

Concurrent Binary Trees (with application to longest edge bisection)

Jonathan Dupuy

▶ **To cite this version:**

Jonathan Dupuy. Concurrent Binary Trees (with application to longest edge bisection). Proceedings of the ACM on Computer Graphics and Interactive Techniques, 2020, 3, 10.1145/3406186 . hal-02898121

HAL Id: hal-02898121
https://hal.science/hal-02898121

Submitted on 21 Jul 2020

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Concurrent Binary Trees (with application to longest edge bisection)

JONATHAN DUPUY, Unity Technologies

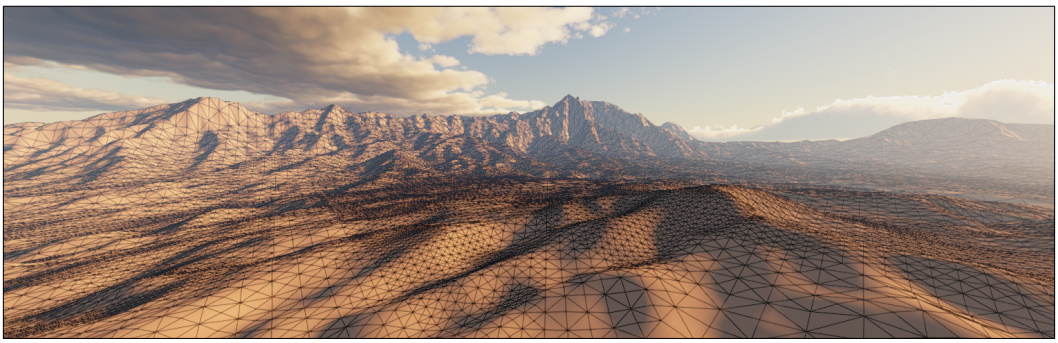


Fig. 1. A crack-free, 54×54 km terrain produced and rendered entirely on the GPU using the Unity game-engine. Internally, the engine leverages our new concurrent binary-tree data-structure to compute adaptive triangle tessellations on the GPU. Our concurrent data-structure is lightweight, easy to implement and can be used to accelerate various computer graphics applications such as this adaptive terrain renderer, which renders in less than five milliseconds at full-HD resolution on an NVIDIA RTX 2080 GPU.

We introduce the concurrent binary tree (CBT), a novel concurrent representation to build and update arbitrary binary trees in parallel. Fundamentally, our representation consists of a binary heap, i.e., a 1D array, that explicitly stores the sum-reduction tree of a bitfield. In this bitfield, each one-valued bit represents a leaf node of the binary tree encoded by the CBT, which we locate algorithmically using a binary-search over the sum-reduction. We show that this construction allows to dispatch down to one thread per leaf node and that, in turn, these threads can safely split and/or remove nodes concurrently via simple bitwise operations over the bitfield. The practical benefit of CBTs lies in their ability to accelerate binary-tree-based algorithms with parallel processors. To support this claim, we leverage our representation to accelerate a longest-edge-bisection-based algorithm that computes and renders adaptive geometry for large-scale terrains entirely on the GPU. For this specific algorithm, the CBT accelerates processing speed linearly with the number of processors.

CCS Concepts: • **Computing methodologies** → **Massively parallel algorithms**; **Rendering**.

Additional Key Words and Phrases: binary tree, concurrent, parallel, binary heap, longest edge bisection, GPU, real-time

ACM Reference Format:
Jonathan Dupuy. 2020. Concurrent Binary Trees (with application to longest edge bisection). *Proc. ACM Comput. Graph. Interact. Tech.* 3, 2, Article 21 (August 2020), 20 pages. https://doi.org/10.1145/3406186

Author's address: Jonathan Dupuy, Unity Technologies, jonathan.dupuy@outlook.com.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
© 2020 Copyright held by the owner/author(s). Publication rights licensed to ACM.
2577-6193/2020/8-ART21 $15.00
https://doi.org/10.1145/3406186

1 INTRODUCTION

Motivation. Subdivision is a family of recursive algorithms that exhibits some form of exponential growth with respect to recursion depth. Such behaviors appear in an important number of applications in computer graphics, including (but not restricted to) quadtrees, octrees and kd-trees, subdivision curves and surfaces, as well as recursive ray/path tracing. Due to their fundamentally exponential nature, the computational cost induced by such applications as subdivision depth increases can quickly become a limitation. In order to amortize this cost, a solution consists in evaluating subdivision both adaptively and in parallel. While adaptive subdivision is straightforward to implement sequentially, it proves to be a challenge to couple with parallel processing in the general case. In this work, we are interested in addressing this difficulty by introducing a data-structure suitable for processing adaptive subdivision on parallel processors.

Subdivision as Binary Trees. Our work builds upon the observation that the *canonical* subdivision algorithm is that of the binary tree (whose number of leaf nodes doubles at each recursion step). The binary tree is *canonical* in the sense that any subdivision algorithm can be viewed as a binary tree: Under the binary tree interpretation, the leaf nodes describe the recursive state via the path they form from the root node. It follows that if we are able to process the leaf nodes of a binary tree in parallel, then we have a means to accelerate any subdivision algorithm. Surprisingly, the computer graphics literature seems to lack contributions towards this direction, the closest work being dedicated to constructing BVHs in parallel [Apetrei 2014; Garanzha et al. 2011; Karras 2012]. Unfortunately, these specific trees are unsuitable here as they lack the ability to evolve their topology through time.

Contributions and Outline. In the following sections, we introduce a novel binary tree data-structure, which we refer to as a concurrent binary tree (CBT), that is particularly suited for building and updating binary tree topologies in parallel. CBT are very simple to implement: they solely consist of a binary heap, i.e., a 1D array (Section 2), of 2^{D+1} non-negative integer values, where $D \geq 0$ denotes the maximum subdivision depth of the binary tree we wish to represent; Figure 4 illustrates the structure of a CBT in the case $D = 4$. A CBT carries the two following key properties, both of which are illustrated in the video that supplements this article:

- The last 2^D elements of the CBT implicitly describe a binary tree of maximum depth D using binary values, where any element set to one corresponds to a leaf node. We split and/or merge such leaf nodes in parallel using atomic bitwise operations.
- Each element of the first 2^D elements of the CBT stores the sum of its two children. This allows to retrieve the i-th leaf node of the implicit binary tree via a binary-search algorithm, which runs in $O(D)$ worst-case complexity.

The main benefit of CBTs is that their processing speed increases linearly with the number of processors. We derive and demonstrate this effectiveness throughout the remainder of this article as follows:

- We introduce the CBT data-structure along with dedicated algorithms for iterating over, splitting, and/or merging the leaf nodes of the binary trees they encode (Section 3).
- We derive a generic pipeline suitable for processing CBTs in parallel and show that its processing speed increases linearly with the number of threads (Section 4).
- We leverage our CBT data-structure to compute adaptive subdivisions using the longest edge bisection subdivision algorithm in the context of terrain rendering (Section 5).

Note that to emphasize on the practical aspect of our CBT data-structure, we provide all the algorithms suitable for writing an implementation of CBTs; some source code is also available online: https://github.com/jdupuy/LongestEdgeBisection2D.

2 PERFECT BINARY TREES AS BINARY HEAPS

The CBT data-structure is closely linked to the binary heap data-structure. In this section, we recall the fundamental properties of binary heaps that CBTs build upon (Section 2.1). Next, we provide a high-level overview of the CBT data-structure to position our contribution with respect to binary heaps (Section 2.2).

2.1 Properties and Definition

A binary tree whose leaf nodes are of same depth forms a perfect binary tree, which has a total of $2^{D+1} - 1$ nodes, where $D \geq 0$ denotes the depth of the leaf nodes; Figure 2 (a) illustrates the geometry of a perfect binary tree with leaf nodes of depth $D = 4$. A binary heap builds upon an implicit representation for the topology of such trees, which takes the form of an indexing scheme that we refer to as *heap indexing*. Heap indexing works as follows: it starts at index 1 for the root node, and then proceeds with breadth-first increments; Figure 2 (a) shows nodes labelled according to heap indexing. Such an approach yields the following properties:

Algebraic Relationships. Given a node with heap index $k \geq 1$, its parent node has heap index $\lfloor k/2 \rfloor$ and its children have respective heap indexes $2k$ and $2k + 1$; Figure 2 (a) illustrates these relationships. It follows that at subdivision depth $d \geq 0$, the heap indexes range from 2^d to $2^{d+1} - 1$. Additionally, we can retrieve the subdivision depth of any heap index by taking the integer part of its base-2 logarithm, i.e., $d_k = \lfloor \log_2(k) \rfloor =: \textsc{FindMSB}(k)$.

Implicit Path Encoding. The binary representation of any heap index carries the entire path from the root node to the node it is actually indexing, where 0 denotes a one-level descent towards the left child, and 1 towards the right child. This path is preceded by a most-significant-bit set to 1, whose bit offset is equal to the node's subdivision depth. For instance, the node with heap index $27 = 11011_b$ has most-significant-bit located at bit offset (and hence subdivision depth) $d_{27} = \textsc{FindMSB}(27) = 4$ and path 1011, i.e., right-left-right-right from the root node; Figure 2 (a) illustrates the link between the binary representation of a heap index and the path it encodes.

Binary Heap Definition. A binary heap of maximum depth $D \geq 0$ is a memory-representation for a perfect binary tree of depth D. It solely consists of an array of fixed-size 2^{D+1}, whose elements are sorted as follows:

- The first element has heap index 0 and stores the depth D of the binary tree.
- The remainder elements store the data associated with each node of the tree, ordered according to their heap index, i.e., the root node's data is stored at array index 1, etc.

Thanks to this construction, binary heaps provide a pointerless implementation for perfect binary trees that trivially supports node iteration. In addition, the fact that binary heaps are pointerless make them appealing for parallel processing on the GPU because GPUs rarely provide support for pointers (or do so inefficiently).

2.2 Relation to CBTs

The fundamental limitation of binary heaps is that they are restricted to perfect binary trees: while it is perfectly valid to alter node data within a binary tree stored as a binary heap, its topology has to remain fixed to that of a perfect binary tree. CBTs provide a means to alleviate this limitation by providing a pointerless representation for arbitrary binary trees whose topology can be changed through time; Figure 2 (b) shows an example of an arbitrary binary tree. Interestingly, CBTs are also binary heaps. This implies a somewhat surprising result: the topology of an arbitrary binary tree can be represented using a perfect binary tree.

3 CBT REPRESENTATION

In this section, we derive the CBT data-structure in a self-contained way. First, we show how to encode an arbitrary binary tree as a bitfield (Section 3.1). In this bitfield, each bit set to one corresponds to a leaf node, and we show how to efficiently iterate over each one of them using the reduced-sum of this bitfield (Section 3.2). The association of the bitfield and the reduced-sum produces a perfect binary tree that we store as a compact binary heap; this compact binary heap effectively forms a CBT (Section 3.3). Note that this section is solely dedicated to the memory representation of CBTs and we reserve the discussion of its concurrent properties for the next section.

3.1 Binary Trees as Bitfields

Here, we show how to encode a binary tree using a bitfield; an informal presentation of the bitfield is provided in the video that supplements this article.

Bitfield Construction. We represent a binary tree of maximum depth $D \geq 0$ using a bitfield of size 2^D. In this bitfield, each bit set to one encodes a leaf node that we determine unambiguously based on the number of zeroes that follow; Figure 2 provides two examples of binary trees encoded as bitfields for the case $D = 4$. Note that we decode each node by retrieving its associated heap index $k \in [1, 2^{D+1} - 1]$. Since the heap indexes we decode carry the path from the root node to each leaf node as discussed in Section 2, we therefore have access to the entire topology of the binary tree.

Bit to Heap Index. The bit located at index $x \in [0, 2^D - 1]$ encodes up to $N = \text{FindLSB}(x) + 1$ nodes if $x > 0$ and $N = D + 1$ otherwise, where FindLSB returns the index of the least significant bit; this property is depicted visually in Figure 2, where each bit may encode any node located above it, e.g., the bit indexes 0, 4 and 14 respectively encode up to 5, 3, and 2 different nodes whose heap indexes are $\{16, 8, 4, 2, 1\}$, $\{20, 10, 5\}$, and $\{30, 15\}$. We can determine which node is actually encoded by counting the number of zeroes $N_0 \in [0, 2^D - 1]$ that follow the bit. Then, the corresponding node has subdivision depth $d = D - \log_2(1 + N_0)$. Since each bit may encode only one node per subdivision depth, the bitfield provides an unambiguous mapping from bit to heap index.

Heap to Bit Index. In order to encode a binary tree within our bitfield representation, we simply set one bit to one per leaf node. We determine which bit to set for each leaf node as follows: Given a leaf node with heap index k, the bit that encodes it has index $x_k = k \times 2^{D-d_k} - 2^D$, where $d_k = \text{FindMSB}(k)$. This construction allows to implement node splitting and merging operations straightforwardly.

Node Splitting. Splitting a given node with heap index k translates into setting the bit that encodes its right child, which has index $2k + 1$, to one; Figure 3 shows the impact of node splitting on the bitfield. What is particularly powerful with this construction is that it is insensitive to splitting non-leaf nodes: in such cases, we are setting a bit to one multiple times.

Node Merging. Merging two sibling nodes with heap indexes $\{2^d, 2^d + 1\}$, $d \in [1, D]$ translates into setting the bit that encodes the right sibling with bit index x_{2^d+1} to zero; Figure 3 shows the impact of node merging on the bitfield. What is particularly powerful with this construction is that it is insensitive to merging non-existent nodes: in such cases, we are setting a bit to zero multiple times.

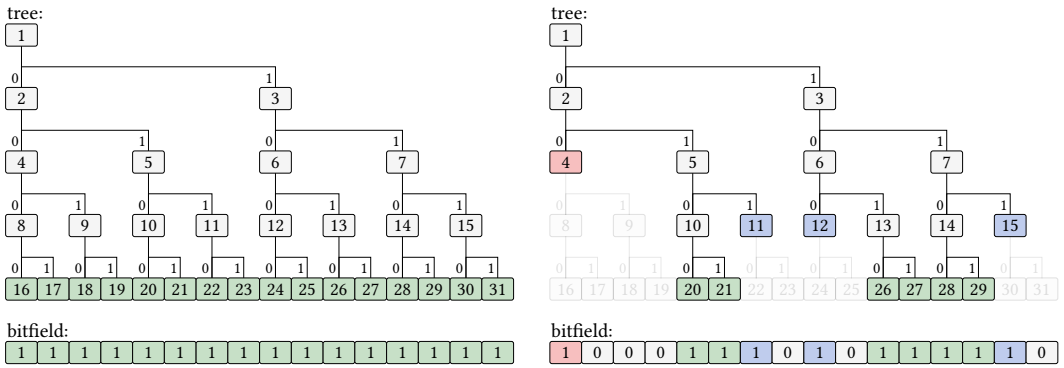


Fig. 2. (top) Illustration of (left) a perfect binary tree and (right) an arbitrary binary tree. (bottom) Equivalent bitfield representation. The tree nodes are labelled according to their binary heap index and the leaf nodes are colored according to their depth (the depths for the red, blue, and green nodes are respectively 2, 3, and 4). Each bit set to one in the bitfield is colored according to the leaf node it encodes in the binary tree.

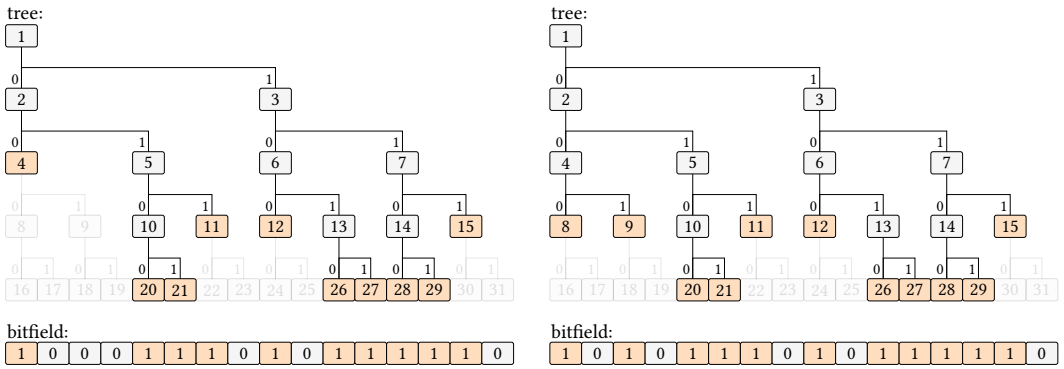


Fig. 3. Impact of (left to right) node-splitting and (right to left) node-merging on the bitfield representation of a binary tree. Under the bitfield representation, node splitting and merging effectively translate into setting bits to one and zero, respectively (the third bit is affected in this illustration).

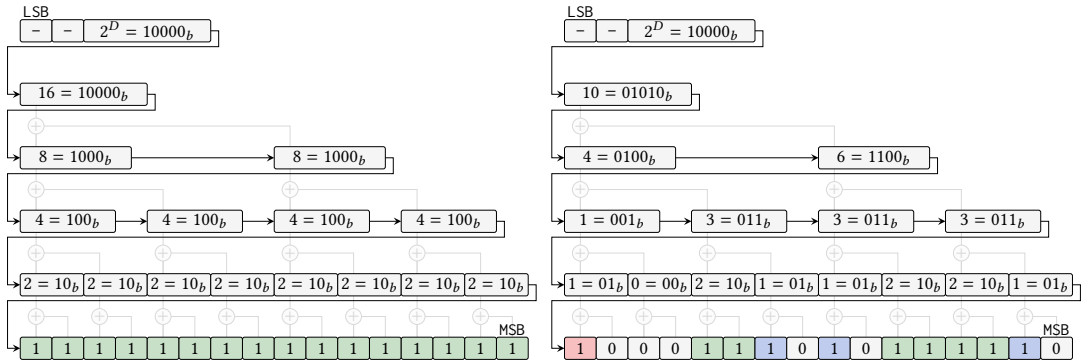


Fig. 4. Optimized memory representation of CBTs of maximum depth 4. Both CBTs encode the binary trees from Figure 2, which are constructed bottom-up starting from their respective bitfields. The width of each block is proportional to its bit-size, where the smallest block is 1-bit wide. The black arrows and gray binary tree structure respectively denote block contiguity and summing via reduction.

3.2 Augmenting the Bitfield to a Binary Heap

The bitfield representation from the previous section provides a means to encode and split and/or merge leaf nodes. What it lacks however is a means to efficiently iterate over the actual leaf nodes it encodes. This property is especially relevant for large bitfields, as naive bit iterations over the bitfield grow exponentially with maximum depth.

Introducing the Binary Heap. We augment the bitfield representation to support leaf-node iteration: In order to iterate over the one-bits of a bitfield of size 2^D, we further compute and store the reduced-sum of this bitfield. This process can be seen as a bottom-up construction of a perfect binary tree of depth D, where each leaf node corresponds to a bit in the bitfield. Since we are dealing with a perfect binary tree, we store it as a binary heap such that the values are stored in breadth-first order starting from the root (see Section 2); Figure 4 provides the constructions of two such contiguous memory blocks in the case $D = 4$. We refer to this binary heap as a CBT of maximum depth D, and discuss how we quickly retrieve the leaf nodes of the tree it encodes among other properties in the following paragraphs; an informal presentation of the CBT is provided in the video that supplements this article.

Maximum Depth of a CBT. Following the binary heap construction from Section 2.1, we store the maximum depth D of the CBT as its first element at heap index 0. Note that this maximum depth is set once at the instantiation of a CBT and can not be changed after that.

Number of Leaf Nodes. By construction, the second element of the CBT, which is located at heap index 1, is the number of leaf nodes $L \in [0, 2^D]$ it encodes. This property is a first step towards having the ability to iterate over leaf nodes (see Algorithm 4, line 2).

Leaf to Heap Index. Given a CBT with L leaf nodes, we retrieve the l-th leaf node heap index, $l \in [0, L-1]$, using a binary search algorithm that runs in $O(D)$ worst-case complexity; Algorithm 1 provides our binary search algorithm. The binary-search algorithm is what allows to quickly iterate over the leaf nodes of the CBT (see Algorithm 4, lines 3-6).

Algorithm 1 Leaf- to heap-index using a binary search

```
1:  function DecodeNode(CBT: heap, leafID: int)
2:      heapID ⟵ 1                                  ▷ initialize to root node
3:      while CBT[heapID] > 1 do                    ▷ binary search loop
4:          if leafID < CBT[heapID] then            ▷ left child
5:              heapID ⟵ 2 × heapID
6:          else                                    ▷ right child
7:              leafID ⟵ leafID − CBT[heapID]
8:              heapID ⟵ 2 × heapID + 1
9:          end if
10:     end while
11:     return heapID
12: end function
```

3.3 Memory Considerations

In this section, we show how we allocate and access the values stored within a CBT using a minimal amount of memory. Note that this is not required for understanding the inner workings of the CBT.

Rather, we provide it for the sake of completeness and to quantify the memory requirements of CBTs.

Memory Layout. By construction, the deepest elements of the CBT are binary values. Therefore we can bound the values produced by the sum reduction at any level, and hence the number of bits required to represent them within a CBT. More specifically, any value at depth $d \in [0, D-1]$ requires exactly $N_d = D - d + 1$ bits. Thus a CBT of maximum depth D requires N_b bits, where

$$\begin{aligned} N_b &= \sum_{d=0}^{d \leq D} 2^d N_d \\ &= 2^{D+2} - (D+3). \end{aligned} \tag{1}$$

In practice, we also allocate $D+1$ bits to store the maximum depth of the CBT at heap index 0, and 2 additional bits to round up the storage requirement to exactly 2^{D+2} bits; Figure 4 illustrates the memory layout of a CBT in the case $D = 4$, where the 2 additional bits we store are shown on the top row. Note that although CBTs grow exponentially in size, they remain quite compact at moderate depths, as, e.g., a CBT of maximum depth $D = 16$ requires simply 32 KiB of memory.

Data-Access. In order to access the k-th element of the CBT, we need to compute its bit range. We already know from the previous paragraph that the k-th element consumes $N_{d_k} = D - d_k + 1$ bits, where $d_k = \lfloor \log_2(k) \rfloor$ and $d_0 = 0$. Therefore, we only require the bit-offset $x_k \in [2, 2^D - 1]$ of the k-th element, which is given by

$$x_k = 2^{d_k+1} + k \times N_{d_k}. \tag{2}$$

Therefore, the k-th element has bit-range $[x_k,\ x_k + N_{d_k}]$. As an example, the first bit of the bitfield that encodes the leaf nodes of the CBT has heap index $k = 2^D$ and is thus located within the bit-range $[3 \times 2^D,\ 3 \times 2^D + 1]$ of the CBT.

4 CBT PROCESSING

With the presentation of the CBT data-structure complete, we now describe a generic pipeline for initializing and updating it in parallel. Both initialization and update consist of an iteration step over the bitfield, followed by a sum-reduction. We describe the latter step first and then focus on the details of the former depending on whether we are initializing or updating the binary tree. We start with a purely sequential exposition (Section 4.1). Next, we describe a generic pipeline suitable for updating the CBT in parallel, possibly on the GPU (Section 4.2). Finally, we provide performance measurements to assess the scalability of CBTs over multiprocessors (Section 4.3).

4.1 Initialization and Update

Sum-Reduction Step. Algorithm 2 provides pseudocode for an implementation of the sum-reduction step. Given a CBT of maximum depth D, we update the reduced sum following a classic reduction technique over the 2^D bits of the bitfield, which parallelizes well on GPUs [Harris et al. 2007]; Figure 4 illustrates the relation between each block of the reduced sum memory. In practice, this step requires $D-1$ steps. Starting from $d = D-1$, each step performs 2^d additions until reaching the top-level element that provides the total number of leaf nodes encoded in the CBT. The sum-reduction runs at time-complexity $O(D + 2^D/P)$, where $P \geq 1$ denotes the number of processors.

Initialization. Algorithm 3 provides pseudocode for an implementation. Given a CBT of maximum depth D, our initialization routine builds a binary tree starting at a uniform subdivision depth $d \in [0, D]$. To achieve this, we set every 2^{D-d} bit to one and others to zero in the CBT's bitfield;

Figure 4 (a) provides an example of initialization for the case where $d = D = 4$. Once this step complete, we launch the reduction step, which effectively completes the initialization process.

Update. Algorithm 4 provides pseudocode for an implementation. The update step simply iterates over the leaf nodes that, in turn, can invoke node splitting or merging operations in the CBT. The decision to invoke such operations is entirely generic and left to the user. Once the iteration complete, we launch a reduction step, which effectively completes the update.

Algorithm 2 Sum-reduction tree computation

```
1:  procedure ComputeSumReduction(CBT: heap)
2:      D ← MaxDepth(CBT)                                   ▷ see Alg. 9
3:      d ← D − 1
4:      while d ≥ 0 do                                      ▷ depth loop
5:          for all k ∈ [2^d, 2^(d+1)) do                   ▷ reduction loop
6:              CBT[k] = CBT[2k] + CBT[2k + 1]              ▷ sum
7:          end for
8:          d ← d − 1
9:      end while
10: end procedure
```

Algorithm 3 Specific depth initialization

```
1:  procedure InitAtDepth(CBT: heap, d: depth)
2:      D ← MaxDepth(CBT)                                   ▷ see Alg. 9
3:      Memset(CBT, 0)
4:      for all heapID ∈ [2^d, 2^(d+1) − 1] do
5:          CBT[heapID × 2^(D−d)] = 1
6:      end for
7:      ComputeSumReduction(CBT)                            ▷ see Alg. 2
8:  end procedure
```

Algorithm 4 Typical update routine

```
1:  procedure Update(CBT: heap)
2:      L ← NodeCount(CBT)                                  ▷ see Alg. 10
3:      for all leafID ∈ [0, L − 1] do                      ▷ leaf node iteration
4:          heapID ← DecodeNode(CBT, leafID)                ▷ see Alg. 1
5:          UserSpecificCallback(CBT, heapID)
6:      end for
7:      ComputeSumReduction(CBT)                            ▷ see Alg. 2
8:  end procedure
```

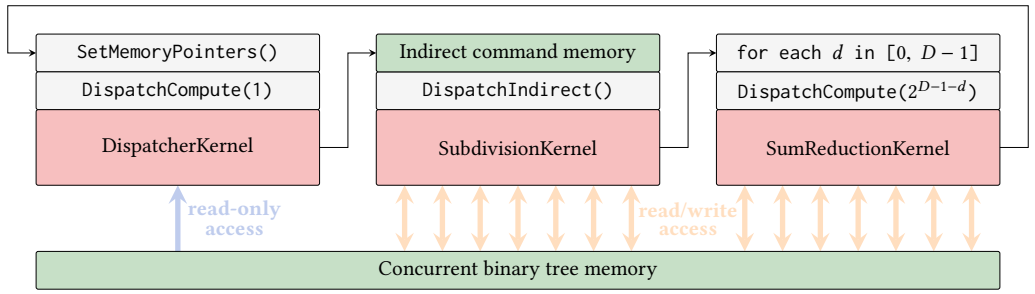


Fig. 5. Generic pipeline for updating our binary tree representation in parallel. Each kernel is implicitly followed by a memory barrier operation.

4.2 Updating in Parallel

The algorithms from the previous section can be evaluated in parallel using a parallel-for paradigm. Here, we provide a parallel and GPU-compatible processing pipeline for the update stage based on this paradigm. Thanks to the concurrent nature of CBTs, only infinitesimal precautions are needed to accelerate their sequential processing. As a result, our pipeline consists of three kernels, the combination of which allows us to progressively compute a binary tree in parallel; Figure 5 diagrams our generic pipeline. We refer to these three kernels as respectively the *dispatcher* kernel, the *subdivision* kernel, and the *sum-reduction* kernel, which we now describe independently:

Dispatcher Kernel. The dispatcher kernel is only relevant for an asynchronous GPU implementation, which typically relies on indirect commands provided by modern APIs. It invokes a single thread that is responsible for preparing a command for the subdivision kernel. In practice, this simply consists in writing the number of leaf-nodes, i.e., an integer, to a small buffer that stores the arguments of an indirect command, which is used to invoke threads for the next kernel.

Subdivision Kernel. The subdivision kernel is responsible for deciding whether each leaf node should split, merge or remain at the same level. It effectively implements a parallel-for loop over the leaf nodes encoded by the CBT, as shown in Algorithm 4, lines 3-6. This kernel is the only one that requires special care in the context of parallel-processing: As it may result in concurrent split and or merge operations of the same leaf nodes, we simply require that the induced bitfield operations be made atomic (see Algorithm 12). Note that this is straightforward to implement.

Sum-Reduction Kernel. The sum-reduction kernel is a standard reduction algorithm, which implements the reduction step. The kernel is executed $D - 1$ times, where $D \geq 0$ represents the maximum depth of the processed CBT, and each execution invokes 2^{D-1-d} threads, where $d \in [0, D-1]$ denotes the d-th kernel execution. It effectively implements a parallel-for loop over the nodes encoded by the CBT, as shown in Algorithm 2, lines 5-7.

4.3 Synthetic Performance Measurements

The algorithms we have introduced so far provide a means to implement an application based on dynamic binary tree topologies that exploits multiprocessors. In order to quantify the scalability of CBT-processing across processor count, we provide here some performance measurements for both the subdivision and sum-reduction kernels. We conducted the measurements on both a CPU-based and GPU-based platform using dedicated implementations written respectively in C with OpenMP and GLSL450 shaders.

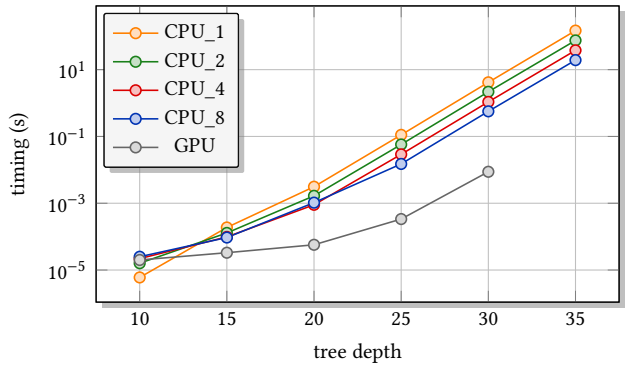


Fig. 6. Sum-reduction computation timings as a function of tree depth for varying processor counts and architectures. The CPU is an AMD Ryzen Threadripper 3960X (24-cores) and the GPU an NVIDIA RTX 2080.

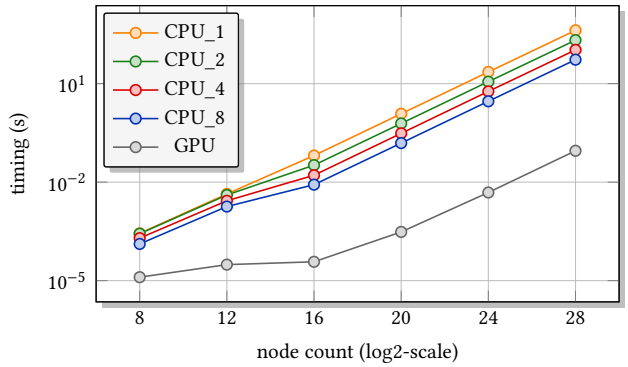


Fig. 7. Subdivision kernel timings as a function of node count for varying processor counts and architectures. The CPU is an AMD Ryzen Threadripper 3960X (24-cores) and the GPU an NVIDIA RTX 2080.

Sum-Reduction Kernel. We measured the performance of the sum-reduction kernel by initializing CBTs of varying maximum depth with procedurally-filled bitfields and timing the computing of their sum-reduction tree, i.e., our parallel implementation of Algorithm 2; Figure 6 provides the results of our measurements. Note that since the sum-reduction kernel is independent from the way CBTs are used, we expect its performance and scalability to remain approximately constant on these platforms.

Subdivision Kernel. The subdivision kernel is highly dependent on the user-specific callback that determines whether a node should be split or merged (see Algorithm 4, line 5). In order to provide a significant measurement, we implemented a callback that stochastically splits or merges nodes based on their heap index. Therefore, our performance measurements for this specific kernel should only be seen as a measure of scalability. Figure 7 provides the results of our measurements.

Scalability. As demonstrated by the reported timings, the processing speed of CBTs increases linearly with the number of threads. We also mention that for deep binary trees, the sum-reduction kernel quickly becomes the bottleneck. In practice, we believe it is safe to expect a sum-reduction bottleneck whenever the bitfield of the CBT is sparse. We leave it up to the user to determine the best maximum depth of the CBT based on his application requirements.

5 APPLICATION: LONGEST EDGE BISECTION

In this section, we show how to leverage CBTs to compute adaptive triangle subdivisions using the longest edge bisection (LEB) algorithm. First, we provide some preliminary background on the LEB algorithm and show how CBTs are relevant for its computation (Section 5.1). In practice, CBTs provide a means to symbolically manipulate the triangles produced by LEB as heap indexes. We provide all the necessary algorithms to compute adaptive subdivisions (Section 5.2). Finally we provide all the implementation details suitable for reproducing our results, along with performance measurements (Section 5.3).

5.1 Preliminaries

LEB is a subdivision algorithm that consists in recursively splitting triangles in two along their longest edge; Figure 8 (a) illustrates how LEB operates on a triangle. Note that LEB is quite popular in the scientific literature, although it tends to appear under different names. Possible names include, e.g., bisection refinement [Maubach 1995], triangle bintrees [Duchaineau et al. 1997], 4-8 refinement [Velho 2000; Velho and Zorin 2001], diamonds [Weiss and De Floriani 2011; Yalcin et al. 2011], right-triangulated irregular networks [Evans et al. 2001; Lindstrom et al. 1996], hierarchical simplicial meshes [Atalay and Mount 2007], and finally longest-edge bisection [Lindstrom and Pascucci 2002; Özturan 1996; Rivara 1984]. The reason we choose LEB over any of the aforementioned names is because it best describes how the subdivision operates. LEB carries the following properties:

Conforming-Adaptive Tessellations. This is a rare and powerful property that motivated us to implement LEB in the first place: When applied adaptively, LEB produces conforming triangle tessellations, i.e., free of T-junctions, whenever it is constrained so as to guarantee that no neighboring triangles differ by more than one subdivision level; Figure 8 (c) illustrates this property for the green and blue triangles. As such, LEB is fundamentally suited for building multi-resolution tessellations. In order to satisfy such a constraint, it is known [Rivara 1984] that each triangle subdivision should propagate along a path of $d \geq 0$ triangles, where d denotes subdivision depth; Figure 10 illustrates this property.

Link to Binary Trees. The canonical primitive of LEB is the isosceles right triangle, which splits into 2 similar triangles at each subdivision step according to the splitting matrices[1]

$$M_0 = \begin{bmatrix} 1 & 0 & 0 \\ \frac{1}{2} & 0 & \frac{1}{2} \\ 0 & 1 & 0 \end{bmatrix}, \quad \text{and} \quad M_1 = \begin{bmatrix} 0 & 1 & 0 \\ \frac{1}{2} & 0 & \frac{1}{2} \\ 0 & 0 & 1 \end{bmatrix}; \tag{3}$$

the result of this process is illustrated in Figure 8 (a) after 1 subdivision step and in Figure 8 (b) after 4 subdivision steps. LEB is thus a self-similar, binary process that runs recursively. As such, it can be seen as a binary tree, where each node of the tree corresponds to a specific triangle produced by the subdivision.

Motivation for CBTs. We observe that the binary nature of LEB allows to transpose the problem of computing it into that of modifying the topology of a binary tree. Under the binary tree interpretation, each triangle produced by LEB corresponds to a node in the binary tree, and only the leaf nodes form the actual geometry; Figure 8 (a, b) illustrates the tessellations produced by LEB associated to the binary trees from Figure 2. It follows that CBTs provide an ideal tool to evaluate LEB by providing a means to manipulate the triangles it produces symbolically as heap indexes; the

[1]The LEB splitting matrices apply to the vertices of the input triangle. See Figure 8 (a) for instance, where we have $(A_0, B_0, C_0)^T = M_0 \times (A, B, C)^T$ and $(A_1, B_1, C_1)^T = M_1 \times (A, B, C)^T$.

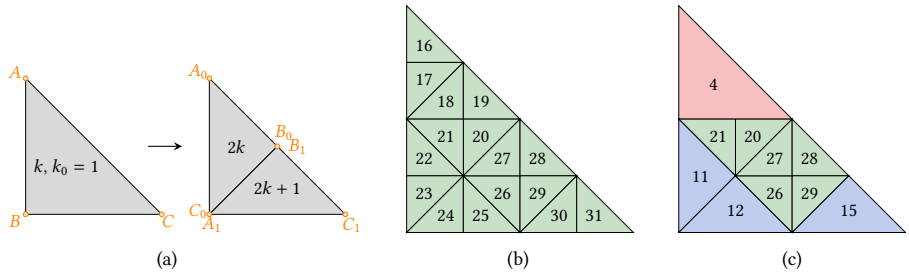


Fig. 8. (a) LEB rule applied (b) uniformly, and (c) adaptively. The subdivision depths for the red, blue, and green triangles are respectively 2, 3, and 4.

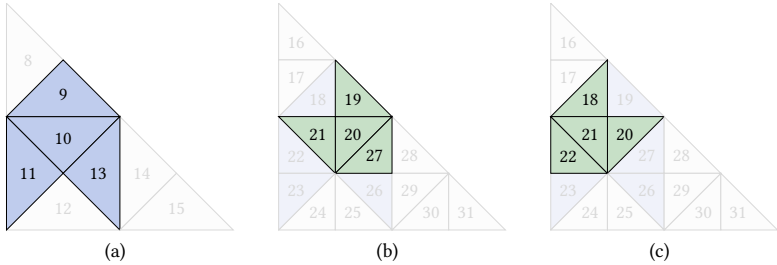


Fig. 9. Geometry of (a) a triangle (index 10) and its direct neighbors (index 9, 11, and 13) compared to its (b) left child (index 20) and (c) right child (index 21). Notice the self-similarity of the configuration across the subdivision level.

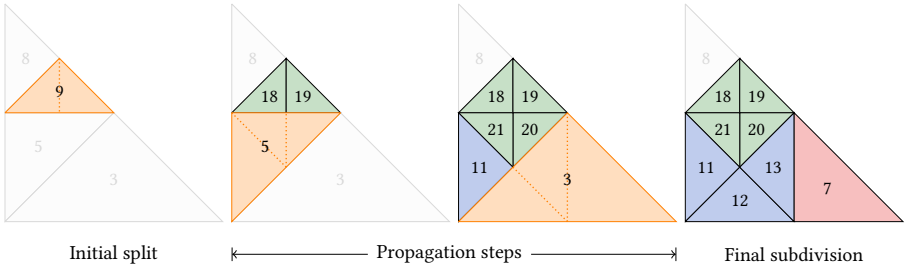


Fig. 10. LEB produces conforming geometry whenever a triangle split is propagated along the path that connects longest-edge neighbors. Here, the propagation path consists of index 10, 5, 6, and 3. Each propagation step involves exactly 2 splits within the same triangle.

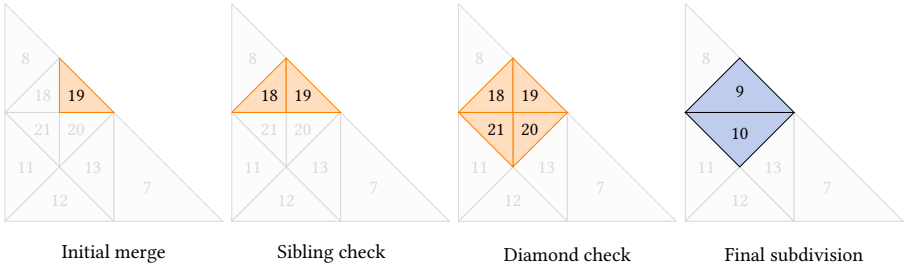


Fig. 11. LEB produces conforming geometry whenever a triangle merge results in decimating an edge of a diamond-like configuration. Here, the diamond-configuration consists of index 18, 19, 20, and 21. LEB merging effectively merges 4 triangles into 2.

triangles from Figure 8 are labelled according to their heap indexes. For the sake of completeness, we show below how to decode the triangle produced by LEB from its associated heap index:

Heap Index to Transformation Matrix. Any triangle produced by LEB is similar to the (canonical) root triangle. As such, they can be linked to one-another via a transformation matrix. Given a triangle with heap index $k \geq 1$, we retrieve the transformation by following the path it encodes through the binary tree and chaining the proper splitting matrix from Equation (3) via multiplication at each level. For instance, the transformation matrix associated with heap index $20 = 10100_b$ is $M_{20} = M_0 \times M_0 \times M_1 \times M_0$ (notice the reversed multiplication order with respect to the binary code). Algorithm 5 provides pseudocode for an implementation.

Algorithm 5 LEB matrix computation

```
1: function LebMatrix(heapID: int)
2:     M ← Identity()
3:     d ← FindMSB(heapID)
4:     for bitID ← d − 1; bitID ≥ 0; bitID ← bitID - 1 do
5:         b ← GetBitValue(heapID, bitID)
6:         M ← M_b × M
7:     end for
8:     return M
9: end function
```

5.2 Dedicated LEB Algorithms for CBTs

Here, we provide the algorithms suitable for evaluating LEB adaptively using a CBT. In order to produce both adaptive and conforming LEBs, we use a specific procedure depending on whether we consider triangle splitting or merging. Both procedures rely on a function that retrieves the heap index of neighboring triangles, so we first introduce this particular function before focusing on the procedures.

Same Depth Neighboring Heap Indexes. Any triangle produced by LEB has at most three direct neighbors (one per edge); Figure 9 illustrates the typical geometry of this direct neighborhood. Given a triangle with heap index $k \geq 1$, the neighbor's heap indexes are retrieved by following the path encoded by k and mapping the initial 4-vector $\mathbf{n} = (\emptyset, \emptyset, \emptyset, 1)$ via compositions of either of the following map:

$$g_0(\mathbf{n}) = (2n_4 + 1,\ 2n_3 + 1,\ 2n_2 + 1,\ 2n_4\ \ ), \tag{4}$$

$$g_1(\mathbf{n}) = (2n_3\ \ ,\ 2n_4\ \ ,\ 2n_1\ \ ,\ 2n_4 + 1), \tag{5}$$

using the algebraic rule $2\emptyset = \emptyset + 1 = \emptyset$. Then, the element n_3 corresponds to the longest-edge neighbor, n_1 and n_2 the 2 others, and $n_4 = k$. For instance, the node index $20 = 10100_b$ yields the 4-vector

$$(g_0 \circ g_1 \circ g_0 \circ g_0)(\mathbf{n}) = (21,\ 19,\ 27,\ 20);$$

Figure 9 (b) illustrates this particular configuration, and Algorithm 6 provides pseudocode for an implementation. Note that the neighboring heap indexes computed here correspond to those located at the same subdivision level of that of the input node.

Algorithm 6 LEB same-depth neighboring heap index computation

```
1: function LebNeighbors(heapID: int)
2:     n ← (∅, ∅, ∅, 1)
3:     d ← FindMSB(heapID)
4:     for bitID ← d − 1; bitID ≥ 0; bitID ← bitID - 1 do
5:         b ← GetBitValue(heapID, bitID)
6:         n ← g_b(n)
7:     end for
8:     return n
9: end function
```

Conforming Splitting. This operation consists in splitting a given triangle produced by LEB into its two children, which is relevant for producing finer geometry. Given a conforming LEB, splitting a specific triangle will produce a finer conforming LEB if and only if the splitting is propagated along the path that solely consists of longest-edge neighbors; Figure 10 illustrates this procedure. Note that it follows trivially that the propagation path is always unique (since triangles have only one longest-edge neighbor) and consists of at most $d \geq 0$ triangles, where d denotes the subdivision depth of the LEB triangle considered for splitting. Note also that each propagation step involves exactly 2 splits within the same triangle. Under the binary tree interpretation, conforming splitting results in invoking several node splits throughout the binary tree. Algorithm 7 provides pseudocode for an implementation based on CBTs. CBTs provide a means to implement such an operation straightforwardly despite the topology constraints it imposes.

Algorithm 7 Conforming node splitting for LEB

```
1: procedure LebSplit(CBT: heap, heapID: int)
2:     SplitNode(CBT, heapID)                          ▷ split the actual triangle
3:     heapID ← LebNeighbors(heapID).edge              ▷ get longest edge neighbor
4:     while heapID > 1 do                             ▷ propagation for conforming tessellations
5:         SplitNode(CBT, heapID)
6:         heapID ← heapID/2
7:         SplitNode(CBT, heapID)
8:         heapID ← LebNeighbors(heapID).edge
9:     end while
10: end procedure
```

Conforming Merging. This operation consists in merging two sibling triangles into their original triangle, which is relevant for producing coarser geometry. In this respect, conforming triangle-merging can be seen as the undoing of a conforming-split operation and can therefore be implemented as such. In practice however, this approach leads to ambiguous situations whenever a triangle located along the propagation path requires splitting [Duchaineau et al. 1997]. Therefore, we restrict conforming-merging propagation to an edge collapse over a diamond-like configuration; Figure 11 illustrates how the conforming merge process operates. Such an approach requires the two following conditions to be met. First, the sibling of the triangle considered for merging should exist. Second, the longest-edge neighbor of the parent triangle should be split exactly once, thereby forming the diamond-like configuration, which involves two pairs of sibling triangles. Then, we implement the conforming merging of these two sibling pairs by collapsing the edge

of the diamond-like configuration they form. Under the binary tree interpretation, conforming merging results in invoking two node merges throughout the binary tree. Algorithm 8 provides pseudocode for an implementation based on CBTs. CBTs provide a means to implement such an operation straightforwardly despite the topology constraints it imposes.

Algorithm 8 Conforming node merging for LEB

```
1:  procedure LebMerge(CBT: heap, heapID: int)
2:      siblingID ← Xor(heapID, 1)
3:      diamondID ← LebNeighbors(heapID/2).edge
4:      leftID ← diamondID × 2
5:      rightID ← diamondID × 2 + 1
6:      if IsLeafNode(siblingID, leftID, rightID) then
7:          MergeNode(CBT, heapID)
8:          MergeNode(CBT, rightID)
9:      end if
10: end procedure
```

5.3 Adaptive Terrain Rendering on the GPU

In order to demonstrate the benefits of CBTs with respect to LEB, we implemented a large-scale terrain renderer; Figure 1 and Figure 12 show renderings produced by using our implementation. While LEB has been used before in the context of terrain rendering, we believe that our implementation is the first to provide efficient parallel algorithms that can run entirely on the GPU. In the following paragraphs, we provide some implementation details and some performance measurements.

Implementation Details. Our terrain renderer is written in C++ and GLSL450 shaders. The terrain consists of a triangulated square that we adaptively subdivide and displace using an 8K×8K, 16-bit displacement map, as well as some additional slope-dependant procedural noise [Kemen 2009]. For shading, we compute the exact normals of the terrain and use a diffuse material illuminated by an Bruneton and Neyret's precomputed atmosphere model [Bruneton and Neyret 2008]. We represent the triangulated square as a CBT of maximum depth $D = 27$, where the two root triangle have heap indexes 2 and 3. During rendering, we launch a CBT update pipeline as described in Section 4.2 with a subdivision kernel that performs the following operations:

- *Heap index decoding.* We retrieve the heap index of each leaf node in the CBT as described in Section 3.2 using Algorithm 1 (see also Algorithm 4, line 4).
- *Triangle decoding.* We convert each heap index into its associated LEB triangle using Algorithm 5. Note that we reserve the first bit of the heap index for a mirroring matrix to distinguish the two root triangles within the square.
- *Split/Merge Criteria.* We then decide to split or merge each triangle so as to avoid sub-pixel rasterization. In order to further reduce the number of primitives, we also avoid splitting triangles that lie outside of the view-frustum and those that cover locally-flat regions; Figures 1, 12 show the effect of our criteria. Note that we implement conforming triangle splitting and merging using Algorithms 7, 8 with a slight modification for Algorithm 6: we reserve the first bit of the heap index to initialize the neighbor vector $\mathbf{n}$ to $(\emptyset, \emptyset, 3, 2)$ and $(\emptyset, \emptyset, 2, 3)$ for respectively heap index 2 and 3.

Once the update pipeline complete, we invoke a vertex shader via an indirect command for each leaf node in the CBT. The vertex shader decodes the triangles produced by our adaptive LEB

subdivision, displaces them according to the input displacement map, and sends them down to the rasterizer for fragment processing. In the case of a triangulated square, our LEB algorithm produces an edge-subdivision factor as high as $2^{\frac{D-1}{2}}$. In order to produce even denser tessellations, we also provide an option to use uniformly-tessellated triangles for each leaf node of the CBT. Note that our approach allows for only single-subdivision-depth alterations per triangle and per iteration. This turns out to work well in practice, even for fast camera movements. Our approach allows us to render a 52-km-wide terrain at 6-meter precision without relying on uniformly tessellated triangles, and down to 0.1-meter precision otherwise.

Performances. Our terrain implementation takes less than five milliseconds to render at full-HD resolution on an NVIDIA RTX 2080 GPU. In order to provide more atomic performance evaluations, we performed the following experiments:

- *Geometric Overhead.* We measured the performance of our terrain renderer with a fragment shader that output a constant color so as to measure the overhead caused by geometric processing. In addition, we also measured the performance of our terrain renderer with displacement mapping disabled so as to measure the overhead caused by our displacement procedure. For our tests, the camera's orientation was fixed, looking downwards, so that the terrain would occupy the whole framebuffer, thus maintaining constant rasterizeration activity. Table 1 provides the results of our measurements. As demonstrated by the reported numbers, the overhead caused by our CBT update pipeline is largely dominated by the sum-reduction kernel, which takes 1.484ms. As for the rendering kernel, we evaluate that the use of a CBT adds an overhead of 0.277ms.
- *Parallel Processing Scalability.* We also conducted a standalone experiment to measure the scalability of our LEB algorithm with respect to processor count. The experiment consisted in refining a triangle around a specific point down to a maximum depth of $D = 27$. We measured the speed at which the subdivision was computed on a multicore processor by running the algorithm over a varying number of threads. We also compared our CPU-based measurements against a GPU-based one. Table 2 provides the results of our measurements. As demonstrated by the reported numbers, our implementation speeds-up linearly with the number of processors. We also emphasize that the benefits gained from GPU-acceleration are considerable, reaching speed-ups of two orders of magnitude with respect to a serial implementation.

	CBT update ($D = 27$)			render		
Kernel	dispatch	subdivision	sum-reduction	shaded	flat	w/o disp.
Timing (ms)	0.017	0.035	1.484	1.975	0.390	0.277

Table 1. GPU timings on an NVIDIA RTX 2080 averaged over 100 frames.

Processor	3960X (1)	3960X (2)	3960X (4)	3960X (8)	3960X (16)	RTX 2080
Timing (s)	12.55	6.32	3.19	1.62	0.88	0.03
Acceleration	×1	×1.99	×3.93	×7.75	×14.26	×418.33

Table 2. LEB-processing scalability using a CBT of maximum depth $D = 27$ on multiple threads of an AMD Ryzen Threadripper 3960X CPU with 24 cores, as well as an NVIDIA RTX 2080.

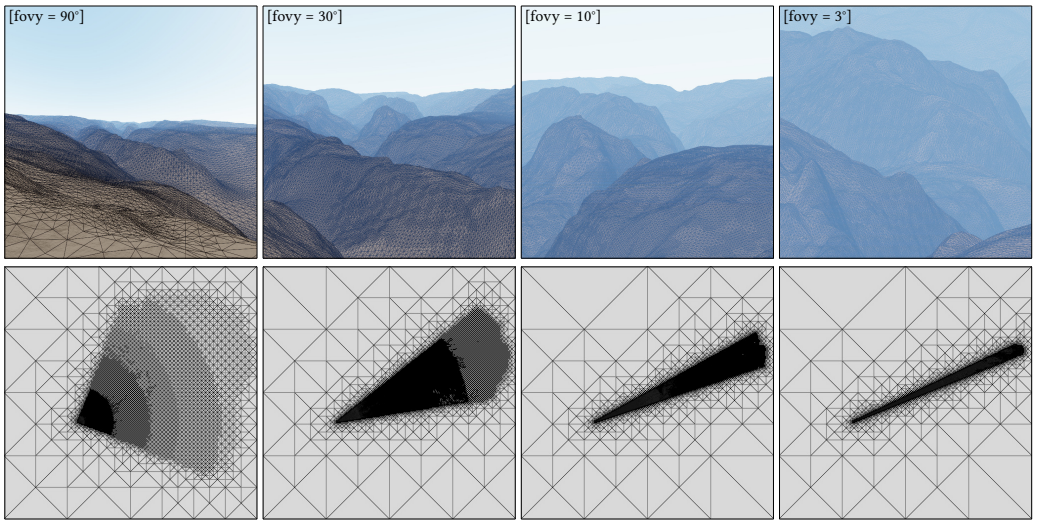


Fig. 12. (top) Frames taken from a zoom-in animation over the terrain. (bottom) Wireframe top-views.

6 DISCUSSION AND FUTURE WORK

We presented CBTs, a concurrent data-structure to compute binary trees in parallel. We see two main directions that can be taken for future work:

Applications for LEB. The first direction points towards applying our LEB algorithms to other problems that benefit from adaptive triangle tessellations. In particular, we would be interested in accelerating the subdivision surface algorithms of Velho [Velho 2000; Velho and Zorin 2001], as well as the adaptive fluid solver of Ando et al. [2013]. For the latter application, we mention that an additional algorithm is required to compute the true neighbors of any triangle produced by LEB; we provide such an algorithm in Appendix B. We would also be interested in deriving the LEB algorithms suitable for the 3D setting, for which symbolic algorithms are known [Atalay and Mount 2007; Hebert 1994].

Applications for CBTs. The second directions points towards leveraging CBTs for other techniques that exhibit tree-like recursion. Straightforward examples would include quadtrees and octrees, while more complicated ones would include, e.g., dynamic BVHs. In the case of BVHs, we mention that one would require access to each node of the binary tree (rather than just the leaf nodes), as well as associate data to each node. If additional data needs to be associated with each node of the binary tree, two approaches are possible. The first approach is to rely on an additional binary heap of maximum depth D to store the data. The data associated with each node is then located at their respective heap index. This is straightforward to implement but can be wasteful in terms of memory if the binary tree is very sparse with respect to the binary heap. The second approach is to rely on a sparse array large enough to store data for each leaf node of the binary tree. Then, given the heap index of a leaf node encoded within a CBT, we retrieve its leaf index using the inverse of Algorithm 1, which is provided in Algorithm 13. The data associated with each leaf node is then located at their respective leaf index.

ACKNOWLEDGMENTS

This paper was written in France during its lockdown due to the COVID-19 pandemic. Thanks to my fiancée for voicing my supplemental video and supporting me during the deadline when I was struggling to finish the writing due to breathing issues. I love you very much. Thanks to Cyril Crassin and Christophe Riccio for nerdy GPU discussions. Thanks to Laurent Belcour, Arthur Dufay, Eric Heitz, and Kenneth Vanhoey for proofreading early drafts. Thanks to Thomas Deliot for helping me port my code to the Unity game engine; Figure 1 looks good thanks to him.

REFERENCES

Ryoichi Ando, Nils Thürey, and Chris Wojtan. 2013. Highly Adaptive Liquid Simulations on Tetrahedral Meshes. *ACM Trans. Graph.* 32, 4, Article 103 (July 2013), 10 pages. DOI: http://dx.doi.org/10.1145/2461912.2461982

Ciprian Apetrei. 2014. Fast and Simple Agglomerative LBVH Construction. In *Computer Graphics and Visual Computing (CGVC)*, Rita Borgo and Wen Tang (Eds.). The Eurographics Association. DOI: http://dx.doi.org/10.2312/cgvc.20141206

F. Betul Atalay and David M. Mount. 2007. Pointerless Implementation of Hierarchical Simplicial Meshes and Efficient Neighbor Finding in Arbitrary Dimensions. *International Journal of Computational Geometry & Applications* 17, 06 (2007), 595–631. DOI: http://dx.doi.org/10.1142/S0218195907002495

Eric Bruneton and Fabrice Neyret. 2008. Precomputed Atmospheric Scattering. *Computer Graphics Forum* 27, 4 (2008), 1079–1086. DOI: http://dx.doi.org/10.1111/j.1467-8659.2008.01245.x

M. Duchaineau, M. Wolinsky, D. E. Sigeti, M. C. Miller, C. Aldrich, and M. B. Mineev-Weinstein. 1997. ROAMing terrain: Real-time Optimally Adapting Meshes. In *Proceedings. Visualization '97 (Cat. No. 97CB36155)*. 81–88. DOI: http://dx.doi.org/10.1109/VISUAL.1997.663860

W. Evans, D. Kirkpatrick, and G. Townsend. 2001. Right-Triangulated Irregular Networks. *Algorithmica* 30, 2 (June 2001), 264–286. DOI: http://dx.doi.org/10.1007/s00453-001-0006-x

Kirill Garanzha, Jacopo Pantaleoni, and David McAllister. 2011. Simpler and Faster HLBVH with Work Queues. In *Proceedings of the ACM SIGGRAPH Symposium on High Performance Graphics (HPG '11)*. Association for Computing Machinery, New York, NY, USA, 59–64. DOI: http://dx.doi.org/10.1145/2018323.2018333

Mark Harris and others. 2007. Optimizing parallel reduction in CUDA. *Nvidia developer technology* 2, 4 (2007), 70.

D.J. Hebert. 1994. Symbolic Local Refinement of Tetrahedral Grids. *Journal of Symbolic Computation* 17, 5 (1994), 457 – 472. DOI: http://dx.doi.org/https://doi.org/10.1006/jsco.1994.1029

Tero Karras. 2012. Maximizing Parallelism in the Construction of BVHs, Octrees, and k-d Trees. In *Eurographics/ ACM SIGGRAPH Symposium on High Performance Graphics*, Carsten Dachsbacher, Jacob Munkberg, and Jacopo Pantaleoni (Eds.). The Eurographics Association. DOI: http://dx.doi.org/10.2312/EGGH/HPG12/033-037

Brano Kemen. 2009. Procedural terrain algorithm visualization. (2009). https://outerra.blogspot.com/2009/02/procedural-terrain-algorithm.html

Peter Lindstrom, David Koller, William Ribarsky, Larry F. Hodges, Nick Faust, and Gregory A. Turner. 1996. Real-Time, Continuous Level of Detail Rendering of Height Fields *(SIGGRAPH '96)*. Association for Computing Machinery, New York, NY, USA, 109–118. DOI: http://dx.doi.org/10.1145/237170.237217

P. Lindstrom and V. Pascucci. 2002. Terrain simplification simplified: a general framework for view-dependent out-of-core visualization. *IEEE Transactions on Visualization and Computer Graphics* 8, 3 (July 2002), 239–254. DOI: http://dx.doi.org/10.1109/TVCG.2002.1021577

Joseph M. Maubach. 1995. Local Bisection Refinement for N-Simplicial Grids Generated by Reflection. *SIAM Journal on Scientific Computing* 16, 1 (1995), 210–227. DOI: http://dx.doi.org/10.1137/0916014

Can Özturan. 1996. *Worst Case Complexity of Parallel Triangular Mesh Refinement by Longest Edge Bisection.* Technical Report. Institute for Computer Applications in Science and Engineering.

M. Cecilia Rivara. 1984. Algorithms for refining triangular grids suitable for adaptive and multigrid techniques. *Internat. J. Numer. Methods Engrg.* 20, 4 (1984), 745–756. DOI: http://dx.doi.org/10.1002/nme.1620200412

Luiz Velho. 2000. Semi-Regular 4-8 Refinement and Box Spline Surfaces. In *Proceedings of the 13th Brazilian Symposium on Computer Graphics and Image Processing (SIBGRAPI '00)*. IEEE Computer Society, USA, 131–138.

Luiz Velho and Denis Zorin. 2001. 4–8 Subdivision. *Computer Aided Geometric Design* 18, 5 (2001), 397–427. DOI: http://dx.doi.org/https://doi.org/10.1016/S0167-8396(01)00039-5 Subdivision Algorithms.

K. Weiss and L. De Floriani. 2011. Simplex and Diamond Hierarchies: Models and Applications. (2011). DOI: http://dx.doi.org/10.1111/j.1467-8659.2011.01853.x

M. Adil Yalcin, Kenneth Weiss, and Leila De Floriani. 2011. GPU Algorithms for Diamond-based Multiresolution Terrain Processing. In *Eurographics Symposium on Parallel Graphics and Visualization*, Torsten Kuhlen, Renato Pajarola, and Kun Zhou (Eds.). The Eurographics Association. DOI: http://dx.doi.org/10.2312/EGPGV/EGPGV11/121-130

A SUPPLEMENTAL CBT ALGORITHMS

We provide here the exhaustive list of CBT algorithms for the sake of completeness.

Algorithm 9 Retrieve the maximum depth of a CBT

```
1: function MaxDepth(CBT: heap)
2:     return CBT[0]
3: end function
```

Algorithm 10 Retrieve the number of leaf nodes in the CBT

```
1: function NodeCount(CBT: heap)
2:     return CBT[1]
3: end function
```

Algorithm 11 Check if leaf node

```
1: function IsLeafNode(CBT: heap, heapID: int)
2:     return CBT[heapID] == 1
3: end function
```

Algorithm 12 Node splitting and merging

```
1: function BitfieldHeapID(CBT: heap, heapID: int)
2:     D ← MaxDepth(CBT)                          ▷ see Alg. 9
3:     d ← FindMSB(heapID)
4:     return heapID × 2^(D−d)
5: end function
6:
7: procedure SplitNode(CBT: heap, heapID: int)
8:     heapID ← heapID × 2 + 1                    ▷ right child
9:     heapID ← BitfieldHeapID(CBT, heapID)       ▷ bit location
10:    CBT[heapID] ← 1                            ▷ This must be done atomically
11: end procedure
12:
13: procedure MergeNode(CBT: heap, heapID: int)
14:    heapID ← Or(heapID, 1)                     ▷ right sibling
15:    heapID ← BitfieldHeapID(CBT, heapID)       ▷ bit location
16:    CBT[heapID] ← 0                            ▷ This must be done atomically
17: end procedure
```

B ADDITIONAL LEB ALGORITHMS

True Neighboring Heap Indexes. The neighboring heap indexes computed in Algorithm 6 correspond to those located at the same subdivision level of that of the input heap index. In the context of adaptive LEB however, the actual neighboring heap indexes can differ by one subdivision level. While retrieving the true heap indexes turns out to be irrelevant for implementing the conforming-split and conforming merge procedures, it can become useful for other applications. Therefore, we provide the true neighboring heap indexes routines in Algorithm 14.

Algorithm 13 Heap- to leaf-index using a binary search

```
1:  function EncodeNode(CBT: heap, heapID: int)
2:      leafID ← 0                                          ▷ initialize to 0
3:      while heapID > 1 do                                 ▷ binary search loop
4:          if IsOdd(leafID) then
5:              leafID ← leafID + CBT[Xor(heapID, 1)]
6:          end if
7:          heapID ← heapID/2
8:      end while
9:      return leafID
10: end function
```

Algorithm 14 LEB true neighboring heap index computation

```
1:  function LebTrueNeighbors(CBT: heap, heapID: int)
2:      n ← LebNeighbors(heapID)
3:      if !IsLeafNode(CBT, n.edge) then                    ▷ See Alg. 11
4:          n.edge = n.edge/2
5:      end if
6:      if !IsLeafNode(CBT, n.left) then                    ▷ See Alg. 11
7:          n.left = n.left × 2 + 1
8:      end if
9:      if !IsLeafNode(CBT, n.right) then                   ▷ See Alg. 11
10:         n.right = n.right × 2
11:     end if
12:     return n
13: end function
```