

Linear Ray Tracing

A faster screen-space ray tracing alternative that leverages Vulkan Ray Query on Qualcomm® Adreno™ GPUs.

Authors: Shraman Biswas, Alex Bourd
Qualcomm Technologies, Inc.

Problem Statement

Screen-space ray tracing (SSRT) is a rendering algorithm used to determine the closest screen-space texel by stepping along a view vector and sampling the depth buffer at regular intervals. Applications include screen-space reflections, screen-space ambient occlusion, volume rendering, and more.

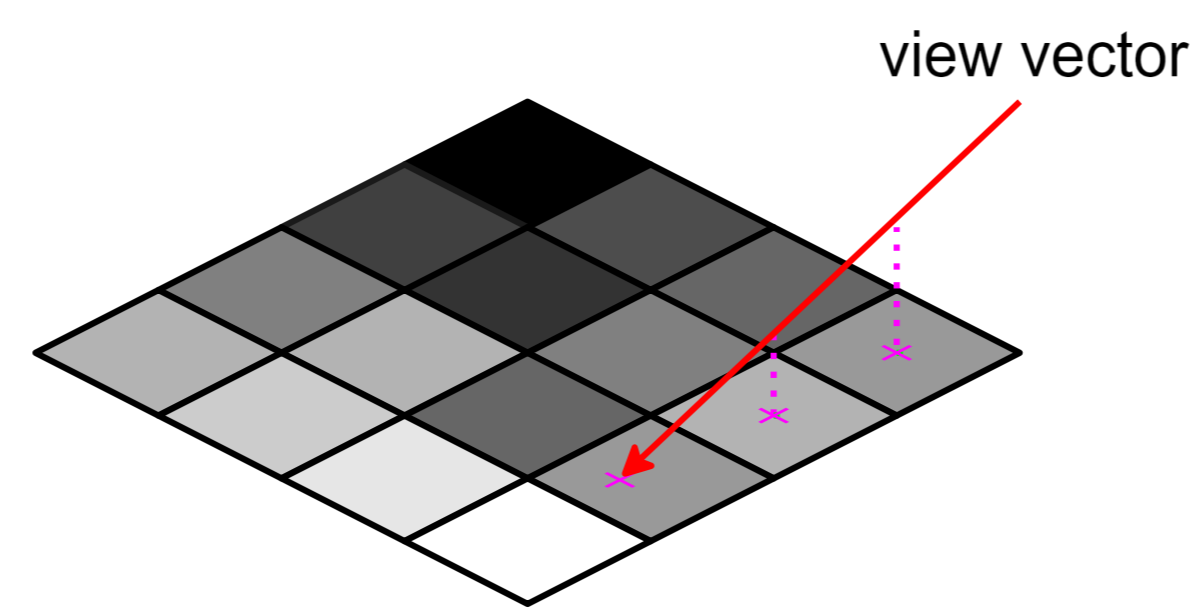
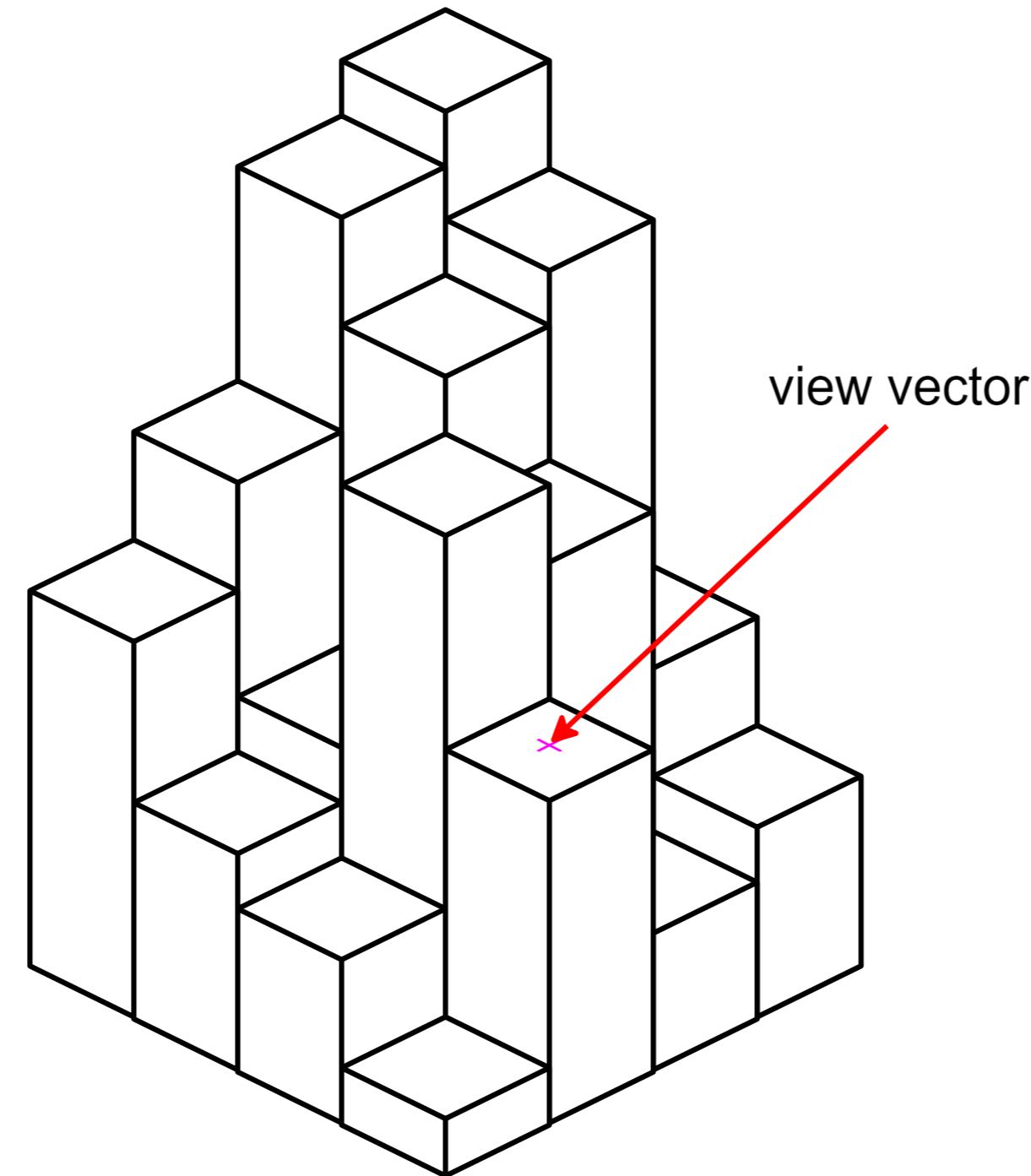
A major drawback of SSRT is that the number of ray-marching steps required varies per texel, which increases wave divergence. This reduces SIMD ALU utilization and increases texture and L1 cache miss rates.

Solution

Interpret the depth buffer as a heightmap and generate a BVH. Then, use the hardware-accelerated Vulkan Ray Query extension to determine the closest texel.

While it is possible to create a BVH with procedural nodes—using heightmap blocks as leaves—this approach tends to be suboptimal, as standard BVH construction and update algorithms are not typically optimized for this use case.

To achieve optimal utilization of ray tracing hardware, a customized BVH tree structure was developed, including tailored memory layout, construction and update algorithms, and a specialized shader for ray traversal.

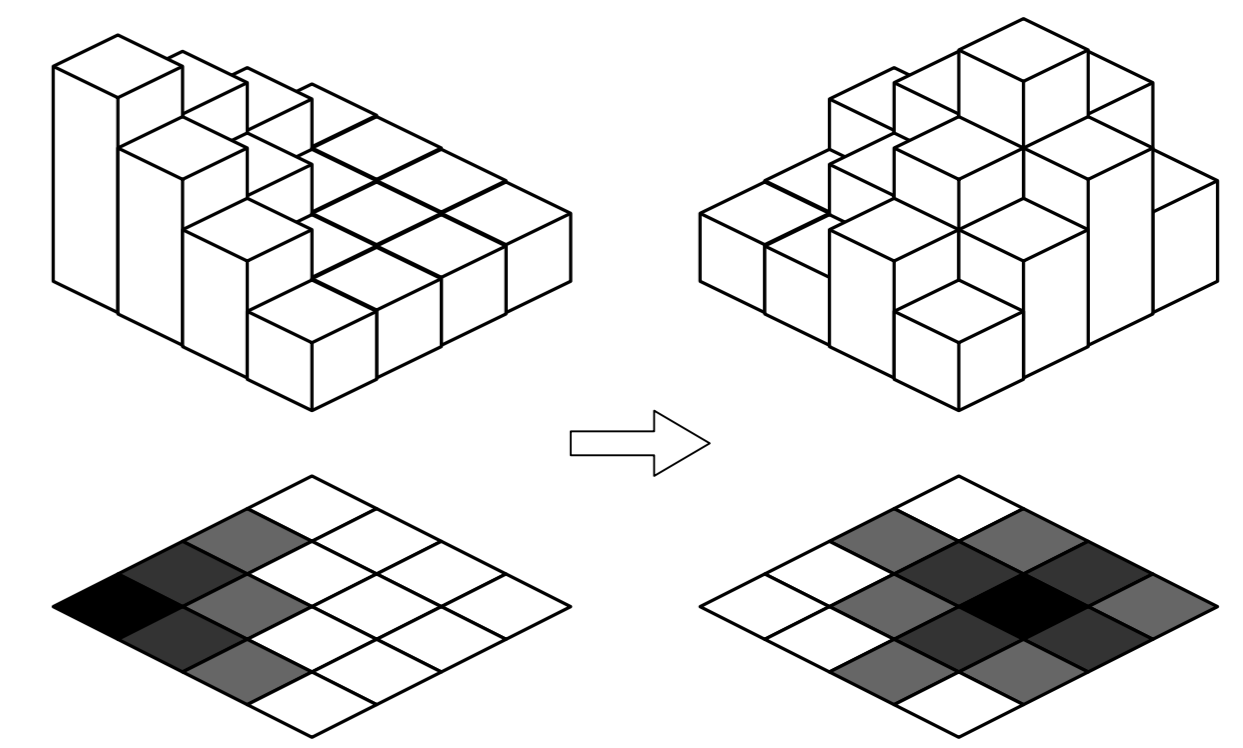


Heightmap (top) and its corresponding depth map (bottom) with screen-space ray tracing samples highlighted (magenta).

Implementation

The system is composed of three main algorithms:

- **BuildBVH:** A single compute kernel that constructs the BVH in a bottom-up manner, executed exactly once. The number of BVH levels and nodes per level is a fixed function of the input depth map dimensions.
- **UpdateBVH:** A single compute kernel that updates only the z-values at each BVH level and propagates the changes up to the root. These updates are a fixed function of changes in the depth map content.
- **RayQuery.Proceed:** A customized traversal kernel that is specialized and optimized for this specific use case—a BVH with no triangles and no instances (i.e., no transforms).



Updating the depth map (bottom) produces an updated heightmap (top) with the same texel footprint but updated texel heights

BuildBVH

Convert the depth map to a heightmap:
 $h := 1 - z$

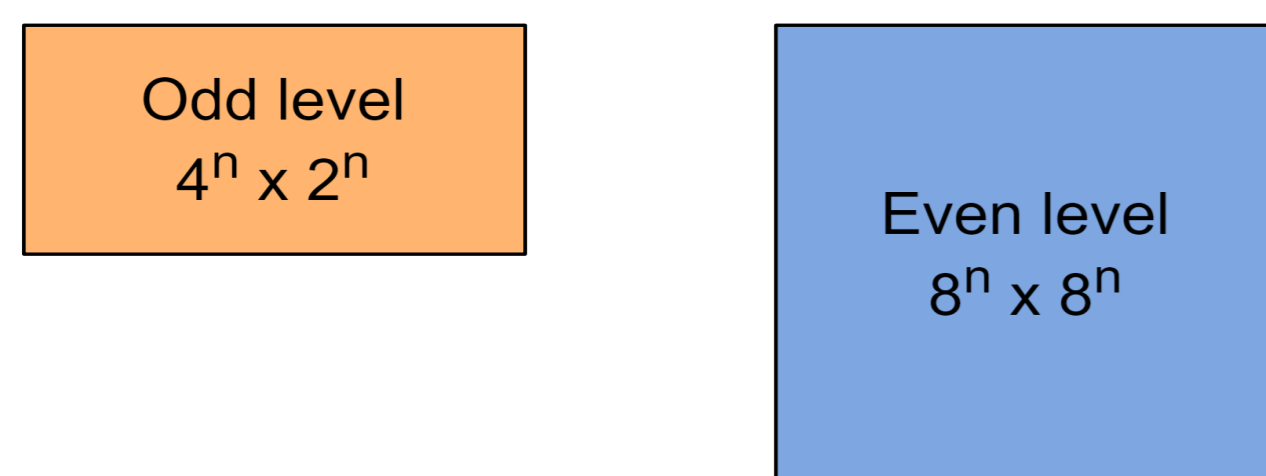
Construct a bounding volume for each texel using:

- Footprint: (dx, dy)
- Height: h

Each texel is represented as a 3D box defined by its 2D footprint and corresponding height derived from the depth value.

Q: How should children be selected per parent in an 8-wide BVH? 8x1? 4x2?

Alternate between 2D selection patterns—such as 4x2 and 2x4—to produce tighter bounding volumes. This approach yields better surface area heuristics (SAH) compared to using naïve linear patterns like 8x1 or 1x8, or fixed 2D patterns alone.

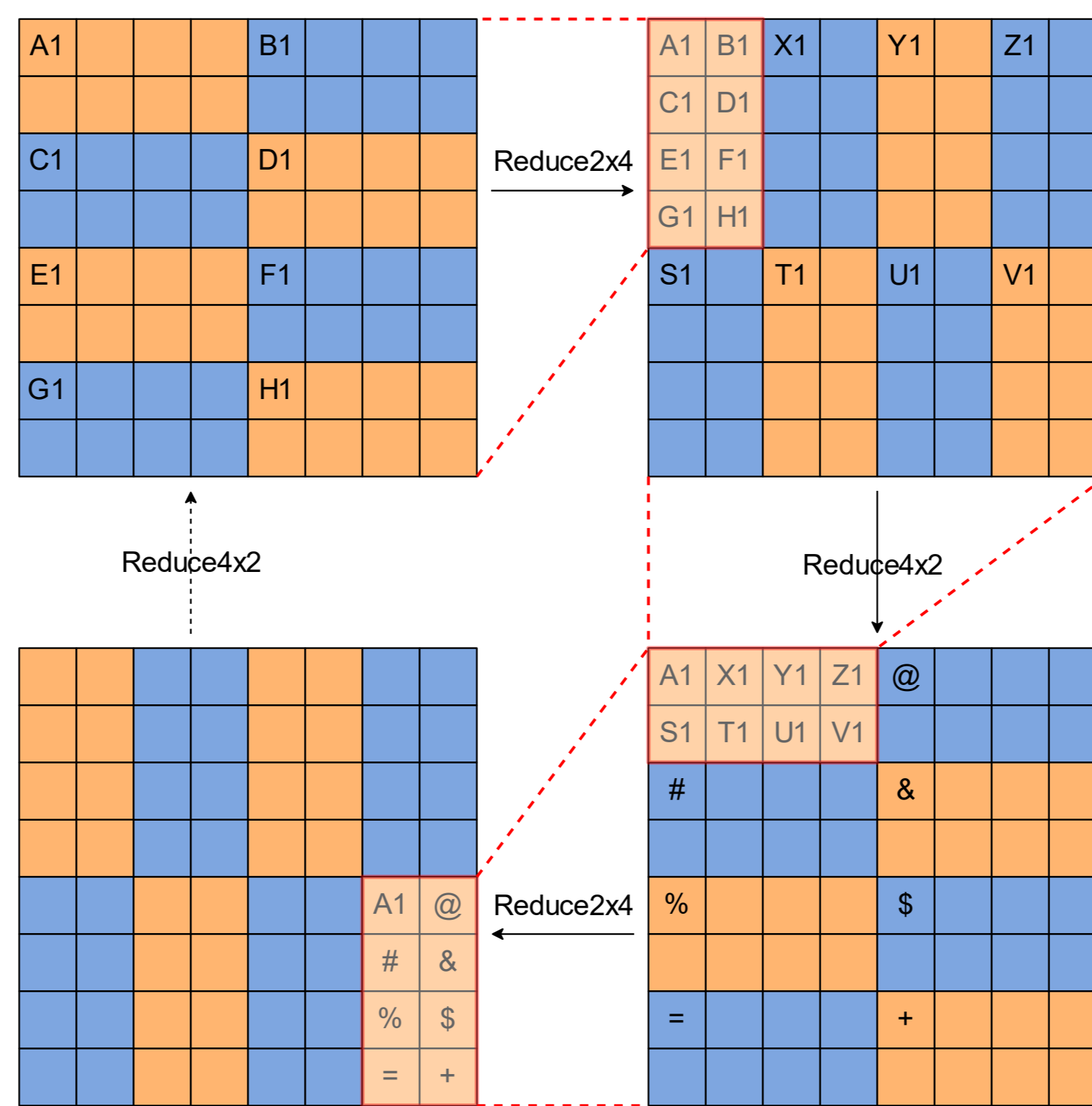


Bounding volume dimensions at odd and even levels of the BVH.

Recursively construct bounding volumes for higher levels until the root.

$$\text{dimensions}_{\text{parent}} := \max(\text{dimensions}_{\text{children}})$$

Parents also store 8-bit or 16-bit lossy compressed z-values of all its children.

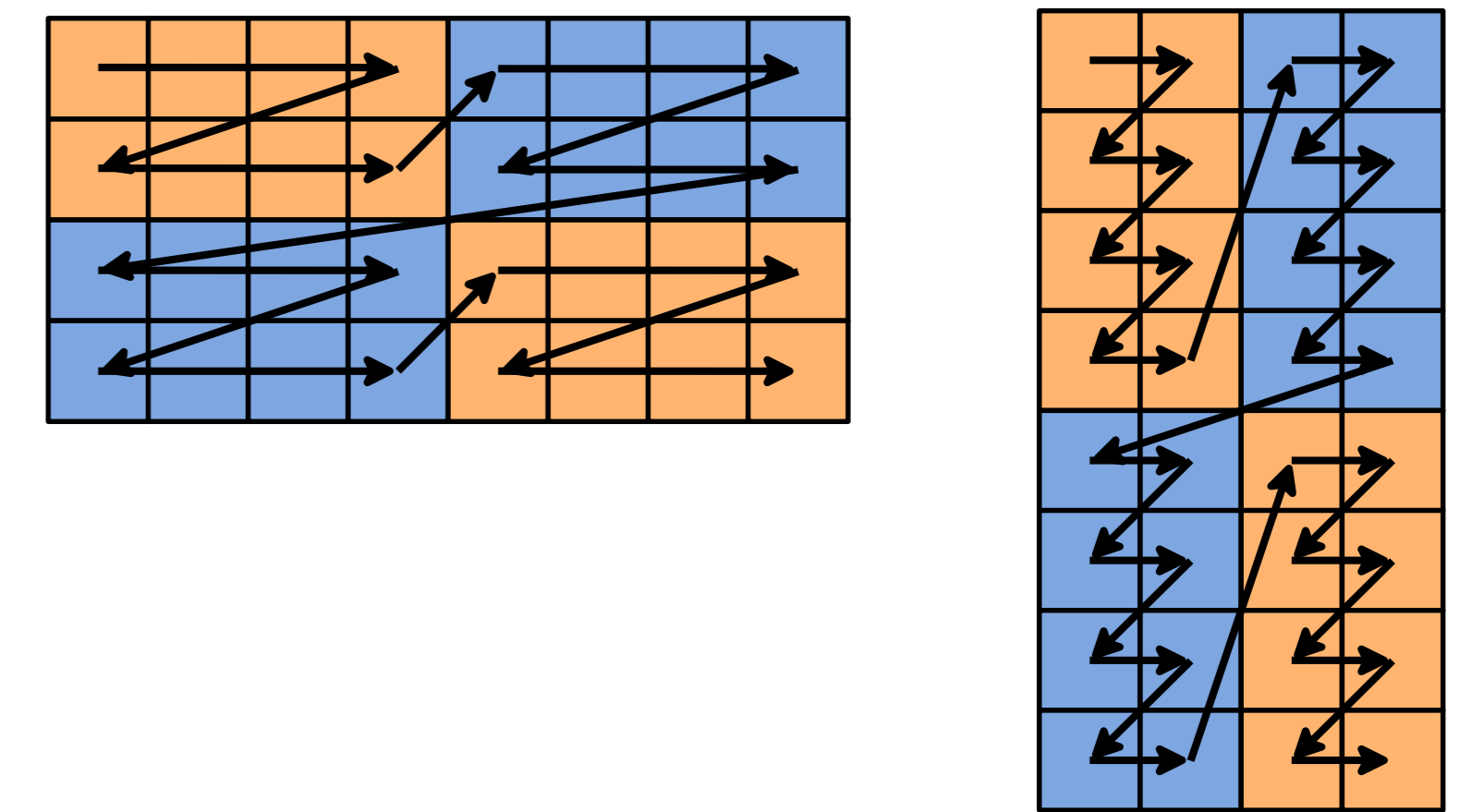


Recursive bottom-up BVH construction sequence with alternating Reduce2x4 and Reduce4x2 operations.

Q: How do you map 2D child coordinates to a 1D BVH memory layout?

Use modified Morton codes to efficiently compute the mapping from 2D coordinates to a linear memory layout.

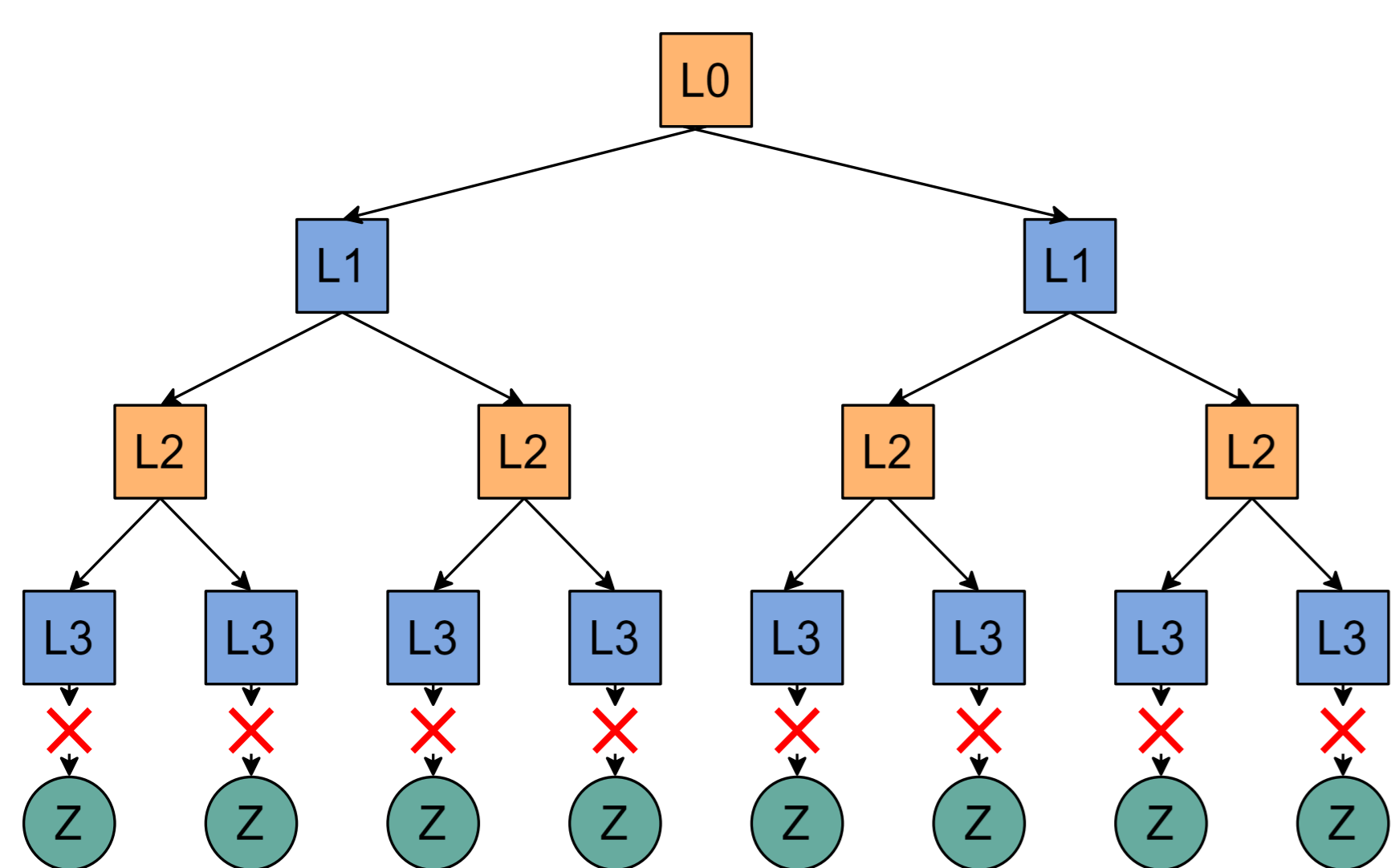
4x2 pattern: $y_n \dots y_1 | x_n \dots x_2 | y_0 x_1 x_0$
2x4 pattern: $y_n \dots y_2 | x_n \dots x_1 | y_1 y_0 x_0$



4x2 and 2x4 modified Morton ordering. The direction of the arrow indicates the order of the 2D nodes "flattened" into a 1D memory layout.

Optimization

Skip leaf nodes, as their z-values are redundant with the compressed z-values stored in their parent nodes. This results in faster traversal and reduces BVH memory usage by approximately 50%.



Example BVH with the leaf nodes (z) omitted. The penultimate level (L3) is considered the last level.

UpdateBVH

Changes in the depth map produce new heightmaps. Only the bounding volume heights (z-values) may change, while their positions (x, y) remain fixed.

The BVH is a balanced tree with a memory layout that is contiguous from the leaf nodes at the lowest level up to the root. As a result, the memory offset of any node can be trivially computed as a fixed function of the depth map dimensions. This enables fast and efficient node updates.

A simplified version of the BuildBVH algorithm is used to update only the bounding volume heights using:

$$z := \max(z_{\text{new}}, z_{\text{current}})$$

These updates are propagated upward through the BVH hierarchy to the root.

Results

Tests were performed on a device with a Snapdragon® 8 Gen 3 platform. Both synthetic and real-world depth maps were used as input with a resolution 1024x1024 and single-channel format. Both 8-bit and 16-bit depth formats were tested.

View rays were constructed originating from the depth map camera origin, with 1 ray per depth map texel. This represents the worst-case runtime with exactly 1 intersection per ray.

Runtime was measured as the UpdateBVH + RayQuery.Proceed. Achieved ~750M ray-intersections/second or ~1.4ms runtime, which was close to our design target of ~1ms.