T before i: $rank_{🍇}(i)$

for $\mathbf{l} = 0 \dots 4$ **do**
  $ones_{before} \leftarrow rank_1^W(W_\mathbf{l} + i) - Z_1[\mathbf{l}]$
  $i \leftarrow i - ones_{before}$
return $ones_{before}$

Compressed Segmentation Volumes (CSGV-R) compression, the longest bit vectors store the wavelet matrices which compress one segmentation volume brick each. These bricks have at most 64^3 operation symbol entries and the wavelet matrix only stores few bits per symbol. Our flat rank structure can cover bit vectors up to a length of $2^{20} + 5 \times 2^{11}$ bits, which is more than enough even for worst case bricks.

2. Rank Queries on Wavelet Trees

Wavelet trees [GGV03, GVX11, Mak12] are compressible random-access data structures that generalize *access* and *rank* queries from binary alphabets to texts T over arbitrary sized alphabets Σ. While wavelet trees fundamentals and $access(i) = T[i]$ queries are described in detail in the main document, we give a more comprehensive explanation of $rank_\omega^T(i)$ queries for $\omega \in \Sigma$ in the following.

Figure S2 gives an exemplary *rank* query on a general wavelet tree and on our streamlined compressed Huffman-shaped wavelet matrix. Because of how wavelet trees partition the alphabet of a text, all identical characters in T end up in the same node. Thus, as we are interested in the number of times a certain character $\omega \in \Sigma$ occurs in T before a position i, we only need to find ω's final tree node in which a single binary *rank* returns the overall result. Wavelet tree (left in Figure S2) *rank* queries behave similar to *access* queries: We start at index i in the root node and repeatedly descend to the left or right child node until all bits of the respective character ω were visited. The main difference is that ω is not the result, but an argument of the *rank* query. The respective traversal bit α is therefore

		local shading				shadow rays				ambient occlusion			
		CELLS	FIBER	H01	AZBA	CELLS	FIBER	H01	AZBA	CELLS	FIBER	H01	AZBA
CSGV Nibble	brick cache	10.2	5.5	-	9.8	11.6	7.3	-	11.8	15.8	8.6	-	16.1
CSGV rANS	brick cache	11.8	6.0	4.1	10.3	13.1	7.8	4.2	12.4	18.1	9.2	4.8	17.2
CSGV-R Nibble *no stop bits no wavelet trees*	no cache	381.9	11.4	-	55.7	492.0	18.3	-	69.7	472.2	20.2	-	67.0
	voxel cache	13.1	6.4	-	9.0	15.8	9.1	-	11.1	18.1	11.7	-	13.1
	voxel cache (es)	10.6	5.9	-	8.7	12.3	8.4	-	10.9	14.4	10.8	-	12.6
	brick^c cache	51.3	8.6	-	15.6	52.8	10.4	-	18.1	85.9	12.6	-	27.6
	brick^c cache (sm)	42.3	7.9	-	13.9	43.4	9.9	-	16.3	64.5	12.6	-	24.2
CSGV-R *no stop bits*	no cache	25.3	7.6	-	19.2	32.9	12.7	-	25.0	37.4	15.4	-	31.1
	voxel cache	10.1	6.4	-	9.3	12.8	9.2	-	11.6	13.2	11.9	-	13.3
	voxel cache (es)	**5.4**	5.8	-	**7.9**	**6.7**	8.1	-	**10.0**	**8.1**	10.7	-	**12.2**
	brick^c cache	8.6	4.7	-	9.6	9.9	6.5	-	11.5	11.8	7.5	-	14.3
	brick^c cache (sm)	8.1	**4.5**	-	9.3	9.4	**6.4**	-	11.2	11.3	**7.4**	-	13.9
CSGV-R+sb	no cache	44.9	12.8	6.5	28.6	59.5	22.5	8.6	37.9	70.4	29.6	14.1	47.2
	voxel cache	9.7	6.3	**1.2**	8.8	12.4	9.0	**1.3**	10.9	13.2	11.9	**1.8**	13.0
	voxel cache (es)	6.1	7.1	75.8	8.1	7.6	9.9	79.9	10.2	9.4	13.3	109.1	12.6
	brick^c cache	11.6	5.8	21.4	11.2	12.8	7.5	21.6	13.2	15.2	8.6	22.1	16.7
	brick^c cache (sm)	11.0	5.6	22.4	10.9	12.3	7.3	22.6	12.9	14.6	8.4	23.6	16.3

Table S1 Extended version of Table 4 from the main document. The table contains average milliseconds per frame when rendering a camera path (2649, 1401, 2649, 2070 frames) for the CELLS, FIBER, H01, and AZBA data sets with different cache and shading modes. H01 uses $b = 64$, others $b = 32$. The cache size is 4 GiB for H01 with Fast Compressed Segmentation Volumes (CSGV) and 1 GiB for others. For CSGV-R without stop bits, the compressed H01 does not fit into VRAM. CSGV uses rANS compression and only supports brick caching. With shadow rays and ambient occlusion, one secondary ray is cast per frame per pixel after the primary hit. Our random access CSGV-R can decode bricks cooperatively within a subgroup (brick^c) where each thread decodes one output voxel. As one brick is decoded per subgroup (as opposed to one brick per thread in CSGV) we can additionally copy the encoding to shared memory before (brick^c (sm)) To assess the effect of wavelet trees in our encoding apart from compression, CSGV-R Nibble implements our random access scheme without wavelet trees. This has low performance as it lacks constant-time palette index $rank_{\bullet}(i)$ queries and must therefore compute palette indices using a for loop over all operations $j < i$ per output voxel.

not obtained through an *access* on the current node's bit vector but directly from ω's bit code instead. The lookup position in the next level is updated with $rank_\alpha(i)$ in the current node as usual. Once the tree node of the final character bit α is reached, a last $rank_\alpha(i)$ returns the overall result of the wavelet tree $rank^T_\omega$ query.

In our implementation, we store wavelet trees in form of a pointer-less Huffman-shaped wavelet matrix [CNP15] (right in Figure S2). The Huffman-shaped wavelet matrix compresses the text T by assigning variable numbers of bits to symbols. The respective bit codes are given by the CHC of the symbols. Our CHC follow a simple pattern, similar to RICE coding [RP71], where any 1 bit terminates the character. Thus, the resulting wavelet matrix contains no right child nodes which allows for a very efficient implementation of queries without branching. Furthermore, as only one node exists per level (the 0^l prefix), we can optimize out the iterative tracking of node bit vector intervals within wavelet matrix levels that an implementation for arbitrary symbol frequencies requires [CNP15]. Our resulting streamlined wavelet tree *rank* for a Huffman-shaped wavelet matrix with the layout from Figure S2 is extremely simple (Algorithm S2). For our CSGV segmentation volume encoding, we only ever need to compute a wavelet matrix *rank* for the symbol in practice. Hard-coding the symbol as in Algorithm S3 streamlines the implementation even further.

3. Effect of Wavelet Trees on Render Timings

Table S1 lists additional render timings for our methods. We compare our random access CSGV-R compression utilizing wavelet trees against the serially operating previous work CSGV [PD24] which uses ANS [Dud13] for variable bit-length coding. For assessing the benefits of wavelet trees apart from compression, we add timings for CSGV Nibble which serially decodes one brick per thread from a stream of plain 4-bit operation codes. Comparing the serial modes from previous work CSGV, the rANS decoder has only minimal overhead. However, our random access CSGV-R wavelet tree modes are even *faster* than CSGV Nibble in almost all cases.

The reasons for this are two-fold: First, while it may initially seem that wavelet tree queries are expensive due to the necessary tree traversal, the actual overhead is not as problematic. The layout of our Huffman-shaped wavelet matrix (Figure S2) allows for extensive implementation optimizations as presented in the algorithms in the main document and this supplemental. Our alphabet is extremely small compared to usual wavelet tree tasks, limiting the tree depths. Further, most wavelet tree *access* queries terminate at early depths of the tree - for the most frequent operation immediately in the first level - and a wavelet matrix *rank* is only computed once per output voxel. Second, our random access decoding maps better to GPU hardware leading to strong performance increases: When not using caching or with voxel caching, our CSGV-R removes the need for the to shading atlas streaming-like shader stages [MVD*18] that

CSGV requires for the brick cache management. Voxels are decoded directly in the rendering shader. Even with brick caching, CSGV-R achieves faster timings as it can cooperatively decode one single brick per subgroup (brick^c cache) which improves utilization of GPU caches. This also allows copying brick encodings to shared memory before decompression (brick^c cache (sm)) which is even faster by a small margin. The serially operating CSGV on the other hand must decode one brick per *thread*, i.e. many different bricks within the same subgroup (brick cache). As brick encoding lengths differ, this can lead to stalling threads within subgroups.

Note that wavelet matrices are not only used for operation stream compression but are an essential component of our CSGV-R random access scheme as they also efficiently answer *rank* queries. For each decoded output voxel, such a query must be computed once to obtain the palette index of its label. To highlight the importance of fast *rank* support, we implement a random access decoder that operates on a plain 4-bit stream (CSGV-R Nibble). It does not use stop bits and, as required by our random access scheme, does not use the $\clubsuit^i$ operation. While *access* queries are faster, performance is significantly worse than with wavelet trees: For lack of an efficient $rank_{\clubsuit}(i)$ query, palette indices must be obtained with an expensive for loop over all brick operations $0 \dots i$ per output voxel.

References

[CNP15] CLAUDE F., NAVARRO G., PEREIRA A. O.: The wavelet matrix: An efficient wavelet tree for large alphabets. *Inf. Syst. 47* (2015), 15–32. doi:10.1016/j.is.2014.06.002. 3

[Dud13] DUDA J.: Asymmetric numeral systems: entropy coding combining speed of huffman coding with compression rate of arithmetic coding. *arXiv preprint arXiv:1311.2540* (2013). doi:10.48550/arXiv.1311.2540. 3

[GGV03] GROSSI R., GUPTA A., VITTER J. S.: High-order entropy-compressed text indexes. In *SODA* (2003), ACM/SIAM, pp. 841–850. 2

[GVX11] GROSSI R., VITTER J. S., XU B.: Wavelet trees: From theory to practice. In *CCP* (2011), IEEE Computer Society, pp. 210–221. doi:10.1109/CCP.2011.16. 2

[Jac89] JACOBSON G.: Space-efficient static trees and graphs. In *FOCS* (1989), pp. 549–554. doi:10.1109/SFCS.1989.63533. 1

[Kur22] KURPICZ F.: Engineering compact data structures for rank and select queries on bit vectors. In *SPIRE* (2022), vol. 13617 of *Lecture Notes in Computer Science*, Springer, pp. 257–272. doi:10.1007/978-3-031-20643-6_19. 1

[Mak12] MAKRIS C.: Wavelet trees: A survey. *Comput. Sci. Inf. Syst. 9*, 2 (2012), 585–625. doi:10.2298/CSIS110606004M. 2

[MVD*18] MUELLER J. H., VOGLREITER P., DOKTER M., NEFF T., MAKAR M., STEINBERGER M., SCHMALSTIEG D.: Shading atlas streaming. *ACM Transactions on Graphics (Proc. SIGGRAPH) 37*, 6 (Dec. 2018). doi:10.1145/3272127.3275087. 3

[PD24] PIOCHOWIAK M., DACHSBACHER C.: Fast compressed segmentation volumes for scientific visualization. *IEEE Transactions on Visualization and Computer Graphics 30*, 1 (2024), 12–22. doi:10.1109/TVCG.2023.3326573. 1, 3

[RP71] RICE R., PLAUNT J.: Adaptive variable-length coding for efficient compression of spacecraft television data. *IEEE Transactions on Communication Technology 19*, 6 (1971), 889–897. doi:10.1109/TCOM.1971.1090789. 3