CHAPTER 5

Computing Minima and Maxima of Subarrays

Ingo Wald
NVIDIA

ABSTRACT

This chapter explores the following problem: given an array A of N numbers A_i, how can we efficiently query the minimal or maximal numbers in any sub-range of the array? For example, "what is the minimum of the 8th to the 23rd elements?"

5.1 MOTIVATION

Unlike the topics of other chapters, this particular problem does not *directly* relate to ray tracing in that it does not cover how to generate, trace, intersect, or shade a ray. However, it is a problem occasionally encountered when ray tracing, in particular when rendering volumetric data sets. Volumetric rendering of data sets, whether structured or unstructured volumes, usually defines a scalar field, $z = f(x)$, that typically is rendered with some form of ray marching. As with surface-based data sets, the key to fast rendering is quickly determining which regions of the volume are empty or less important, and speeding up computation by skipping these regions, taking fewer samples, or using other approximations. This typically involves building a spatial data structure that stores, per leaf, the minimal and maximal values of the underlying scalar field.

In practice, this chapter's problem arises because a scalar field is rarely rendered directly—instead, the user interactively modifies some sort of *transfer function* $\mathrm{t}(z)$ that specifies which color and opacity values map to different scalar field values (e.g., to make muscle and skin transparent, and ligaments and bone opaque). In that case, the extremal values of a region's scalar field are not important for rendering. Instead, we need the extremal values of the *output of our transfer function* applied to our scalar field. In other words, assuming we represent our transfer function as an array $A[i]$, and the minimum and maximum of the scalar field map to array indices i_{lo} and i_{hi}, respectively, what we want is the minimum and maximum of $A[i]$ for $i \in [i_{lo}, i_{hi}]$.

At first glance, our problem looks similar to computing the sum for a subarray, which can be done using summed-area tables (SATs) [3, 9]. However, min() and max() are not invertible, so SATs will not work. The remainder of this chapter discusses four different solutions to this problem, each having different trade-offs regarding the memory required for precomputation and query time.

E. Haines, T. Akenine-Möller (eds.), *Ray Tracing Gems*, https://doi.org/10.1007/978-1-4842-4427-2_5

5.2 NAIVE FULL TABLE LOOKUP

The naive solution precomputes an $N \times N$ sized table, $M_{j,k} = \min \{A_i, i \in [j, k]\}$, and simply looks up the desired value.

This solution is trivial and fast, providing a good "quick" solution (see, e.g., `getMinMaxOpacityInRange()` used in OSPRay [7]). It does, however, have one big disadvantage: storage cost is quadratic ($O(N^2)$) in array size N, so for nontrivial arrays (e.g., 1k or 4k entries), this table can grow large. In addition to size, this table has to be recomputed every time the transfer function changes, at a cost of at least $O(N^2)$.

Given this complexity, the *full table* method is good for small table sizes, but larger arrays probably require a different solution.

5.3 THE SPARSE TABLE METHOD

A less known, but worthwhile, improvement upon the full table method is the *sparse table* approach outlined in the online forum *GeeksForGeeks* [6]. We were unaware of this method until performing our literature search (and we did not find it discussed elsewhere); as such, we briefly describe it here.

The core idea of the sparse table method is that any n-element range $[i..j]$ can be seen as the union of two (potentially overlapping) power-of-two sized ranges (the first beginning at i, the other ending at j). In that case, we do not actually have to precompute the full table of *all* possible query ranges, but only those for power-of-two sized queries; then we can look up the precomputed results for the two power-of-two ranges and finally combine their results.

In a bit more detail, assume that we first precompute a lookup table $L^{(1)}$ of all possible queries that are $2^1 = 2$ elements wide; i.e., we compute $L_0^{(1)} = \min(A_0, A_1)$, $L_1^{(1)} = \min(A_1, A_2)$, and so on. Similarly, we then compute table $L^{(2)}$ for all $2^2 = 4$ wide queries, $L^{(3)}$ for all $2^3 = 8$ wide queries, etc.[1]

Once we have these $\log N$ tables $L^{(i)}$, for any query range $[lo, hi]$ we can simply take the following steps: First, compute the width of the query as $n = (hi - lo + 1)$. Then, compute the largest integer p for which 2^p is still smaller than n. Then, the range $[lo, hi]$ can be seen as the union of the two ranges $[lo, lo + 2^p - 1]$ and $[hi - 2^p + 1, hi]$. Since the queries for those have been precomputed in table $L^{(p)}$, we can simply look up the values $L_{lo}^{(p)}$ and $L_{hi-2^p+1}^{(p)}$, compute their minimum, and return the result. A detailed illustration of this method is given in Figure 5-1.

[1]At least logically, we can also assume a table $L^{(0)}$ of 1 wide queries, but this is obviously identical to the input array A and thus would not get stored.

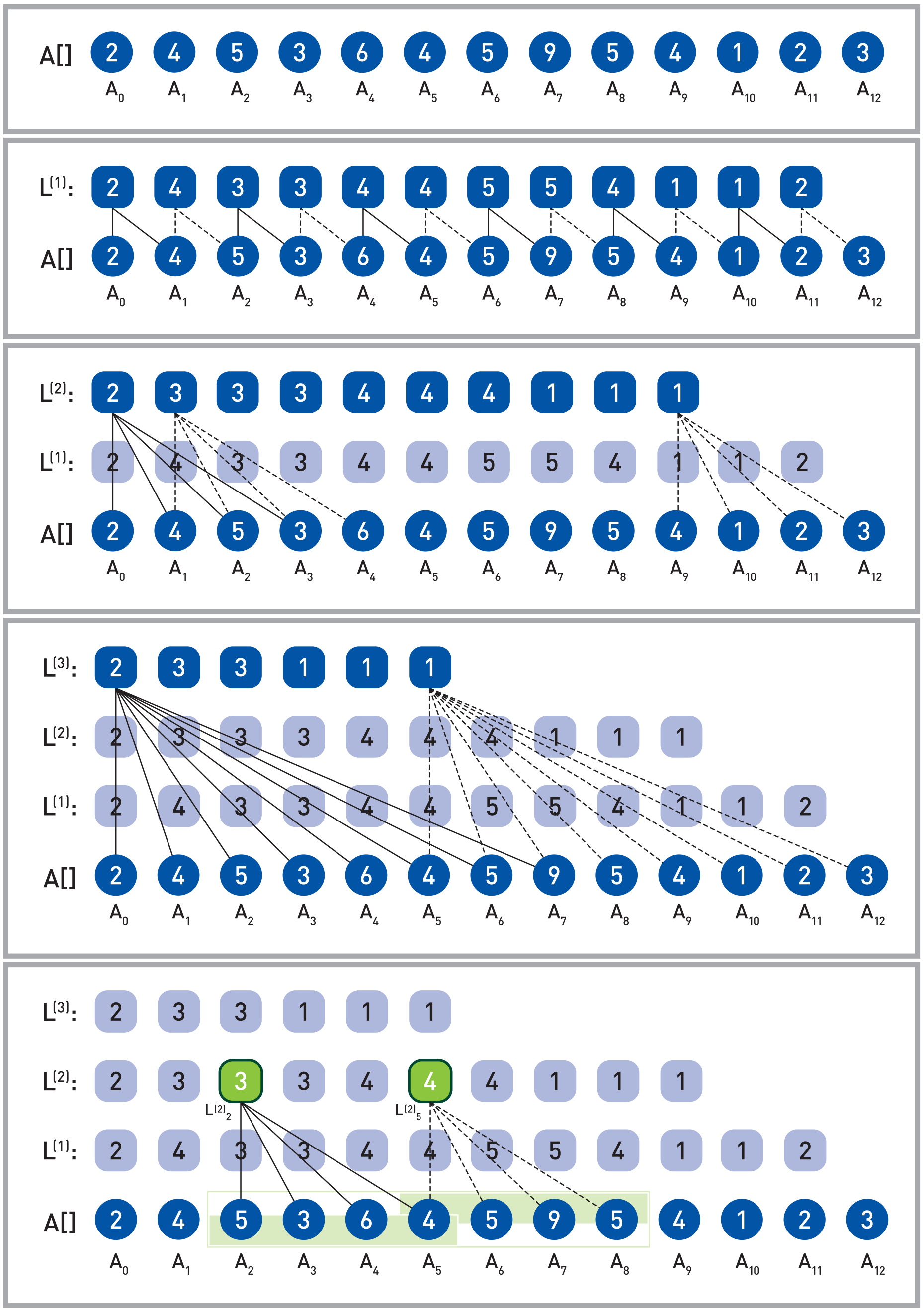

Figure 5-1. *Example of the sparse table method: from our 13-element input array A[], we precompute tables $L^{(1)}$, $L^{(2)}$, and $L^{(3)}$ containing all 2, 4, and 8 wide queries. Assuming that we query for the minimum of the 7-element range $[A_2 . . A_8]$, we can decompose this query into the union of two overlapping 4-wide queries ($[A_2 . . A_5]$ and $[A_5 . . A_8]$). These decomposed queries were precomputed in table $L^{(2)}$. Thus, the result is* $min\left(L_2^{(2)}, L_5^{(2)}\right) = min\left(3, 4\right) = 3$.

For a non–power-of-two input range the two sub-ranges will overlap, meaning that some array elements will be accounted for twice. This makes the method unsuitable for other sorts of reductions such as summation and multiplication; for minimum and maximum, however, this double-counting does not change the results. In terms of compute cost, the method is still $O(1)$ because all queries can be completed with exactly two lookups. In terms of memory cost, there are $N - 1$ entries in $L^{(1)}$, $N - 3$ in $L^{(2)}$, etc., for a total storage cost of $O(N \log N)$—which is a great savings over the full table method's $O(N^2)$.

5.4 THE (RECURSIVE) RANGE TREE METHOD

For ray tracing—where binary trees are, after all, a common occurrence—an obvious solution to our problem is using some type of *range tree*, as introduced by Bentley and Friedman [1, 2, 8]. An excellent discussion of applying range trees to our problem can be found online [4, 5].[2]

A range tree is a binary tree that recursively splits the range of inputs and, for each node, stores the corresponding subtree's result. Each leaf corresponds to exactly one array element; inner nodes have two children (one each for the lower and upper halves of its input range) and store the minimum, maximum, sum, product, etc. of the two children. An example of such a tree—for both minimum and maximum queries—is given in Figure 5-2.

[2]Note that those articles use the term *segment tree* but describe the same data structure and algorithm. This chapter adopts the *range tree* term used by both Bentley and Wikipedia.

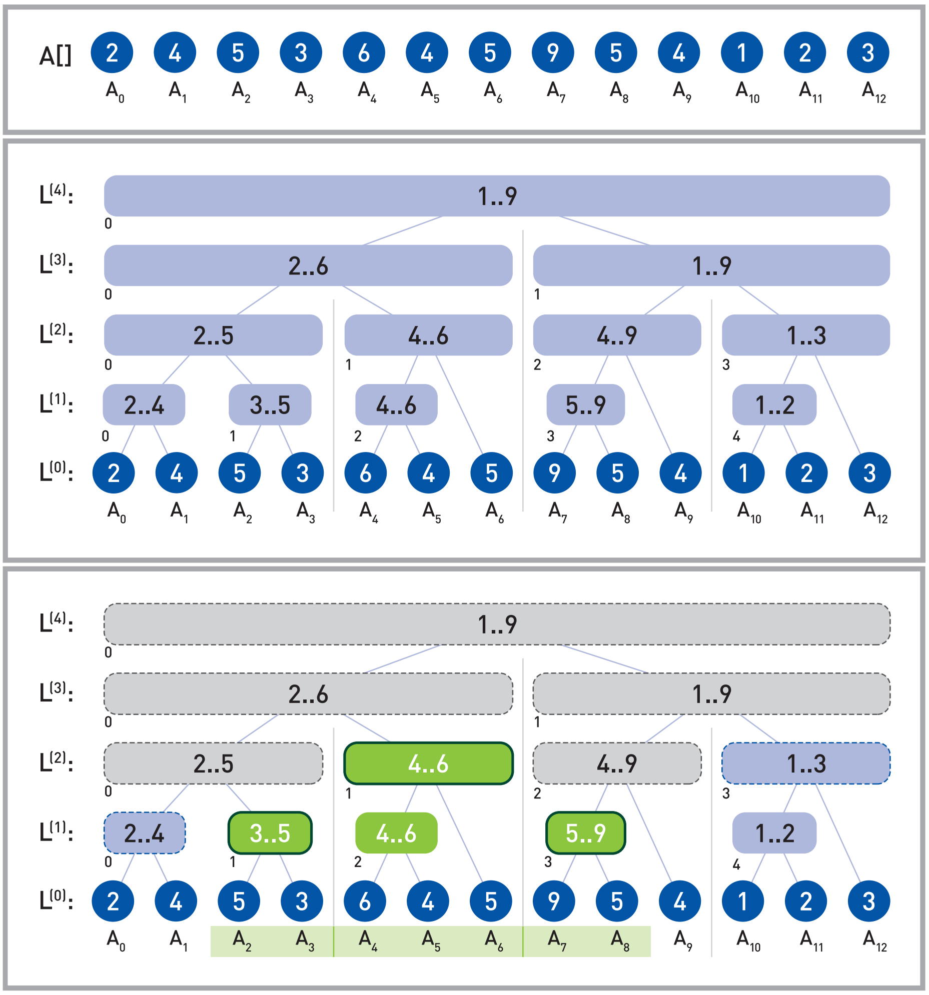
A[]
2 4 5 3 6 4 5 9 5 4 1 2 3
A0 A1 A2 A3 A4 A5 A6 A7 A8 A9 A10 A11 A12
L[4]: 1..9
L[3]: 2..6 1..9
L[2]: 2..5 4..6 4..9 1..3
L[1]: 2..4 3..5 4..6 5..9 1..2
L[0]:

Figure 5-2. *Illustration of the recursive range tree method. Given input array A (top), we compute a binary tree (middle) where each node stores the minimum and maximum of its corresponding leaf nodes. Our recursive traversal for a query range (bottom) uses all three cases from the pseudocode: gray nodes recurse into both children (case 3), green nodes with dark outlines get counted and terminate (case 2), and blue nodes with dashed outlines lie outside the range (case 1).*

Given such a range tree, querying over any range [*lo*, *hi*] requires finding the set of nodes that exactly spans the input range. The following simple recursive algorithm performs this query:

```
RangeTree::query(node,[lo,hi]) {
    if (node.indexRange does not overlap [lo,hi])
        /* Case 1: node completely outside query range -> ignore. */
        return { empty range }
    if (node.indexRange is inside [lo,hi])
        /* Case 2: node completely inside query range -> use it. */
        return node, valueRange
    /* Case 3: partial overlap -> recurse into children, & merge. */
    return merge(query(node.leftChild,[lo,hi]),
                 query(node.rightChild,[lo,hi])
}
```

Range trees require only linear storage and preprocessing time, which can be integer factors less than the sparse table method. On the downside, queries no longer occur in constant time, but instead have $O(\log N)$ complexity. Even worse, recursive queries can incur relatively high "implementation constants" (especially on SIMD or SPMD architectures), even with careful data layouts and when avoiding pointer chasing.

5.5 ITERATIVE RANGE TREE QUERIES

In practice, the main cost of range tree queries lies not in their $O(\log N)$ complexity, but rather in the high implementation constants for recursion. As such, an iterative method would be highly preferable.

To derive such a method, we now look at a logical range tree from the bottom up, as a successive merging of respectively next-finer levels. On the finest level $L^{(0)}$, we have the $N_0 = N$ original array values, $L_i^{(0)} = A_i$. On the next level, we compute the min or max of each (complete) pair of values from the previous level, meaning there are $N_1 = \lfloor N_0/2 \rfloor$ values of $L_i^{(1)} = f\left(L_{2i}^{(0)}, L_{2i+1}^{(0)}\right)$, where f could be min or max; level 2 has $N_2 = \lfloor N_1/2 \rfloor$ such merged pairs from $L^{(1)}$, and so on. For non–power-of-two arrays, some of the N_i can be odd, meaning some nodes will not have a parent; this is somewhat counterintuitive, but for our traversal algorithm it will turn out just fine.

See Figure 5-3 for an illustration of the resulting data structure, which forms a series of binary trees (one tree if N is a power of two, and more otherwise). A node n on any level L is the root of a binary tree representing all array values within this (sub)tree.

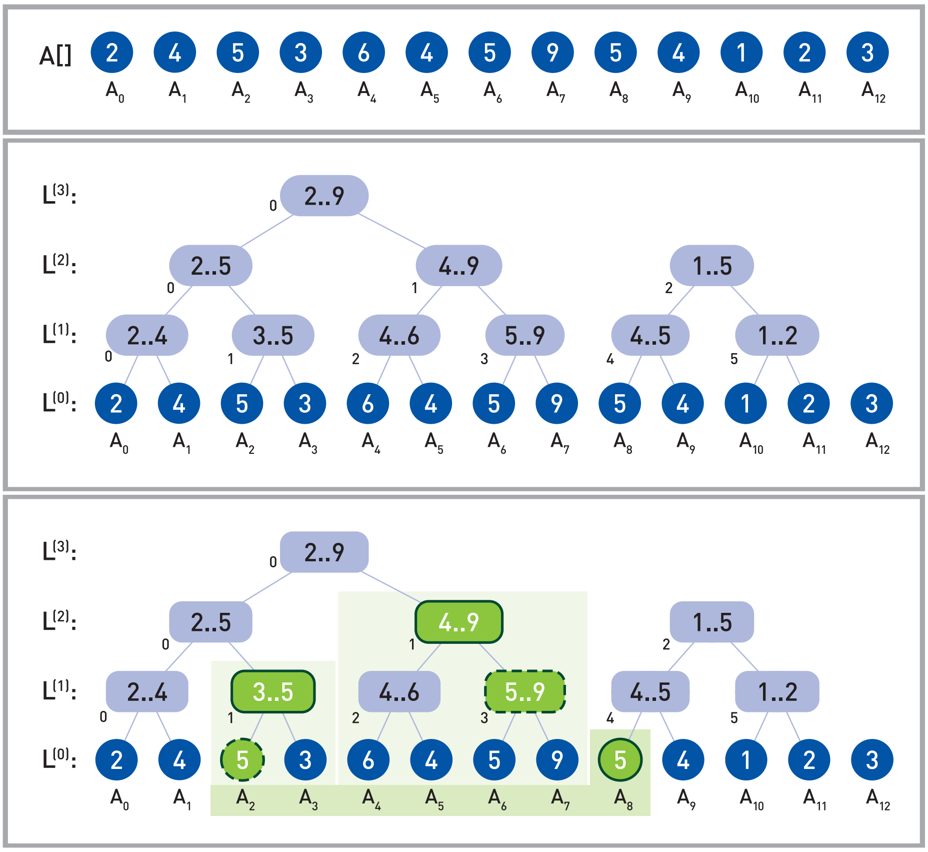

Figure 5-3. *Illustration of our iterative range tree: given an array of 13 inputs, we iteratively merge pairs to successively smaller levels, forming a total of (in this example) three binary trees. For a sample query [lo = 2, hi = 8], we must find the three nodes $L^{(0)}_8$, $L^{(1)}_1$, and $L^{(2)}_1$ marked with dark solid outlines. Our algorithm starts with lo = 2 and hi = 8 on $L^{[0]}$; it determines that hi is even and should be counted (solid circle), and that lo is odd and thus should not (dashed circle). The next step updates lo and hi to lo = 1 and hi = 3 (now in $L^{[1]}$) and correctly counts $L^{(1)}_{lo}$ (solid outline) because lo is odd, while skipping over $L^{(1)}_{hi}$ because hi is not even (dashed outline). It then does the same for lo = 1 and hi = 1 on $L^{[2]}$, after which it steps to lo = 1, hi = 0 on $L^{[3]}$ and then terminates.*

Given a query range $[lo, hi]$, let us look at all subtrees n_0, n_1, n_2, ... whose children fall completely within the query but are not part of a larger tree in the range (circled in bold in Figure 5-3). Clearly, those are the nodes we want to consider—so we need to find an efficient method of traversing those nodes.

To do this, consider the node ranges that our query range spans on each level L; let us call these $[lo_L..hi_L]$. Now, let us first look at lo_L. By construction, we know that lo_L can be the root of a subtree only if its index is odd (otherwise, it is another subtree's left child). Whether odd or even, the leftmost index in the next coarser level can be computed as $lo_{L+1} = (lo_L + 1)/2$.[3] Similar arguments can be made for the right-side index hi_L, except that "odd" and "even" get exchanged and that the next index gets computed as $hi_{L+1} = (hi + 1)/2 - 1$ (or, in signed integer arithmetic, as $(hi - 1) \gg 1$). This iterative coarsening continues until lo_L becomes larger than hi_L, at which point we have reached the first level that no longer contains any subtrees.[4] With these considerations, we end up with a simple algorithm for iterating through subtrees:

```
Iterate(lo,hi) {
    Range result = { empty range }
    L = finest level
    while (lo <= hi) {
        if (lo is odd) result = merge(result,L[lo])
        if (hi is even) result = merge(result,L[hi])
        L = next finer Level;
        lo = (lo+1)>>1
        hi = (hi-1)>>1 /* Needs signed arithmetic, else (hi+1)/2-1 */
        return result
    }
}
```

As noted in the pseudocode, care must be taken to properly handle computation of the high index when $hi = 0$, but following the pseudocode takes care of this. As in classical range trees, this iterative method accounts for each value in the input range exactly once and could thus be used for queries other than minimum and maximum.

With regard to memory layout, we have *logically* explained our algorithm using a sequence of arrays (one per level). In practice, we can easily store all levels in a single array that first contains all N_1 values for L_1, then all values for L_2, and so on. Since we always traverse from the finest to successively coarser levels, we can even compute level offsets implicitly, yielding a simple—and equally tight—inner loop. See our reference implementation online, at `http://gitlab.com/ingowald/rtgem-minmax`.

[3]Here is a brief proof. If lo_L was a root node in L then it was odd, so this moves it to the next subtree on the right side; if not, it moves up to lo_L's parent, which is still the leftmost subtree. Either way the index can be computed as $lo_{L+1} = (lo_L + 1)/2$.

[4]The case where lo_L and hi_L meet at exactly the same node is fine: the value is either odd (and counted on the low side) or even (and counted on the high side), and the next step will terminate.

5.6 RESULTS

Theoretically, our iterative method has the same storage complexity, $O(N)$, and computational complexity, $O(\log N)$, as the classical range tree method. However, its memory layout is much simpler, and the time constant for querying is significantly lower than in any recursive implementation. In fact, with our sample code this iterative version is almost as fast as the $O(1)$ sparse table method, except for tables with at least hundreds of thousands of elements—while using significantly less memory.

For example, using an array with 4k elements and randomly chosen query endpoints *lo* and *hi*, the iterative method is only about 5% slower than the sparse table method, at 10× lower memory usage. For a larger 100k-element table, the speed difference increases to roughly 30%, but at 15×; lower memory usage. While already a interesting trade-off, it is worth noting that randomly chosen query endpoints are close to the iterative method's worst case: since iteration count is logarithmic in $|hi\text{-}lo|$, “narrower” queries actually run faster than very wide ones performed by uniformly chosen lo and hi values. For example, if we limit the query values to $|hi\text{-}lo| \leq \sqrt{N}$, the iterative method on the 100k-element array changes from 30% slower to 15% faster than the sparse table method (at 15× less memory)

5.7 SUMMARY

In this chapter, we have summarized four methods for computing the minima and maxima for any sub-range of an array of numbers. The naive full table method is the easiest to implement and is fast in query—but suffers from $O(N^2)$ storage and recomputation cost, which limit its usefulness. The sparse table method is slightly more complex but significantly reduces the memory overhead, while retaining the $O(1)$ query complexity. The recursive range tree method reduces this memory overhead even more (to $O(N)$), but at the cost of a significantly higher query complexity—not only theoretically (at $O(\log N)$) but also in actual implementation constants. Finally, our iterative range tree retains the low memory overhead of range trees, uses a simpler memory layout, and converts the recursive query into a tight iterative loop. Though asymptotically still $O(\log N)$, in practice its queries perform similar to the $O(1)$ sparse table method, at lower memory consumption. Overall, this makes the iterative method our favorite, in particular since both precomputation code and query code are surprisingly simple.

Sample code for the sparse table and the iterative range tree methods are available online, at `https://gitlab.com/ingowald/rtgem-minmax`.

REFERENCES

[1] Bentley, J. L., and Friedman, J. H. A Survey of Algorithms and Data Structures for Range Searching. http://www.slac.stanford.edu/cgi-wrap/getdoc/slac-pub-2189.pdf, 1978.

[2] Bentley, J. L., and Friedman, J. H. Algorithms and Data Structures for Range Searching. *ACM Computing Surveys 11*, 4 (1979), 397–409.

[3] Crow, F. Summed-Area Tables for Texture Mapping. *Computer Graphics (SIGGRAPH) 18*, 3 (1984), 207—212.

[4] GeeksForGeeks. Min-Max Range Queries in Array. https://www.geeksforgeeks.org/min-max-range-queries-array/. Last accessed December 7, 2018.

[5] GeeksForGeeks. Segment Tree: Set 2 (Range Minimum Query). https://www.geeksforgeeks.org/segment-tree-set-1-range-minimum-query/. Last accessed December 7, 2018.

[6] GeeksForGeeks. Sparse Table. https://www.geeksforgeeks.org/sparse-table/. Last accessed December 7, 2018.

[7] Wald, I., Johnson, G. P., Amstutz, J., Brownlee, C., Knoll, A., Jeffers, J. L., Guenther, J., and Navratil, P. OSPRay—A CPU Ray Tracing Framework for Scientific Visualization. *IEEE Transactions on Visualization 23*, 1 (2017), 931–940.

[8] Wikipedia. Range Tree. https://en.wikipedia.org/wiki/Range_tree. Last accessed December 7, 2018.

[9] Wikipedia. Summed-Area Table. https://en.wikipedia.org/wiki/Summed-area_table. Last accessed December 7, 2018.