Succinct Opacity Micromaps

GUSTAF WALDEMARSON, Dept. of Computer Science, Lund University, Sweden and Arm Ltd, Sweden

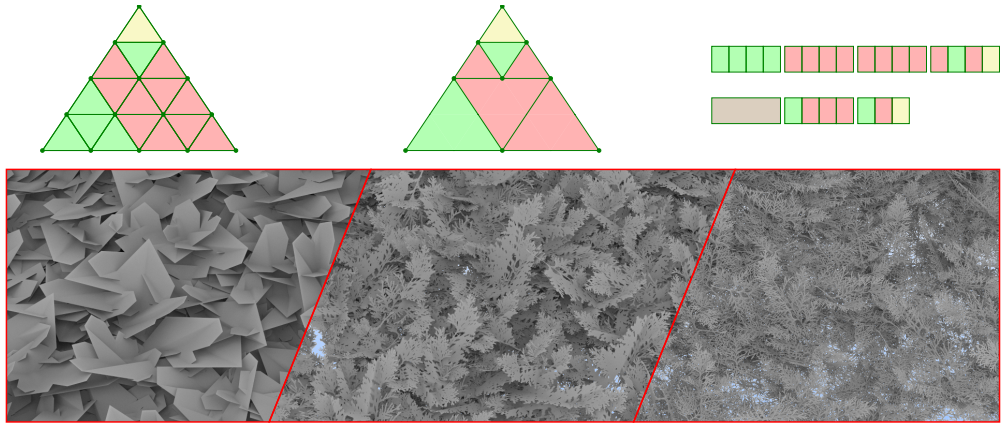MICHAEL DOGGETT, Dept. of Computer Science, Lund University, Sweden

Fig. 1. Depiction of how merging adjacent micromap values into a 4-way tree structure allows it to be compressed to around 75 % of the original size with an example rendering from the New Sponza [2022] scene showing the impact of increasing subdivision levels.

Alpha masked geometry such as foliage has long been one of the trickier things to render efficiently, both for rasterization based approaches and for hardware accelerated ray-tracing. Recently, a new type of primitive was introduced to the Vulkan® and DirectX® ray-tracing APIs that promises to alleviate this issue: Opacity Micromaps, a structure that uses a bit of extra memory as hints to the pipeline when it should *actually* call the AnyHit-shader. In this paper, we extend this primitive with a novel compression method that uses the concept of succinct 4-way trees to reduce the memory footprint by up to 110 times, including an algorithm for looking up micromap values directly from this compressed form. Further, we perform a comprehensive analysis of the generated micromaps to demonstrate their performance in terms of both memory footprint and frame render time compared to a number of similar structures. Finally, we highlight some aspects of the extension that developers and artists should be aware of to make the most out of it.

CCS Concepts: • **Computing methodologies** → **Ray tracing**; *Mesh models*; *Image compression*.

Additional Key Words and Phrases: Ray Tracing, Compression, Opacity Micromaps

ACM Reference Format:
Gustaf Waldemarson and Michael Doggett. 2024. Succinct Opacity Micromaps. *Proc. ACM Comput. Graph. Interact. Tech.* 7, 3, Article 1 (July 2024), 19 pages. https://doi.org/10.1145/3675385

Authors' addresses: Gustaf Waldemarson, gustaf.waldemarson@cs.lth.se, Dept. of Computer Science, Lund University, Lund, Sweden and Arm Ltd, Lund, Sweden; Michael Doggett, michael.doggett@cs.lth.se, Dept. of Computer Science, Lund University, Lund, Sweden.

---

1 INTRODUCTION

Transparency has always been a challenging effect to apply for computer graphics systems. Particularly for rasterization based methods that frequently have to resort to various tricks to ensure that the effect looks correct [McGuire and Mara 2017]. In contrast, the linear propagation of rays simplifies this matter for ray-tracing methods, typically removing the need for these tricks. Thus, with the advancement in hardware accelerated ray-tracing methods, the hope has been that transparency effects would become much more prevalent. This has not materialized in practice, in part due to the remaining dependence on the rasterization pipeline in many frameworks, but also due to the so called AnyHit-shader [Werness 2023] that must be invoked to correctly apply most transparency effects. As this shader is situated in one of the innermost loops of the ray-tracing pipeline it has ended up as a major bottleneck for these effects.

Recently, there has been an effort to improve this matter for one particular transparency effect: Alpha masked geometry, which is frequently used on foliage, leaves and branches in many scenes. This is done by introducing a new kind of abstraction known as a *opacity micromap* that encodes a limited amount of transparency information on a subsection of each triangle and allows this information to be quickly accessed with the ray-triangle intersection data, without having to load any other metadata. And most crucially, without having to call the AnyHit-shader during the traversal or even interrupt the hardware traversal [Sjöholm 2022], thus promising an improved ray-tracing performance for these effects.

2 RELATED WORK

The concept of micromaps were first presented by [Gruen et al. 2020]: A relatively simple format that divided a triangle into a regular grid with a simple indexing algorithm. This basic concept has remained in the current Vulkan® and DirectX® extension, but the subdivision scheme has been changed as shown in figure 2 and discussed further in Section 4.3.

Further, [Fenney and Ozkan 2023] were first to present a scheme for compressing a two-dimensional single channel opacity map to the same states as the micromap presented by Gruen et al.. This scheme compresses the maps very well (down to between 50 % to 25 % of the original size), but are fundamentally different from the final micromaps in use by Vulkan® and DirectX®. A direct comparison to our work is difficult to accomplish for two reasons: (1) Their methods are not directly available in any Graphics API, and (2) the encoding scheme uses assets with quads in a way that is not widely used in practice. Interestingly, Fenney and Ozkan also mention investigating an explicit 3-level quad-tree method as well as a wavelet mod 3 scheme. However, no details regarding this investigation was included in their work.

Coincidentally, work related to compressing displacement data into micromaps are covered by [Maggiordomo et al. 2023]. Notably, this type of micromap is primarily used to reduce the footprint of displacement data rather than to provide any frame-time improvement. However, as this work is not related to *opacity* micromaps, it will not be covered by this paper.

Fig. 2. A comparison of the subdivision schemes used by Gruen et al. (left), Vulkan® and DirectX® (right) at equivalent levels (i.e., $N = 2, 4$ and $n = 1, 2$ respectively). Note in particular the indexing order and the rounding pattern at the edges and corners, here shown with red arrows.

3 BACKGROUND

This section provides a general background to the technologies used to develop our algorithms.

3.1 Micromaps

In brief, a micromap is simply a linear array of values mapped into fixed sub-areas of a triangle specified by a space-filling curve as shown in figures 2 and 3. To date, there are only two kinds of *official* micromaps:

- Opacity Micromaps (OMM), and
- Displacement Micromaps (DMM).

Of these, only the opacity micromaps is currently available as a generally available Vulkan® and DirectX® extension [Werness 2022]. Further, an opacity micromaps can only contain a very specific set of opacity values depending on whether it is operating in the so-called 2-state or 4-state *mode*:

2-State		4-State	
		`0b00`	Fully Transparent
`0b0`	Fully Transparent	`0b01`	Fully Opaque
`0b1`	Fully Opaque	`0b10`	Unknown Transparent
		`0b11`	Unknown Opaque

As these values only occupy either 1 or 2 bits each, the opacity micromap is typically handled as a type of bit-vector. Functionally, the values are mostly self-explanatory:

- Fully transparent and opaque means that that particular sub-triangle is either completely opaque or transparent.
- Unknown values should look up the actual opacity values using some other method, by e.g., looking it up in an alpha texture. Additionally, these values can be converted to an equivalent 2-state value, e.g., when it is undesirable to perform the alpha texture lookup, such as for shadow-rays in the ray-tracing pipeline.

Thus, this extension is primarily aimed at improving the rendering performance of alpha mapped geometry, such as foliage for the DirectX® and Vulkan® ray-tracing and ray-query extensions by avoiding most, if not all, AnyHit calls or returns to the calling shader. A case where ray-tracing has long promised to excel over rasterization based methods, but failed in practice, primarily due to having to call the AnyHit-shader inside the ray-tracing traversal pipeline.

Opacity micromaps are intended to solve this: Provide a relatively small amount of extra data with hints to the ray-tracing pipeline when to avoid calling the AnyHit-shader, and consequently
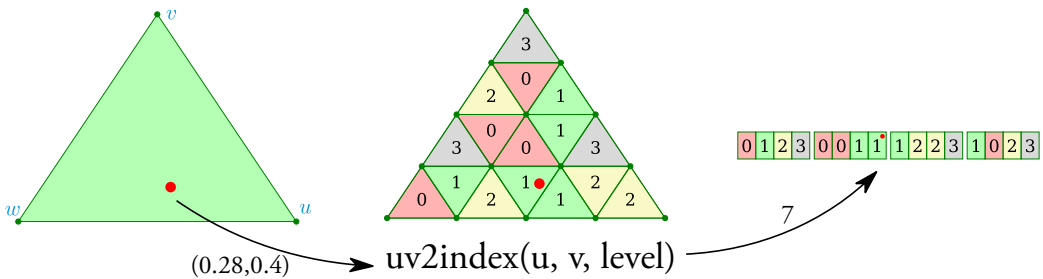

Fig. 3. Description of the barycentric coordinate to opacity micromap indexing process.

reduce the overall texture and memory bandwidth. However, these optimizations are entirely in the hands of the users: It is up to them to generate appropriate micromaps and apply them correctly to see any appreciable improvements.

3.1.1 Highest Useful Subdivision Level. In `4-state` mode, the subdivision level is typically only another tool to dial in performance, effectively trading runtime at the expense of memory. In `2-state` mode, the triangle shape and subdivision level directly influence the final geometry as there are no *unknown* values, and consequently no way of calling an `AnyHit`-shader. This shape may be distinctive, especially compared to low resolution alpha texels, as seen in figures 1 and 6. Thus, if the intent is to conservatively recreate the alpha texture with micromaps, it is useful to estimate the maximum useful subdivision level n by computing when the texture coordinate length of a subtriangle along the longest edge e is less than one pixel, i.e., when the subtriangles are smaller than the pixels. In other words:

$$\frac{\max(|e_0|, |e_1|, |e_2|)}{2^n} \geq 1 \Rightarrow \log_2 \max(|e_0|, |e_1|, |e_2|) \leq n \tag{1}$$

This is valid regardless of any minification or magnification issues introduced by camera motion as long as the texture coordinates are static. However, this becomes more complicated for primitives with coordinate transforms, but if it is possible to estimate the maximum deformation, the same approach can still be used.

3.1.2 Special Indices. In Vulkan®, Opacity micromaps are applied on triangles by providing an array of so-called triangle indices at the creation of a bottom level acceleration structure (BLAS). However, it is relatively common for micromaps to contain only a single value, which may be wasteful, especially at high subdivision levels. To alleviate this, Vulkan® provides a set of special index values to map directly to these cases:

−1	`VK_OPACITY_MICROMAP_SPECIAL_INDEX_FULLY_TRANSPARENT_EXT`
−2	`VK_OPACITY_MICROMAP_SPECIAL_INDEX_FULLY_OPAQUE_EXT`
−3	`VK_OPACITY_MICROMAP_SPECIAL_INDEX_FULLY_UNKNOWN_TRANSPARENT_EXT`
−4	`VK_OPACITY_MICROMAP_SPECIAL_INDEX_FULLY_UNKNOWN_OPAQUE_EXT`

These values can be used in some cases to reduce bandwidth or otherwise simplify the micromap processing.

3.2 Succinct Data Structures

One type of data structure that is not widely used throughout the field of computer graphics is the so-called succinct data structure, originally introduced by [Jacobson 1989]. These types of data structure are so called because of their small memory footprint: Typically only a constant factor from the information-theoretical minimum.

As a concrete example, consider all binary trees with n nodes. There are only a finite number of these trees, given by the Catalan number C_n [2023, A000108]. As such, there are approximately 4^n distinct trees for large values of n. Consequently, it is possible to enumerate them and encode the trees using only $\log_2(4^n) = 2n$ bits.

Depending on the desired properties, a number of different encoding schemes could be used: The simplest of which is to perform a depth-first search over the tree, setting a bit to 1 if the node is an internal node and 0 otherwise. Thus representing the entire tree and allowing us to readily count the number of internal and leaf nodes in the tree using population counts on the bits themselves. Further, these counts can be used as indices for retrieving data stored in the abstract tree nodes.

4 ALGORITHMS

In this section we will present the primary algorithmic contributions of this paper.

4.1 Succinct Tree Encoding

There are quite a few ways of encoding an existing micromap as a succinct tree, but one of the simplest can be expressed as follows:

- Construct the *perfect* 4-way tree from the flat micromap from the bottom up by merging adjacent nodes with identical values (algorithm 1), then
- encode the resulting *perfect* tree as a *succinct* tree (algorithm 2).

An example of this is illustrated in figure 4.

Data: Micromap map
Result: Perfect-Tree
for *level in map.level* − 1 *to* 0 **do**
    foreach *subtriangle at level* **do**
        if *all child nodes contain the same micromap value* **then**
           | Convert this node to a leaf node
        else
           | Mark the node as an internal node
        end
    end
end

Algorithm 1: Algorithm used to construct a perfect tree from a flat micromap in a bottom-up fashion.

Data: *Perfect-Tree*
Result: Bit-vectors *Succinct-Tree* and *Data*
stack ← root node of *Perfect Tree*;
while *stack is not empty* **do**
    node ← stack.pop();
    if *node is an internal node* **then**
        Succinct-Tree.append(1);
        add all child nodes to stack;
    else
        Succinct-Tree.append(0);
        Data.append(node.value);
    end
end

Algorithm 2: Algorithm used to encode a perfect tree into its succinct representation using two separate bit-vectors *Succinct-Tree* and *Data*. Note that we represent internal nodes as 1 and leaf nodes as 0.

4.2 Succinct Tree Decoding

Given an encoded tree, it is straightforward to convert this back to a non-encoded tree, and by extension, the original micromap. It is however also possible to traverse this bit-vector to directly
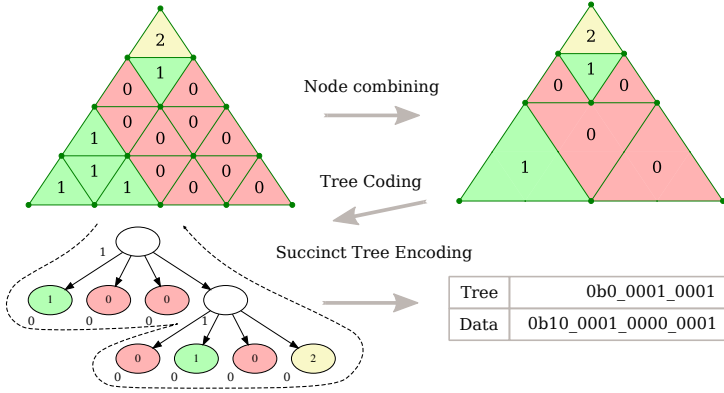
Fig. 4. A simple, but illustrative example on how to encode a flat `4-state` micromap with 2 subdivision levels as a succinct tree.

access the underlying micromap value. This can be done with algorithm 3. Note that the algorithm is strongly tied to the bit-representation of the tree. If the representation is changed, the algorithm must change accordingly.

Data: Succinct-Tree, Data
Result: Micromap value

Fn tree-lookup(u, v):
    $t = 0; d = 0$;
    while *true* **do**
        if *t is internal node in tree* **then**
            $t = t + 1$;
            $child = $ step(u, v);
            $t, d = $ bitscan($tree, data, child, t, d$);
        else
            return data[d];
        end
    end
End Fn

Fn bitscan($tree, data, child, t, d$):
    while $t < tree.length$ *and* $child > 0$ **do**
        if *tree[t] is internal* **then**
            $child = child + 4$;
        else
            $d = d + 1$;
        end
        $child = child - 1$;
        $t = t + 1$;
    end
    return t, d;
End Fn

Algorithm 3: Algorithm to directly look-up a micromap value from the tree representation. Note that the step function is described in algorithm 4 and here returns the index of the child node to visit in the range 0 to 3.

4.3 Micromap Indexing

The Vulkan® micromap extension includes an algorithm for converting barycentric coordinates to an index into the micromap structure, in this paper referred to as the *reference* algorithm [Werness 2022]. However, this algorithm is *arguably* not very easy to understand due to the opaque nature of the numerous bit-wise operations. To that end, we devised an arguably simpler, iterative algorithm (4), that to our knowledge, has not been published elsewhere yet. In brief, it does the following:

- Explicitly compute all barycentric coordinates for the hit point, then use these to determine which of the 4 sub-triangles (*L*eft, *M*iddle, *R*ight or *T*op) is hit by the intersection.

- Increment the index and recompute the barycentric coordinates according to the intersected subtriangle.
- Recurse until the desired subdivison level is reached.

Note that the majority of the conditions and their ordering is done to correspond exactly with the *reference* algorithm, as it has some peculiar rounding behavior in the edge and corner cases, as shown in figure 2. Moreover, a detailed description of this algorithm and how the expressions were derived can be found in Appendix A. Additionally, this new algorithm has been proven to be equivalent to the *reference* up to subdivision level 15 using the ACL2 theorem prover [Kaufmann and Moore 2004] using rational numbers. It is not clear whether the discrepancy at level 16 is an error in our algorithm, a failure in the *reference* due to an overflow condition or some kind of floating point issue. However, it is obvious that the *reference* algorithm **must** be rewritten to handle more than 16 subdivision levels as the final interleaving step cannot handle more than 16-bit values. Although, a micromap of that size (4 GiB) appears to be of little practical use at this time.

Data: Barycentric coordinates u, v, subdivision $level$
Result: Opacity micromap $index$
Fn uv2index($u, v, level$):
    $w = 1.0 - u - v$;
    return step($u, v, w, 0,$ **false, false**);
End Fn

Fn step($u, v, w, index, mid\text{-}flip, top\text{-}flip$):
    if $depth = level$ **then**
        return index;
    if $top\text{-}flip$ **then**
        $L, M, R, T = 2, 1, 0, 3$
    else
        $L, M, R, T = 0, 1, 2, 3$
    end
    if $w > 0.5$ **then**
        return step($2u, 2v, (w - u - v), 4 \cdot index + L, mid\text{-}flip, top\text{-}flip$)
    else if $v \geq 0.5$ *and not* ($v = 0.5$ *and* mid-flip) **then**
        return step($2u, (v - u - w), 2w, 4 \cdot index + T, mid\text{-}flip,$ **not** $top\text{-}flip$)
    else if $u \geq 0.5$ *and not* ($v = 0.5$ *and* mid-flip) **then**
        return step($(u - v - w), 2v, 2w, 4 \cdot index + R, mid\text{-}flip, mop\text{-}flip$)
    else
        return step($(u + v - w), (w + u - v), (v + w - u), 4 \cdot index + M,$ **not** $mid\text{-}flip, top\text{-}flip$)
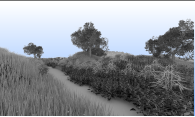   **end**
End Fn

Algorithm 4: Algorithm used to convert a pair of barycentric coordinates to a linear index into an opacity micromap. Note in particular that the *step* function is tail-recursive, and as such can easily be changed to an iterative method.

Table 1. Description over all tested scenes with a number of relevant properties. Note that micromap features may depend on the subdivision level and thus are listed as a range.
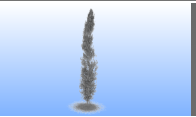
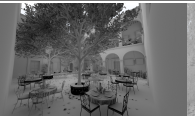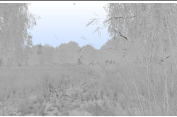| Property	Sponza	Ecosys	New Sponza	San Miguel	Landscape
Instances	25	12 755	4	380 748	407 691
Meshes	25	141	4	808	370
Triangles	262 267	1 171 562	2 023 747	2 503 044	27 885 845
Textures	37 to 38	8 to 9	4 to 5	236 to 237	211 to 212
Alpha Masks	3	8	1	236	211
Opacity Micromaps	22 to 3792	24 to 128	4 to 40	1409 to 52 118	832 to 17 636
Special Indices	218 to 28 427	1080 to 8280	0 to 1 219 113	147 313 to 394 049	287 534 to 7 498 852

| CryTek Sponza [2011] | Ecosys [1998] | New Sponza [2022] | San Miguel [2010] | Landscape [2016] |

5 RESULTS

The evaluation of the algorithms is split into two parts: One representing the compression potential of the succinct tree encoding, the other, the frame rendertime. In both cases, the algorithms are tested on the scenes described in table 1. Opacity micromaps are generated from the original alpha masks up to subdivision level 6, all of which can be found in the supplemental material. In total, we evaluate six different methods:

Micromap   Micromaps emulated in software.
Tree   Tree encoded micromaps.
Vulkan Fast-Build (FB) Vulkan Micromaps built with the *Fast-Build* flag.
Vulkan Fast-Trace (FT) Vulkan Micromaps built with the *Fast-Trace* flag.
Bitmask   A pseudo-texture where every 1 or 2 bits represent the alpha value.
Texture   The original alpha texture.

Every method except the texturing approach can also run in either `2-state` or `4-state` mode as described in Section 3.1.

5.1 Compression

We evaluate each opacity method by estimating their total memory footprint. For micromap approaches, this includes the micromap data itself, the so-called triangle indices, and any metadata structure. Only the micromap data is included for the Vulkan methods however, as they may embed any additional metadata in the acceleration structure. Further, the bitmask and texture approach need to load vertex indices and texture coordinates in addition to the bitmask or texture. This data is listed in table 2 and plotted in figure 5 for increasing subdivision levels. Finally, the tree compression ratio in figure 5 is computed against the original opacity micromap data.

5.2 Runtime

The frame rendertime is evaluated by implementing each method in an `AnyHit` shader, except for the *Vulkan* methods, as they cannot be controlled in such a fashion. All methods are rendered at 1920×1080 with a refining ambient occlusion and stochastic transparency algorithm on a Nvidia RTX 3080 and 4080. Each sample time is gathered from the start- to end-of-pipe as reported by the Vulkan pipeline querying API. Note that the values presented in table 3 and figures 6, 7 and 8 represent the average over all viewpoints in a given scene weighted by the number of rendered frames. A per-pose breakdown is available in the supplemental material, however.
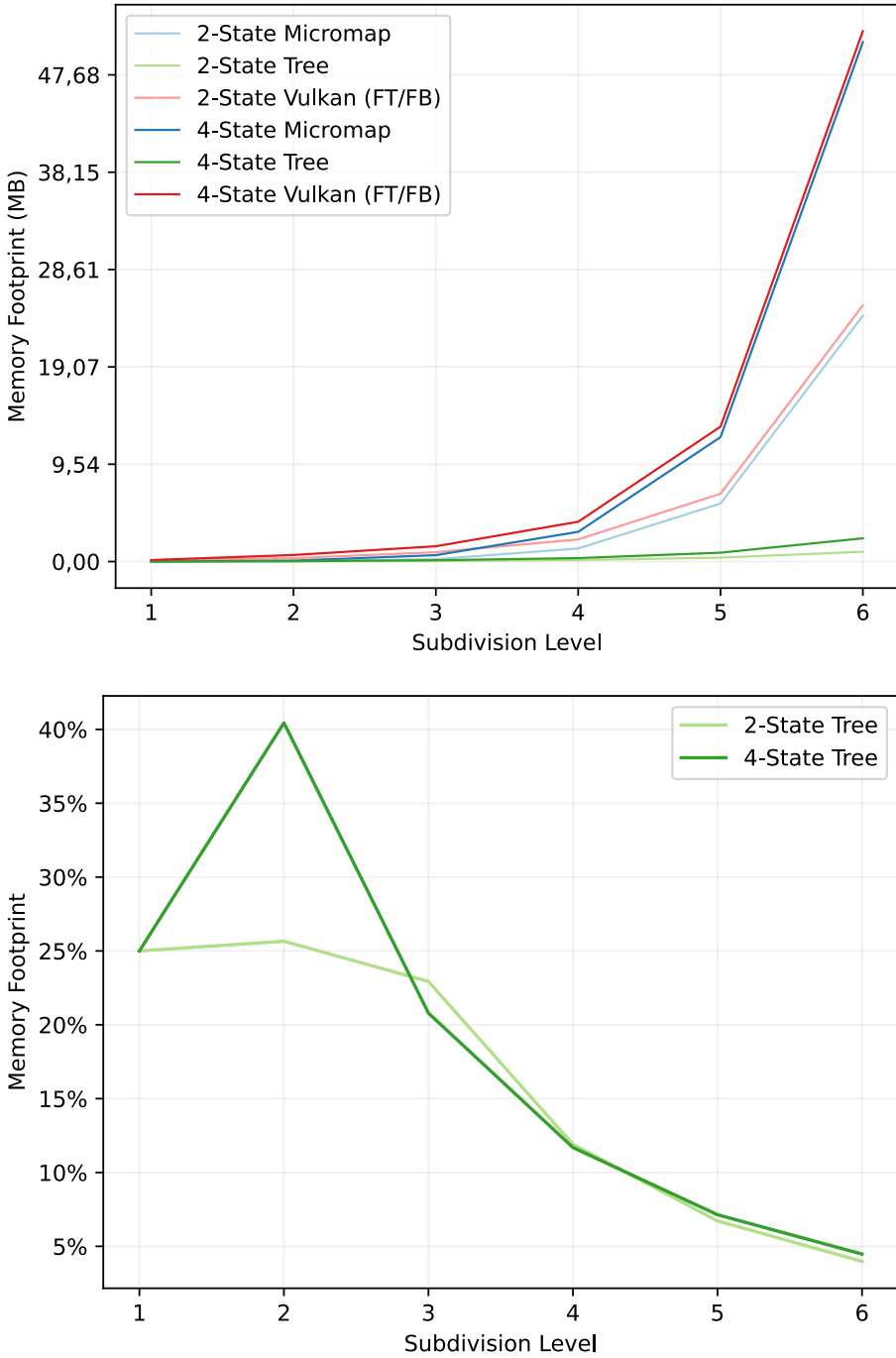
Fig. 5. Plot over the memory footprints and micromap tree compression potential for increasing subdivision levels in the San Miguel [2010] scene. The results are similar for all tested scenes and thus omitted for brevity.

Table 2. Total memory footprint in megabytes (MB) required by the various opacity methods for each scene. Note that the `fast-trace` and `fast-build` has the same footprint and that the micromap and tree method requires the same amount of metadata.

Mode	Method	Sponza	Ecosys	New Sponza	San Miguel	Landscape
2-State	Micromap	1.842	0.062	0.020	24.091	8.192
	Tree	**0.139**	**0.002**	**0.003**	**0.960**	**0.630**
	Vulkan (FT/FB)	1.915	0.067	0.021	25.103	8.574
	Bitmask	0.188	0.320	2.000	5.428	14.402
4-State	Micromap	3.703	0.125	0.039	50.896	17.223
	Tree	**0.346**	**0.004**	**0.007**	**2.279**	**1.446**
	Vulkan (FT/FB)	3.777	0.130	0.041	51.962	17.620
	Bitmask	0.375	0.640	4.000	10.852	28.801
N/A	Texture	1.500	2.558	16.000	43.399	115.195
N/A	Vertex Data	1.258	1.315	52.706	51.910	422.197
N/A	Triangle Index	0.133	0.082	5.167	1.923	35.083
N/A	Micromap/Tree Metadata	0.029	0.001	0.000	0.376	0.128

6 DISCUSSION

In this section we discuss the implications of our findings and attempt to interpret them.

6.1 Frame-time

Performance-wise, the first notable result is that there appear to be a small frametime improvement between a micromap created with the `fast-build` versus one with the `fast-trace` flag. However, the memory footprint is the same in both cases. Thus, it is likely that at this date the Nvidia driver only implement a single type of opacity micromap, but have a slightly more optimized access algorithm for the `fast-trace` case.

Further, even without official micromaps, we were able to detect a substantial improvement: Up to 16 % lower frame-time compared to using alpha masks directly. However, with only software emulation, we found a number of cases where using micromaps would instead increase the frame-time by up to 30 %. It seems as this is the primary case where the *RTX 40*-series significantly improve matters: Using the official micromap methods the worst recorded frame-time is only increased by 2 % and the best recorded one reduces it by 29 %. However, we also want to highlight the results from figure 6, which clearly shows a case where *not* using micromaps seems preferable if an *RTX 40*-series card is not available. However, we again want to note that the difference between all methods is very small: Only around 0.15 ms.

Moreover, there appears to be only an extremely small improvement to using a bitmask instead of a real texture: Presumably the bandwidth cost from loading the vertex data offsets most of the potential gains. Then again, it would be interesting to see if compressing such a map similar to the approach suggested by [Fenney and Ozkan 2023] would improve matters.

Lastly, accessing values from the tree compression is comparable to the other methods for micromaps of subdivision levels 3 or 4 as seen in figures 7 and 8. This also appears to be the more reasonable sizes for the scenes tested in this work, as can be seen among the micromap samples in the supplemental material. At higher levels however, it is arguably too slow to be of practical use, at least in its current form. This is likely caused by the `bitscan` function in algorithm 3: Each time the right-most child is accessed, all intervening subtrees for all other children must be scanned, the

number of which roughly quadruples for each subdivision level. However, improvements to these types of data structures that use a bit more memory (about 26.5 % more) to ensure that the data can still be accessed in an efficient manner already exist [Gog et al. 2014]. Investigating these options and finding a structure with a better trade-off seems like a worthwhile endeavor, and even if these structures cannot improve the access times, the tree structure would still be useful as a storage format, particularly for high subdivision levels.

6.2 Compression

The tree method presented in this work is able to compress the opacity micromap data extremely well: Typically down to between 45 and 15 % of the original size, but in some extreme cases, such as for the New Sponza scene [2022] at 12 subdivision levels, to less than 1 % the original size, or by 110 times. A trend we expect to continue at higher subdivision levels.

However, while the compression is good in terms of memory footprint, the same is not necessarily true for the memory *bandwidth*. Similar to other compression methods such as Huffman coding [Huffman 1952], a potentially large portion of memory may need to be decoded before the sought value is found, as is visualized in figure 9. This is also a notable deviation from the recommendations for texture compression given by [Beers et al. 1996] and the most important aspect that needs to be improved with a different succinct structure.

Table 3. Frametimes in milliseconds (ms) for all methods averaged over all camera poses in a scene. All micromap based methods are using subdivision level 6. See figures 7 and 8 for a per-level breakdown.

| Platform	Mode	Method	Sponza	Ecosys	New Sponza	San Miguel	Landscape
RTX 4080	2-State	Micromap	4.6	7.5	2.1	7.5	16.1
		Tree	**9.8**	**13.9**	**33.3**	**19.2**	**36.5**
		Vulkan (FT)	4.5	7.3	1.3	7.0	13.9
		Vulkan (FB)	4.6	7.4	1.4	7.0	14.2
		Bitmask	4.6	8.1	3.5	7.7	17.8
	4-State	Micromap	4.6	7.5	3.3	7.9	16.9
		Tree	**16.2**	**15.6**	**68.7**	**29.8**	**56.9**
		Vulkan (FT)	4.6	7.4	2.7	7.3	14.5
		Vulkan (FB)	4.7	7.5	2.9	7.4	15.0
		Bitmask	4.7	8.2	3.8	7.9	18.5
	N/A	Texture	4.6	8.2	3.7	8.0	18.5
RTX 3080	2-State	Micromap	7.6	13.3	3.8	15.0	30.7
		Tree	**15.8**	**23.4**	**52.3**	**34.0**	**62.3**
		Vulkan (FT)	7.6	13.6	3.6	14.8	29.9
		Vulkan (FB)	7.7	13.9	4.0	15.1	31.1
		Bitmask	7.7	14.3	7.2	15.5	34.2
	4-State	Micromap	7.7	13.4	6.6	15.6	32.0
		Tree	**25.5**	**26.3**	**108.6**	**50.4**	**94.3**
		Vulkan (FT)	7.8	13.7	6.4	15.4	31.1
		Vulkan (FB)	7.8	14.0	6.8	15.7	32.5
		Bitmask	7.7	14.1	7.0	15.7	34.0
	N/A	Texture	7.8	14.2	7.4	15.8	34.4
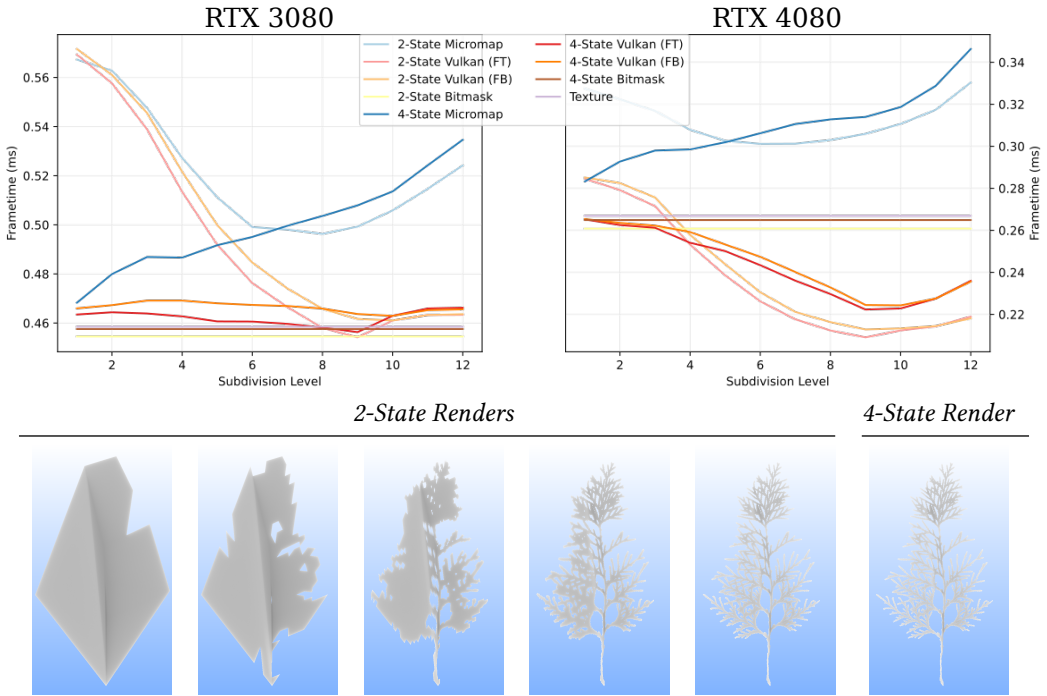
Fig. 6. Frametime plots for subdivision levels 1 to 12 when rendering a single twig from the New Sponza scene [2022]. Note that each 4-state methods yield the same render, whereas the 2-state methods is different for each one, with only subdivision 1, 3, 5, 7 and 9 shown here. Be aware of the Y-axis scale: The difference between each method is very small practice.

6.2.1 Degenerate Cases. The tree compression algorithm seems to be good in the presented scenes, but there are a number of cases that cannot be compressed with this method. E.g., a micromap that *always* alternates between the micromap states will create a succinct tree that requires more memory than the original micromap, as depicted in figure 10. Such cases could be handled by extending the encoding to sub-trees rather than just leaf-nodes but as these cases are exceedingly rare in real content this may not be necessary in practice.

6.3 Comparing Micromaps to Textures

At first glance micromaps may appear to simply be a specialized kind of image texture. However, this is arguably not true, especially not for micromaps generated from alpha mapped foliage. Those kinds of micromaps are strongly tied to both the texture and the (uv) coordinates they were generated from. As such, they are more similar to the textures originally envisioned by [Catmull 1974]. Further, a cursory analysis of our chosen scenes quickly reveal that scenes that only use a single triangle or quad to represent foliage are exceedingly rare. As an example, figure 11 depicts a texture atlas from the CryTek Sponza scene [2011] with all unique triangle texture coordinates overlaid on top of it. This is a representative view of what can happen in practice:

- In some cases, we have a well structured grid of coordinates, in others,
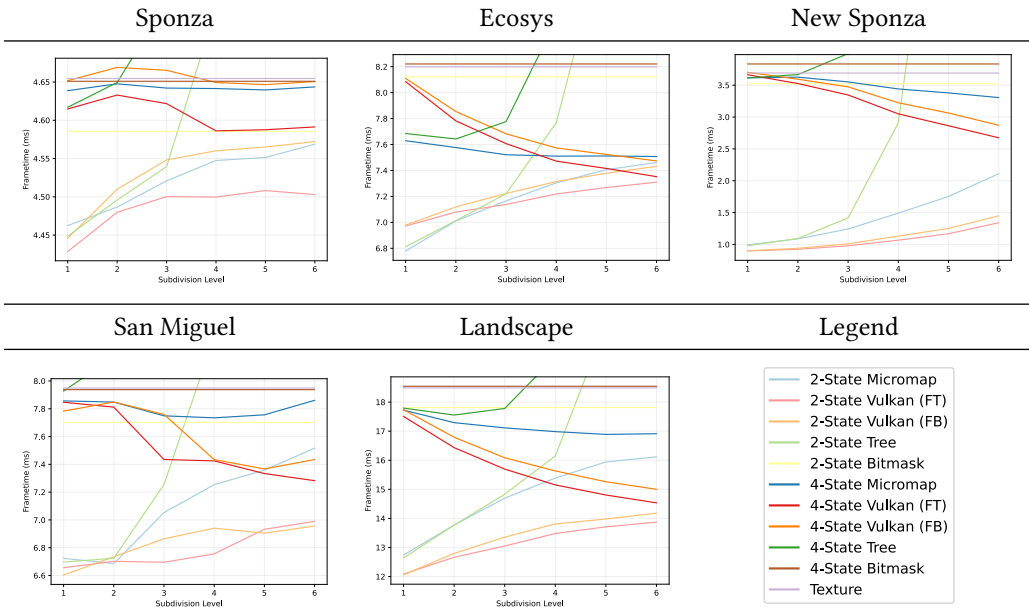- we have many overlapping and crossing edges due to the coordinates being slightly misaligned.

Fig. 7. Plots over how the average frametimes changes for increasing subdivision levels on a *RTX 4080*.

In such cases, each triangle may create unique micromaps despite sharing a common texture and may consequently increase the acceleration structure bandwidth usage rather than reduce it.
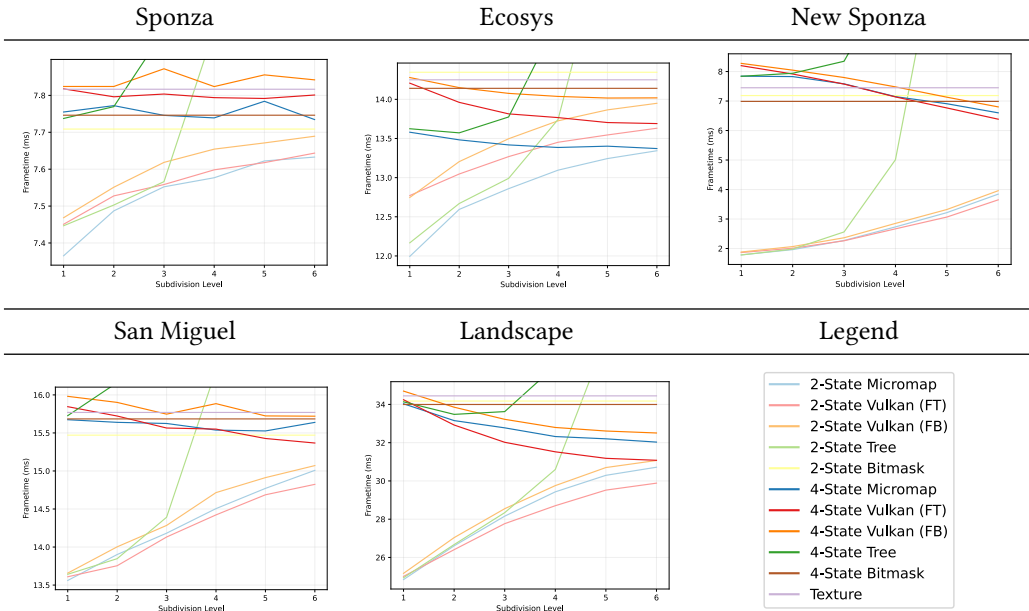

Fig. 8. Plots over how the average frame-times changes for increasing subdivision levels on a *RTX 3080*.
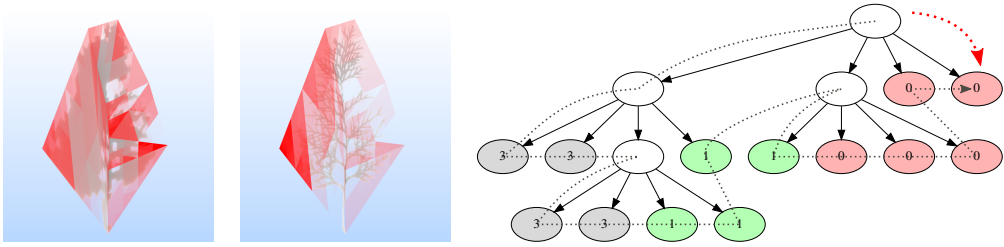
Fig. 9. Visualization of the number of bits of the micromap tree that has to be read by an incoming ray before an opacity value is found. Here shown in two scenes with 2-state micromaps at levels 4 and 9 along with a schematic view over how these asymmetric read patterns arise when attempting to read the right-most tree nodes.
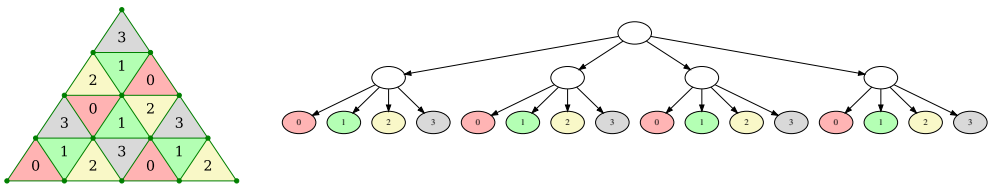

Fig. 10. Example degenerate case for the succinct tree compression: Instead of actually reducing the number of nodes, it is forced to add 5 internal ones.

Further, in more modern scenes such as in San Miguel [2010] and Landscape [2016], leaves and branches are often modeled individually and split into many segments to allow artists to give them more depth as seen in figure 12. In these cases, micromaps work well, and relatively small subdivision levels can be used as only small parts of the mesh cover partially-transparent regions. However, in theses cases it is important to use the special triangle indices (see Section 3.1.2) to avoid having to explicitly represent all triangles, most of which will be fully-opaque.

7 FUTURE WORK

The micromap extensions are currently limited in scope but thanks to the extensible nature of the modern graphics APIs, it will be straight-forward to extend them in the future. The concept itself may even extend beyond these APIs:

The indexing algorithm (4) generalizes in multiple aspects: To other dimensions, shapes and subdivision levels. Investigating this further could lead to a number of interesting applications.

The tree accessing algorithm (3) only considered software implementations. As such, investigating the costs and benefits of these algorithms when implemented in hardware remains open, and while the presented version probably does not translate well into hardware, other uses of succinct data structures may be more amenable to this.

In regard to the extension itself: The current 2-state mode is limited to the fully-opaque or fully-transparent values. Adding one or more 1-bit modes to the opacity micromap extension that replace either of these values with the unknown-opaque or unknown-transparent could be an interesting way to further reduce the memory footprint. We plan to investigate the potential gains from such modes in the future.
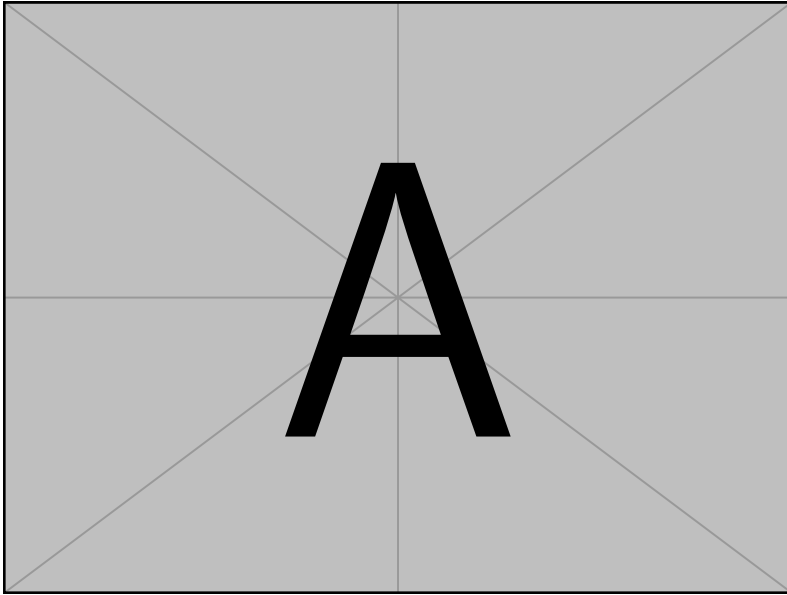
Fig. 11. A representative texture atlas of various plants from the CryTek Sponza [2011] scene with all uniquely unwrapped texture coordinates overlaid on top, along with a histogram showing how many triangles overlap each pixel.
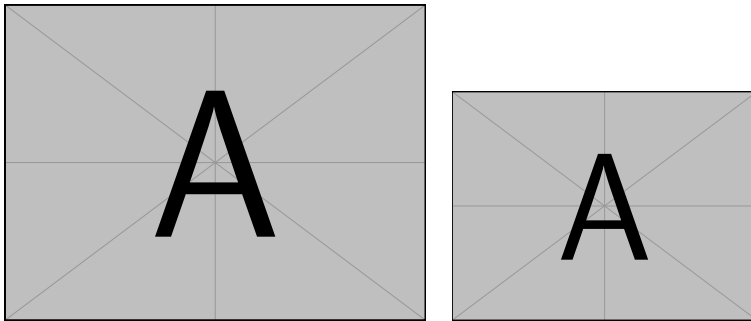

Fig. 12. Two representative textures depicting how modern scenes model leaves with a single texture and a highly tessellated grid to allow artists to give the leaves more depth.

Opacity micromaps is not the only available type: As mentioned in Section 3, *displacement* micromaps are already available on some hardware and other types are under development [Bickford 2023]. Some aspects of the presented tree encoding seem to extend to these types but more research is needed in this regard. Further, the algorithms presented in this paper only considered *lossless* encoding. Allowing lossy encoding of micromap values would open up many new avenues of research.

For the sake of brevity, the subject of *constructing* micromaps were not considered in this paper. Nvidia has published a framework for constructing them either during rendering or as a part of the asset preparation pipeline [GameWorks 2022]. Further, there is ongoing work to make similar tools

available in Blender [Waldemarson 2023]. However, the aspect of how to create micromaps in an efficient and continuous manner is arguably still worth investigating.

Finally, the use-case for micromaps, and particularly *opacity* micromaps, is rather limited at the time of writing. Finding new and clever use-cases for them could be very interesting.

8 CONCLUSION

In this paper we introduced several novel algorithms related to the compression of micromaps by converting them to succinct 4-way trees, reducing their memory footprint by up to 110 times in a number of representative scenes that use alpha mapped foliage extensively while remaining competitive at reasonable subdivision levels, but additional work is needed for this method to be practical in a high performance context.

Further, we have evaluated the potential performance gain from using the official Vulkan® and DirectX® opacity micromaps extension applied to alpha mapped geometry, confirming that the feature can provide an increase in frametime by at most 29 % when used in the so-called `4-state` mode. Further, we have documented several important considerations regarding the use of `2-state` micromaps that may be important for designers to be aware of. Moreover, we have introduced an alternative to the reference barycentric-to-index-algorithm that may be easier to understand.

REFERENCES

Andrew C. Beers, Maneesh Agrawala, and Navin Chaddha. 1996. Rendering from compressed textures. In *Proceedings of the 23rd Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH '96).* Association for Computing Machinery, New York, NY, USA, 373–378. https://doi.org/10.1145/237170.237276

Neil Bickford. 2023. *NVIDIA Micromap Extensions.* Nvidia Inc. https://github.com/NBickford-NV/glTF/tree/micro-mesh

Edwin Earl Catmull. 1974. *A subdivision algorithm for computer display of curved surfaces.* Ph. D. Dissertation. The University of Utah. AAI7504786.

Oliver Deussen, Pat Hanrahan, Bernd Lintermann, Radomír Měch, Matt Pharr, and Przemyslaw Prusinkiewicz. 1998. Realistic modeling and rendering of plant ecosystems. In *Proceedings of the 25th Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH '98).* Association for Computing Machinery, New York, NY, USA, 275–286. https://doi.org/10.1145/280814.280898

Simon Fenney and Alper Ozkan. 2023. Compressed Opacity Maps for Ray Tracing. In *High-Performance Graphics - Symposium Papers*, Jacco Bikker and Christiaan Gribble (Eds.). The Eurographics Association, EUROGRAPHICS Association Groene Loper 3, 5612AE Eindhoven, 23–31. https://doi.org/10.2312/hpg.20231133

Morgan McGuire Frank Meinl, Marko Dabrovic. 2011. CryTek Sponza. https://www.cryengine.com/asset-db/product/cry-tek/sponza-sample-scene.

Nvidia GameWorks. 2022. *Opacity Micromap SDK.* Nvidia Inc. https://github.com/NVIDIAGameWorks/Opacity-MicroMap-SDK

Simon Gog, Timo Beller, Alistair Moffat, and Matthias Petri. 2014. From Theory to Practice: Plug and Play with Succinct Data Structures. In *Experimental Algorithms*, Joachim Gudmundsson and Jyrki Katajainen (Eds.). Springer International Publishing, Cham, 326–337.

Holger Gruen, Carsten Benthin, and Sven Woop. 2020. Sub-Triangle Opacity Masks for Faster Ray Tracing of Transparent Objects. *Proc. ACM Comput. Graph. Interact. Tech.* 3, 2, Article 18 (Aug. 2020), 12 pages. https://doi.org/10.1145/3406180

David A. Huffman. 1952. A Method for the Construction of Minimum-Redundancy Codes. *Proceedings of the IRE* 40, 9 (Sep. 1952), 1098–1101. https://doi.org/10.1109/JRPROC.1952.273898

G. Jacobson. 1989. Space-efficient static trees and graphs. In *Proceedings of the 30th Annual Symposium on Foundations of Computer Science (SFCS '89)*. IEEE Computer Society, USA, 549–554. https://doi.org/10.1109/SFCS.1989.63533

Timm Dapper Jan-Walter Schliep, Burak Kahraman. 2016. Landscape. https://www.laubwerk.com.

M. Kaufmann and J S. Moore. 2004. The ACL2 Home Page. In *http://www.cs.utexas.edu/users/moore/acl2/*. Dept. of Computer Sciences, University of Texas at Austin, Main Building (MAI) 110 Inner Campus Drive Austin, TX 78712, 1.

Guillermo M. Leal Llaguno. 2010. San Miguel. https://www.pbrt.com.

Andrea Maggiordomo, Henry Moreton, and Marco Tarini. 2023. Micro-Mesh Construction. *ACM Trans. Graph.* 42, 4, Article 121 (July 2023), 18 pages. https://doi.org/10.1145/3592440

Morgan McGuire and Michael Mara. 2017. Phenomenological Transparency. *IEEE Transactions of Visualization and Computer Graphics* 23, 5 (May 2017), 1465–1478. https://casual-effects.com/research/McGuire2017Transparency/index.html#bibtex IEEE Transactions of Visualization and Computer Graphics.

Frank Meinl, Katica Putica, Cristiano Siqueria, Timothy Heath, Justin Prazen, Sebastian Herholz, Bruce Cherniak, and Anton Kaplanyan. 2022. Intel Sample Library. https://www.intel.com/content/www/us/en/developer/topic-technology/graphics-processing-research/samples.html.

OEIS Foundation Inc. 2023. The On-Line Encyclopedia of Integer Sequences. Published electronically at http://oeis.org.

Juha Sjöholm. 2022. *Best Practices for Using NVIDIA RTX Ray Tracing (Updated)*. Nvidia. https://developer.nvidia.com/blog/best-practices-for-using-nvidia-rtx-ray-tracing-updated/

Gustaf Waldemarson. 2023. *Handling Custom Data in glTF Files with Exporter/Importer Plugins*. The Blender Foundation, Youtube. https://youtu.be/4fBGM8qc21M?t=1783

Eric Werness. 2022. *VK_EXT_opacity_micromap*. The Khronos Group Inc. https://registry.khronos.org/vulkan/specs/1.3-extensions/man/html/VK_EXT_opacity_micromap.html

Eric Werness. 2023. *Any-Hit Shaders*. The Khronos Group Inc. https://registry.khronos.org/vulkan/specs/1.3-extensions/html/vkspec.html#shaders-any-hit

A BARYCENTRIC TO MICROMAP INDEX ALGORITHM DETAILS

The algorithm described in Section 4.3 works as follows:

(1) Split the base triangle into 4 equal sized sub-triangles using the midpoint of each edge.
(2) Then, using the vertex to edge mid-point relations;

$$\begin{cases} v_0 = m_{01} + m_{02} - m_{12} \\ v_1 = m_{01} + m_{12} - m_{02} \\ v_2 = m_{02} + m_{12} - m_{01} \end{cases}$$

and the typical formula for barycentric coordinates ($P = wv_0 + uv_1 + vv_2$), it is possible to derive the following relations to update the u, v, w coordinates for each of the sub-triangles L, M, R, T:

$$L := \begin{cases} u_L & = u - v - w \\ v_L & = 2v \\ w_L & = 2w \end{cases} \qquad M := \begin{cases} u_M & = u + v - w \\ v_M & = v + w - u \\ w_M & = u + w - v \end{cases}$$

$$R := \begin{cases} u_R & = 2u \\ v_R & = v - u - w \\ w_R & = 2w \end{cases} \qquad T := \begin{cases} u_T & = 2u \\ v_T & = 2v \\ w_T & = w - u - v \end{cases}$$

(3) Note that the indexing and ordering of the updated coordinates are significant to handle rounding and winding changes for the top and middle triangles as the reference algorithm rounds *globally* away from the w coordinate. Thus, the rounding direction has to be maintained as we recurse and change winding, requiring special handling for the top and middle triangles as follows:
   - Each time we recurse into the top triangle, flip the local index of the left and right subtriangles.

- Each time we recurse into the middle triangle, round u and v tie-breaks to the middle rather than the right subtriangle.

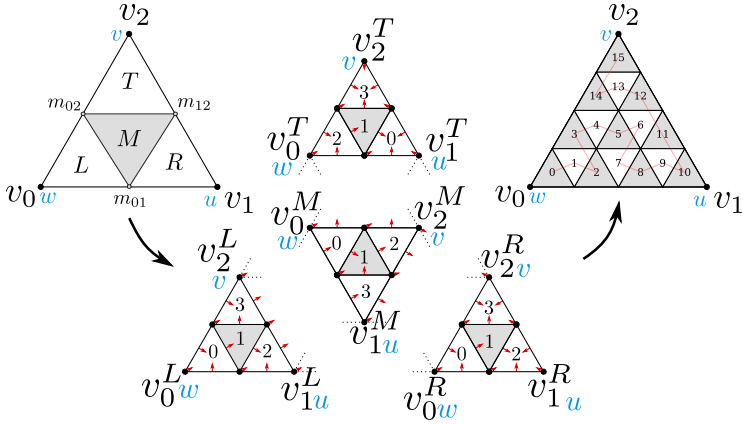See figures 13 and 2 for examples on how this works in practice.

Fig. 13. Description of how a triangle that has already been subdivided once into subtriangles L, M, R, T is further subdivided. Note in particular how the uv coordinates, indices, and rounding change for each of the subtriangles.