

# Succinct Opacity Micromaps

Gustaf Waldemarson & Michael Doggett  
Arm Sweden & Lund University

arm



2025-01-11

Ah, I think we are just about ready to start!

Thanks everyone for coming! I hope you've all got plenty of food during the buffet, but *hopefully* not enough to go into a food-coma!

So, allow me to start this session and introduce, *or I hope*, re-introduce you to the concept of *opacity micromaps*.



# Agenda

---

1. What are Opacity Micromaps?
  - `uv2index` (`BarycentricsToSpaceFillingCurveIndex`)
2. *Succinct* Opacity Micromaps
  - Memory Footprint Comparison
3. Frametime Performance Evaluation

## Agenda

So, to start off: Let us have a quick look at what I will present: In short, I have three main topics I will go through:

- A quick overview about micromaps and (opacity) micromaps in particular, including a new micromap indexing algorithm.
- Then, we will proceed to discuss our main contribution in this paper: A new method for compressing micromap data by converting it to a densely packed tree format, but more on that in a bit.
- Then we will round things off with a frametime performance evaluation of the above format as well as many other methods used to perform opacity testing, including the official micromap formats that are now ostensibly accelerated in hardware on modern Nvidia GPUs.

# Opacity Micromaps

---



2025-01-11

Opacity Micromaps

Opacity Micromaps

---



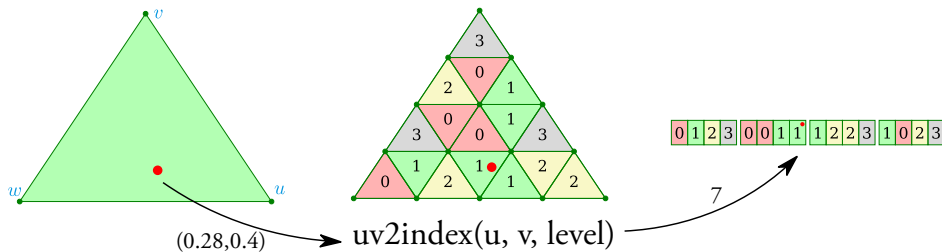
# Opacity Micromaps

2-State

0b0	Fully Transparent
0b1	Fully Opaque

4-State

0b00	Fully Transparent
0b01	Fully Opaque
0b10	Unknown Transparent
0b11	Unknown Opaque



(0.28,0.4)

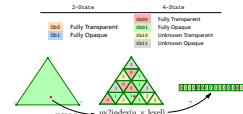
$uv2index(u, v, level)$

7

## Opacity Micromaps

## Opacity Micromaps

### Opacity Micromaps



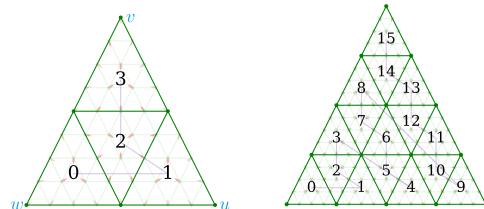
But without further ado, let us jump straight into it and talk about *Micromaps*.

What *exactly* is it? In short, it is simply an array of values mapped into fixed sub-areas of a triangle as specified by a space-filling curve. In theory, we could assign *any* kind of data to each of these sub-triangle faces, but for opacity micromaps we only care about 2 or 4 values. Consequently, these values only occupy either 1 or 2 bits each, hence, the opacity micromap is typically handled as a type of bit-vector and semantically, the values are mostly self-explanatory:

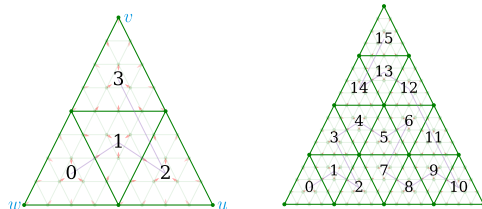
- Fully transparent and opaque means that that particular sub-triangle is either completely opaque or transparent.
- Unknown values should look up the actual opacity values using some other method, by e.g., calling the AnyHit-shader and looking it up in an alpha texture. Additionally, these values can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



# Micromap Evolution



Gruen et al.



Vulkan® & DirectX® [Werness 2022]



Starting from the beginning though, micromaps were originally invented by [Gruen et al.](#) in order to improve the performance when using AnyHit ray-tracing shaders. Simply put: by using a little bit of extra memory as input to the ray-tracing pipeline we could improve the performance by reducing the number of AnyHit call by about 30 % to 40 %.

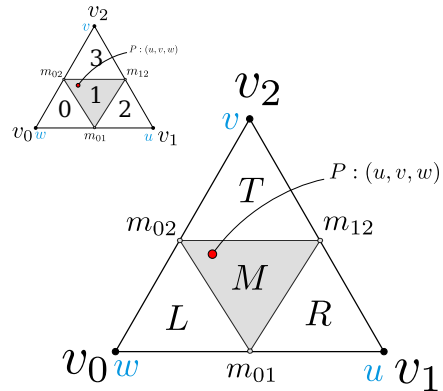
This concept has more or less remained in the current form of opacity micromaps found in Vulkan® and DirectX®: The main difference is simply *how* the subtriangles are ordered in the final bit-stream, which you can see *here*: [Gruen et al.](#) ordered the triangles in strips from the *w*-coordinate going up *like this*. Whereas the current format walk across the subtriangles in a more even fashion, similar to how a Z-curve orders tiles.

One notable consequence of this is that [Gruen et al.](#)'s format is technically more granular as each level only adds a single strip, but similar to data stored row or column order, it could lead to somewhat unbalanced caching behaviors depending on the access pattern.

# Opacity Micromaps Construction

uv2index

BarycentricsToSpaceFillingCurveIndex



2025-01-11

Opacity Micromaps

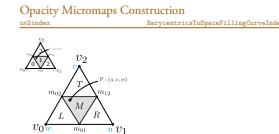
Opacity Micromaps Construction

This naturally begs the question: How do you actually look-up a value in the micromap? Which, as if by coincidence, leads us to our first novel contribution: A *new* simpler algorithm to explain the barycentrics-to-index mapping.

1. Starting with a triangle, we divide each edge at the midpoint, thus forming 4 new triangles, which I have taken to calling the *Left*, *Middle*, *Right* and *Top* subtriangle. These are also indexed as seen earlier or up [here](#).

(Next Slide)

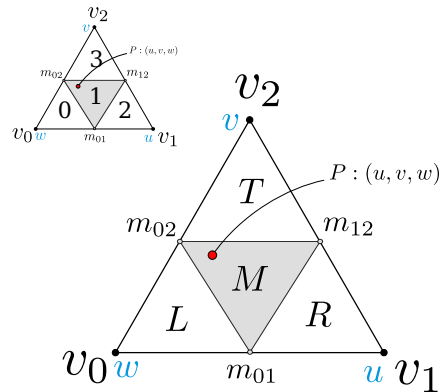
2. Then, we can use a bit of geometry to derive these equations to link the midpoints to the vertices.



# Opacity Micromaps Construction

uv2index

BarycentricsToSpaceFillingCurveIndex

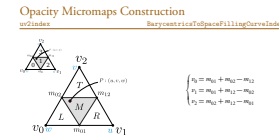


$$\begin{cases} v_0 = m_{01} + m_{02} - m_{12} \\ v_1 = m_{01} + m_{12} - m_{02} \\ v_2 = m_{02} + m_{12} - m_{01} \end{cases}$$

2025-01-11

Opacity Micromaps

Opacity Micromaps Construction



This naturally begs the question: How do you actually look-up a value in the micromap? Which, as if by coincidence, leads us to our first novel contribution: A *new* simpler algorithm to explain the barycentrics-to-index mapping.

1. Starting with a triangle, we divide each edge at the midpoint, thus forming 4 new triangles, which I have taken to calling the *Left*, *Middle*, *Right* and *Top* subtriangle. These are also indexed as seen earlier or up [here](#).

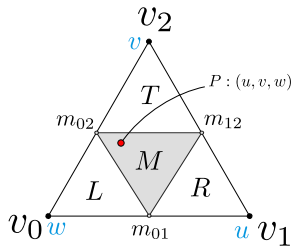
**(Next Slide)**

2. Then, we can use a bit of geometry to derive these equations to link the midpoints to the vertices.

# Opacity Micromaps Construction

uv2index

BarycentricsToSpaceFillingCurveIndex



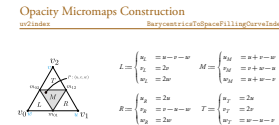
$$L := \begin{cases} u_L &= u - v - w \\ v_L &= 2v \\ w_L &= 2w \end{cases} \quad M := \begin{cases} u_M &= u + v - w \\ v_M &= v + w - u \\ w_M &= u + w - v \end{cases}$$

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

2025-01-11

Opacity Micromaps

Opacity Micromaps Construction



- Furthermore, we can express each vertex and midpoint with barycentric coordinates: And with a bit of substitution, we can find expressions that let us use the original coordinates to *update* the barycentrics in terms of the subtriangles.
- This can then be repeated recursively to any desired subdivision level. (Although, note that the coordinates should be rotated for the *Top* and *Middle* subtriangles.)

There are a little more to this algorithm though, but those details only applies to the *literal* edge cases, to ensure that rounding is correct, so please see the paper for that.

(For the interested reader: The algorithms for this is included among the extra slides.)

# Succinct Opacity Micromaps

---



2025-01-11

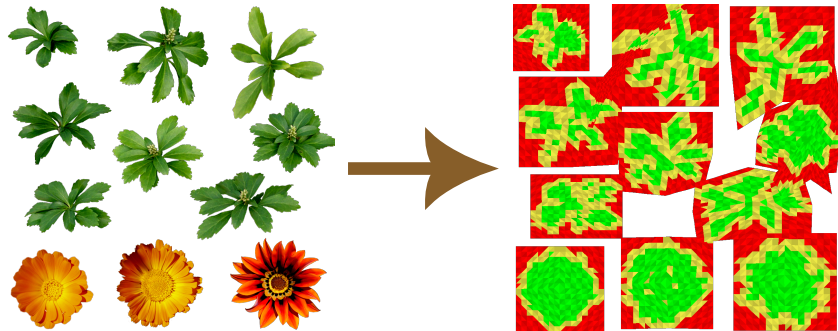
└ Succinct Opacity Micromaps

Succinct Opacity Micromaps

---



# Succinct Opacity Micromaps

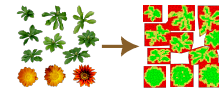


2025-01-11

└ Succinct Opacity Micromaps

└ Succinct Opacity Micromaps

Succinct Opacity Micromaps

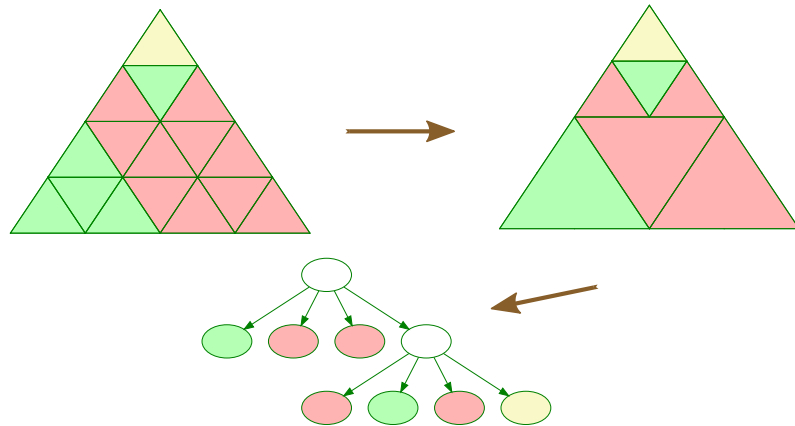


Which leads us to what is arguably our *main* contribution in this paper: the compression algorithm.

As I worked with the micromaps, I found plenty of cases where large swathes of the map would contain the same value, some examples you can see in this small texture atlas from our good old Sponza scene. As such, I looked at the `index`-mapping algorithm and thought: You know, wouldn't it be a good idea if we *didn't* have to go all the way to the bottom to retrieve the value if we know that all values in that sub-area are going to be the same? That is, having a kind of *marker* telling us that there is not going to be anything new after this?

# Succinct Opacity Micromaps

## Tree Construction

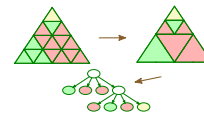


2025-01-11

└ Succinct Opacity Micromaps

└ Succinct Opacity Micromaps

Succinct Opacity Micromaps  
Tree Construction



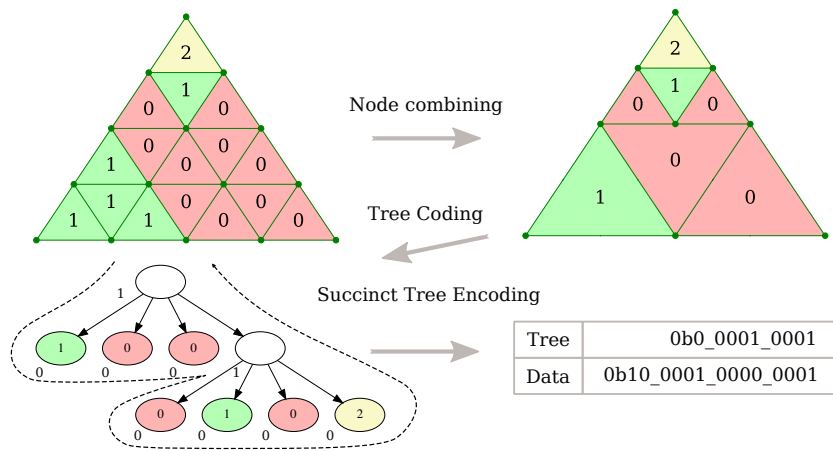
And that is more or less what I did: I converted the micromap to a 4-way tree structure where each node represents the area of all underlying sub-triangles. (Effectively mimicking the call-stack of the indexing algorithm.)

There are a few ways of representing such a tree: Implicitly, in an array; the same way we often represent binary trees, or using pointers.

Storing the tree implicitly would be ideal, but the trees are not likely to be balanced; so this would lead to a lot of wasted space. Pointers could naturally handle this aspect better, but the values we are storing are very small, so adding pointers to them felt like step in the wrong direction.

# Succinct Opacity Micromaps

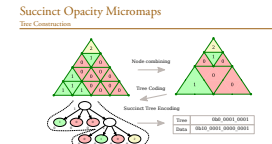
## Tree Construction



2025-01-11

## Succinct Opacity Micromaps

## Succinct Opacity Micromaps



However, by using a type of data-structure referred to as a *succinct* data-structure we can store the tree in an almost optimal way; and one way of creating such a structure is by traversing the tree in a depth-first fashion, setting a bit to 1 for every internal node, and 0 for every leaf node. The resulting bit-string will thus uniquely represent the tree, and any node data, such as opacity values, can be tucked on at the end of this string.

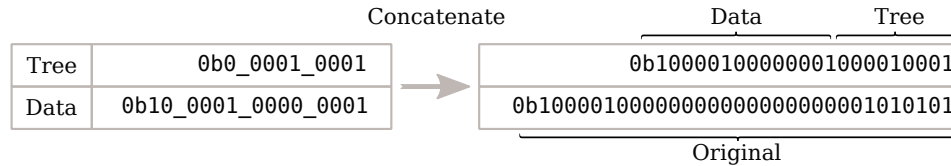
Thus, the way I compressed micromaps are as follows:

1. In a bottom-up fashion, combine any group of sub-triangles with the same value to a single node.
2. Repeat this for all subdivision levels, forming our 4-way tree, and, then
3. Encode this *succinctly* as a bit-stream.
  - And for clarity, the tree and opacity values, or *data* bits are shown separately here.



# Succinct Opacity Micromaps

## Tree Construction



2025-01-11

Succinct Opacity Micromaps

Succinct Opacity Micromaps

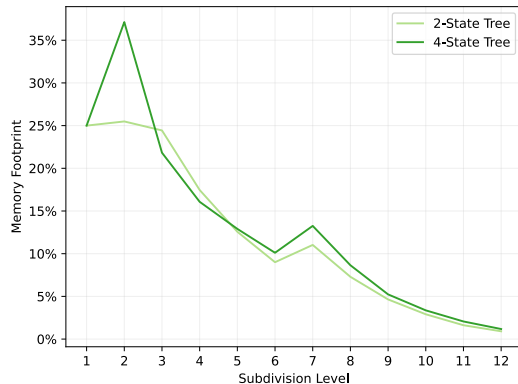
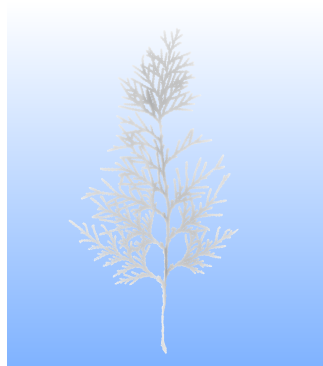
Succinct Opacity Micromaps

Tree Construction

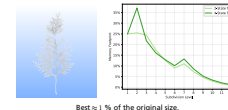


And even in this relatively small case, by comparing final bit-string to the original, it is pretty clear that there are some potential for compression here.

# Memory Footprint Improvement



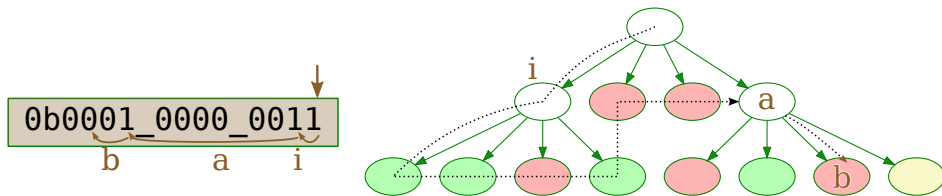
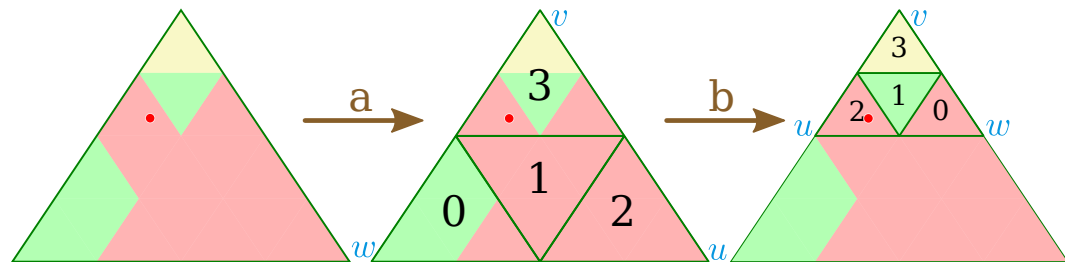
Best  $\approx 1\%$  of the original size.



And looking at the average results over all scenes in this work, this turned out to be very effective at reducing the footprint of the micromaps: Typically down to about 45-25 % of the original size, but in some extreme cases such as for this twig from the New Sponza scene, down to less than 1 %, or expressed another way: compressed 110 times.

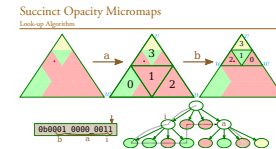
# Succinct Opacity Micromaps

## Look-up Algorithm



## Succinct Opacity Micromaps

## Succinct Opacity Micromaps



However, we cannot just compress the micromap, we also need a way of looking-up values from this new representation. We can do that as follows, starting from the root node:

1. After intersecting a triangle, represented by this dot, figure out the index of the child we should visit, and update the barycentric coordinates to that sub-triangle; both of which can be extracted with a single step from the indexing algorithm I presented earlier.
2. This number is now a counter: We need to step at least this number of bits into the bit-string that represents the tree to find the next node to visit.
  - 2.1 However, If we encounter another internal node before this counter is zero, we need to *also* pass through all children owned by it as well, as you can see at the node and bit locations labeled *i*. In other words, we increment our counter by 4.
3. Finally, when this counter is 0, we have found the next node to investigate:
  - 3.1 If the bit value is 1: We repeat the above procedure, going deeper into the tree.
  - 3.2 If the value is 0: We are done and the number of leaf nodes, that is, zeros passed until this point is the index into the trailing data bit-stream where we will find our opacity value.

# Frametime Performance

---



2025-01-11

└ Frametime Performance

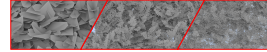
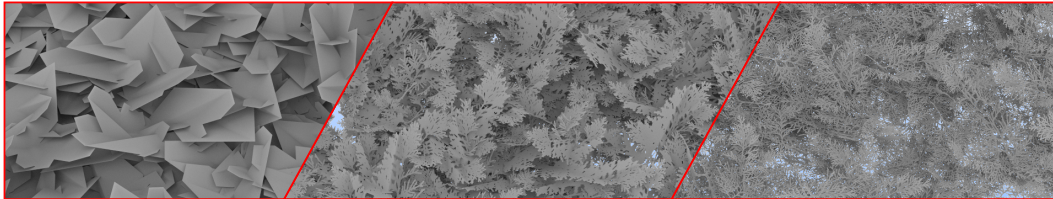
Frametime Performance

---



# Methods

- Software Micromaps
- Succinct Tree
- Fast-Build Micromaps
- Fast-Trace Micromaps
- Bitmask
- Texture



And with a method for looking up values, we obviously need to compare it against other ones. Thus, we have looked at the frame-time performance, and in total, investigated six different opacity algorithms:

*Micromap* Micromaps emulated in software,

*Tree* the tree encoded micromaps I just described,

*Fast-Build (FB)* Vulkan based micromaps built with the *Fast-Build* flag,

*Fast-Trace (FT)* Vulkan based micromaps built with the *Fast-Trace* flag,

*Bitmask* A bitmask where every 1 or 2 bits represents an opacity value, and finally,

*Texture* by using the original alpha texture.

In all cases except the texturing method, we evaluate both the 2-state and 4-state modes.

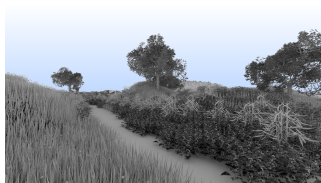
Perhaps a bit arbitrary, we also lock the micromap subdivision levels to a fixed value for each evaluation. And, as you can see here, for 2-state micromaps this will determine the final look of the scene, but has no aesthetic effect on 4-state ones.

(Typically, the subdivision level should be set on a per-triangle basis as discussed in the paper).

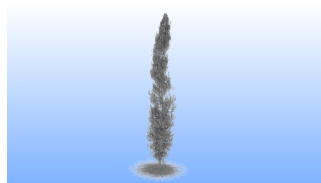
# Scenes



CryTek Sponza [2011]



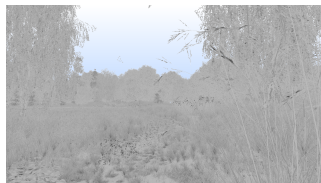
Ecosys [1998]



New Sponza [2022]



San Miguel [2010]



Landscape [2016]

2025-01-11

└ Frametime Performance

└ Scenes

Scenes



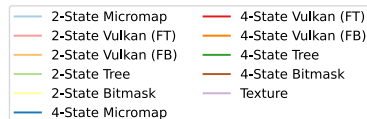
As for the scenes: Micromaps are really only useful in scenes with lots of alpha masks, so I chose to use the following scenes to try out these methods, which includes a decent sampling of old and modern content and several orders of geometric complexity.

And as I hope you can see from the images, I only used a basic refining ambient occlusion rendering algorithm to keep things simple.

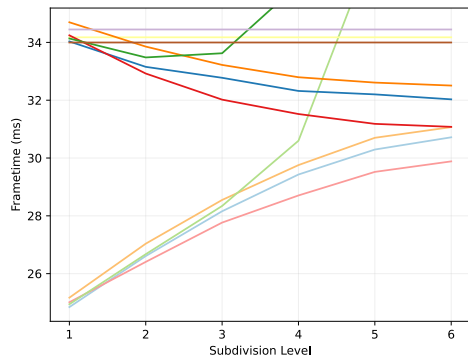
(This also means that more complicated shading algorithms would likely benefit more than what these results are showing, but that is out-of-scope for this paper.)

# Frametime Performance

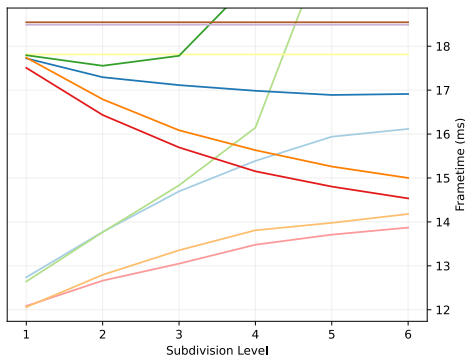
General Results — Landscape [2016]



RTX 3080



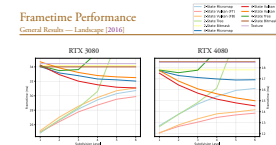
RTX 4080



2025-01-11

Frametime Performance

Frametime Performance



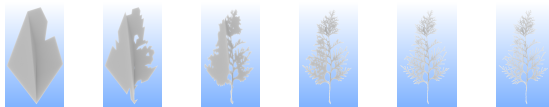
Thankfully, we do not really have to look at reams of data to discern any particular pattern: Basically all scenes have plots with roughly the same shape, and In short, we can see the following:

- Using micromaps, software or hardware based, is in general better than using either bitmasks or textures, but not always by a large margin:
  - And, for software based micromaps, you got a bit of a mixed bag of results: Sometimes you lost as much as 30 % or gained up to 16 %,
  - Whereas a 4080 with hardware acceleration would only lose at most 2 %, but gain up to 29 %.
- The tree-look-up algorithm only performs okay up to 3 or 4 levels, after which it scales out of control. Something I will get back to in a bit.
- (Interestingly, I was expecting the fast-build and fast-trace to perform basically equivalently, but there is a noticable delta between these results, if small (orange/red)).

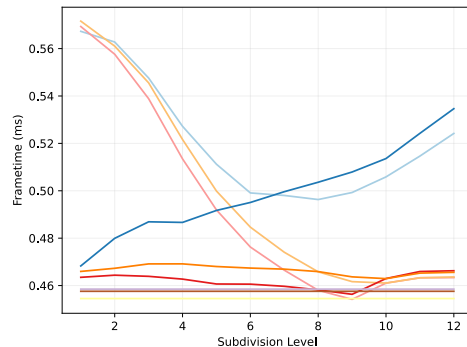
(Errata message: The paper is incorrectly duplicating the Ecosys plots for the Landscape frametime results. These plots refer to the scenes instead.)

# Frametime Performance

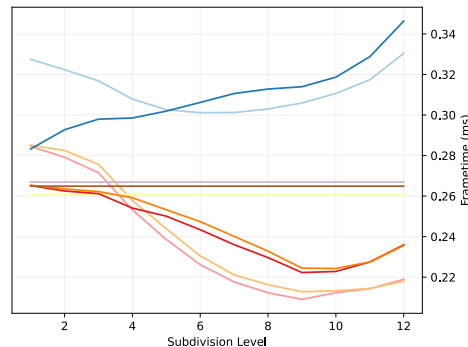
New Sponza [2022]



RTX 3080



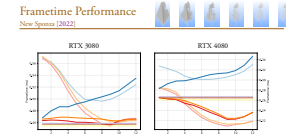
RTX 4080



2025-01-11

└ Frametime Performance

└ Frametime Performance



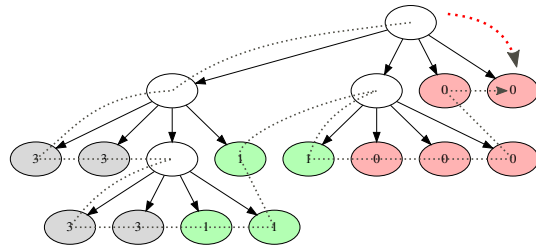
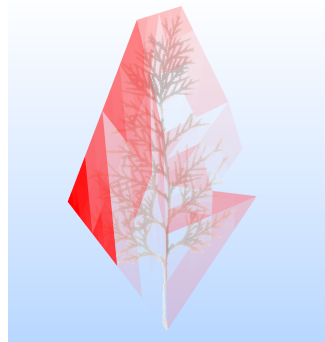
And if we change things around to a somewhat easier scene; a single twig from the New Sponza [2022] scene, we can see a bit more interesting performance patterns. Although, note that I have removed the tree results here to allow us to actually see things.

The most notable part here is that for a 3080, we will only ever benefit over just using the texture in this very small region around 9 subdivision levels. Similarly, for a 4080, we must have *at least* 4 subdivisions before we will see any benefits.

However, please note the axes on these plots: The ranges are relatively small, and the scene is a bit unrealistic.



# The Problem(s)

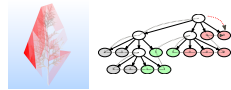


2025-01-11

└ Frametime Performance

└ The Problem(s)

The Problem(s)



Which finally brings me back to the tree-method. Why does it perform so poorly? With the data packed so much more densely, surely caching should help it be at least comparable to the other methods?

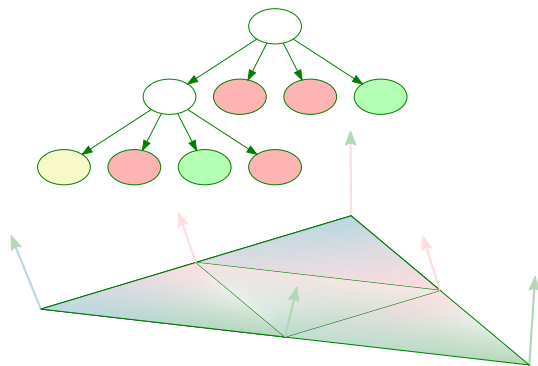
I believe the main issue is this: When we want to find one of the right-most children, we end up having to scan through all the intervening subtrees, as seen here.

This is fine when the tree is small, but is not acceptable as the tree gets bigger: On average it almost quadruples the number of bits that has to be scanned for each additional tree-level.

This is obviously problematic for something that ideally should be comparable to an instant  $\mathcal{O}(1)$  look-up, similar to a texture fetch.

# Future Work

- Need a different (succinct) tree
- Other micromap types?
  - Lossy micromaps?
  - Generalized micromaps?



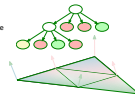
2025-01-11

## └ Frametime Performance

## └ Future Work

## Future Work

- Need a different (succinct) tree
- Other micromap types?
  - Lossy micromaps?
  - Generalized micromaps?



So, how would we fix this in the future? A tree-based structure *do* seem like a good fit for compressing micromaps, but this initial look-up algorithm requires some rework to be practical for larger maps or to make it feasible to realize in hardware along with the rest of the ray tracing pipeline. Thankfully, there are numerous ways of re-structuring the tree, one of which is bound to be more practical.

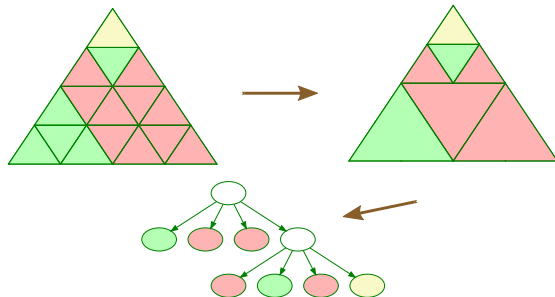
Also, there are currently only two types of micromaps: opacity and displacement. *But*, there seems like there is at least some work ongoing to create several other types of micromaps to store colors, normals or similar data. It would be interesting to see if such *Attribute* Micromaps could be compressed using this or similar techniques.

Further, I only considered lossless techniques during this work, but it may be interesting to also consider some kind of lossy compression to provide some approximation of level-of-detail for far-away objects.

And interestingly, the indexing algorithm I presented in the beginning is very generalizable: The same basic algorithm could be used on a number of different shapes, so investigating this further could lead to some interesting applications.

# Conclusion

1. Indexing Algorithm:
  - uv2index
2. Succinct Opacity Micromaps
3. Performance Comparison



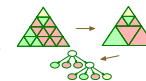
2025-01-11

└ Frametime Performance

└ Conclusion

Conclusion

1. Indexing Algorithm:
  - uv2index
2. Succinct Opacity Micromaps
3. Performance Comparison



But to wrap things up: As a part of this work, we have done the following:

- Provided an arguably more descriptive algorithm for mapping barycentric coordinates to micromap indices, and,
- Used said method to create a compression algorithm for micromaps based on Succinct Trees, which is capable of reducing the memory footprint by more than 110 times.
- We also evaluated the frame-time performance of this method, quickly realizing that while the method works well for small maps, it is not practical for larger ones in its current form.
- And finally, that result was compared to various other methods, as well as that of the official (Vulkan) Opacity Micromaps, which itself showed a performance uplifts by up to 29 %.

# Thanks for Listening

## Questions

---

- Thanks for listening!
- Questions and Answers



2025-01-11

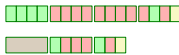
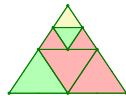
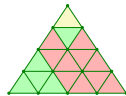
└ Frametime Performance

└ Thanks for Listening

And with all that, I think it's about time to open up for questions, and hopefully, answers!



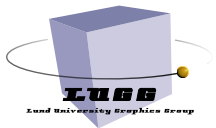
# Acknowledgements



## Succinct Opacity Micromaps

Gustaf Waldemarson   Michael Doggett

*This work was partially supported by the Wallenberg AI, Autonomous Systems and Software Program (WASP) funded by the Knut and Alice Wallenberg Foundation.*



Special thanks to...

- Rikard Olajos @ LUGG
- Simone Pellegrini @ Arm
- Mathieu Robart @ Arm



G.Waldemarson

20/20

2025-01-11

└ Frametime Performance

└ Acknowledgements

Acknowledgements



Succinct Opacity Micromaps

Gustaf Waldemarson   Michael Doggett

This work was partially supported by the Wallenberg AI, Autonomous Systems and Software Program (WASP) funded by the Knut and Alice Wallenberg Foundation.



Special thanks to...

- Rikard Olajos @ LUGG
- Simone Pellegrini @ Arm
- Mathieu Robart @ Arm



The End

2025-01-11

└ Frametime Performance

Extras



2025-01-11

Extras

Extras



# Where may we *Lose* performance?



2025-01-11

└ Extras

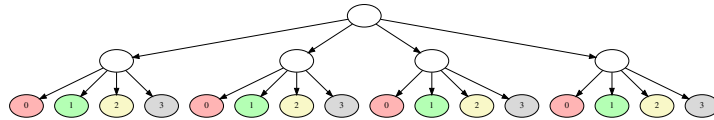
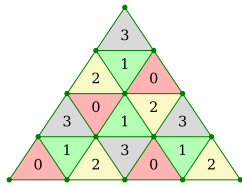
└ Where may we *Lose* performance?

Where may we *Lose* performance?

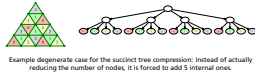




# Degenerate Trees

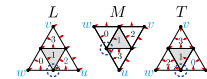
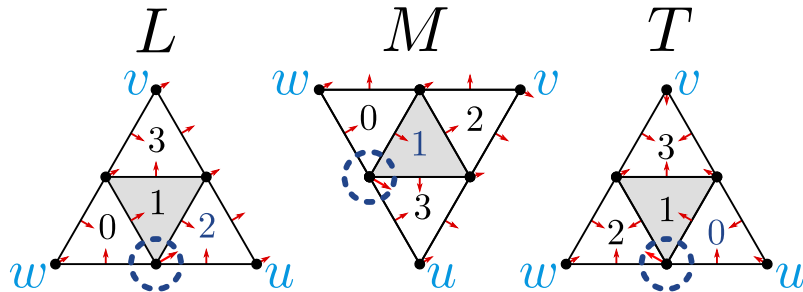


Example degenerate case for the succinct tree compression: Instead of actually reducing the number of nodes, it is forced to add 5 internal ones.



# Opacity Micromaps Splits

## Rounding Issues



This strategy will take you *most* of the way, but it will unfortunately not work for *all* cases. If we zoom in on a few of the subtriangles and highlight the rounding patterns, we get this kind of behavior. One *could* argue whether is wrong or not, but as the micromap extension actually have a *reference* algorithm, we know that the correct pattern should be like *this*. **(Next Slide)**

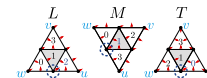
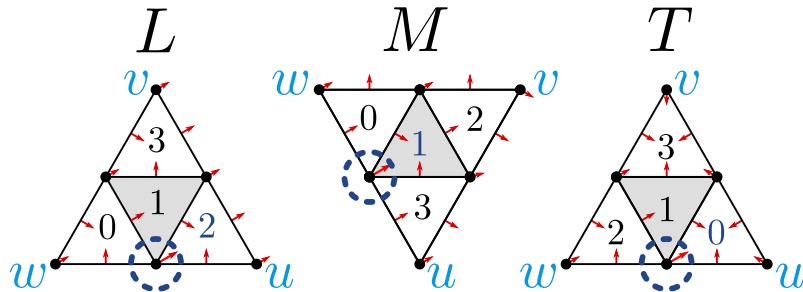
And here I have also highlighted a few vertices that ended being a bit troublesome. **(Flip Back and Forth)**

Thankfully, this is straight-forward to correct by keeping track of whenever we recurse into a *Middle* or *Top* subtriangle, but I will leave those details for the paper itself, as they are conceptually only necessary to ensure correct rounding in *literal* edge cases.

(We also validated that this new algorithm is equivalent to the official one by using the ACL2 solver, at least for rational numbers. It is still not clear whether this is true for real or floating point numbers however, but the extensive testing we have done so far at least suggests so.)

# Opacity Micromaps Splits

## Rounding Issues



This strategy will take you *most* of the way, but it will unfortunately not work for *all* cases. If we zoom in on a few of the subtriangles and highlight the rounding patterns, we get this kind of behavior. One *could* argue whether is wrong or not, but as the micromap extension actually have a *reference* algorithm, we know that the correct pattern should be like *this*. **(Next Slide)**

And here I have also highlighted a few vertices that ended being a bit troublesome. **(Flip Back and Forth)**

Thankfully, this is straight-forward to correct by keeping track of whenever we recurse into a *Middle* or *Top* subtriangle, but I will leave those details for the paper itself, as they are conceptually only necessary to ensure correct rounding in *literal* edge cases.

(We also validated that this new algorithm is equivalent to the official one by using the ACL2 solver, at least for rational numbers. It is still not clear whether this is true for real or floating point numbers however, but the extensive testing we have done so far at least suggests so.)

# Opacity Micromaps Lookup Function – 1

```
def uv2index(u, v, level):  
    w = 1.0 - (u + v)  
    def rec(idx, d, u, v, w):  
        if d == level:  
            return idx  
        L, M, R, T = 0, 1, 2, 3  
        if w > 0.5:  
            return rec(4 * idx + L, d + 1, 2*u, 2*v, (w - u - v))  
        elif v >= 0.5:  
            return rec(4 * idx + T, d + 1, 2*w, (v - u - w), 2*u)  
        elif u >= 0.5:  
            return rec(4 * idx + R, d + 1, (u - v - w), 2*v, 2*w)  
        else:  
            return rec(4 * idx + M, d + 1, (u + v - w), (w + u - v), (v + w - u))  
    return rec(0, 0, u, v, w)
```

Implemented in Python, the basic mapping algorithm could look like this.

```
def uv2index(u, v, level):  
    w = 1.0 - (u + v)  
    def rec(idx, d, u, v, w):  
        if d == level:  
            return idx  
        L, M, R, T = 0, 1, 2, 3  
        if w > 0.5:  
            return rec(4 * idx + L, d + 1, 2*u, 2*v, (w - u - v))  
        elif v >= 0.5:  
            return rec(4 * idx + T, d + 1, 2*w, (v - u - w), 2*u)  
        elif u >= 0.5:  
            return rec(4 * idx + R, d + 1, (u - v - w), 2*v, 2*w)  
        else:  
            return rec(4 * idx + M, d + 1, (u + v - w), (w + u - v), (v + w - u))  
    return rec(0, 0, u, v, w)
```

# Opacity Micromaps Lookup Function – 2

```
def uv2index(u, v, level):
    w = (1.0 - (u + v))
    def rec(i, d, mflip, tflip, u, v, w):
        if d == level: return i
        L, M, R, T = (2, 1, 0, 3) if tflip else (0, 1, 2, 3)
        if w > 0.5:
            return rec(4 * i + L, d + 1, mflip, tflip, 2*u, 2*v, (w - u - v))
        elif v >= 0.5 and not (v == 0.5 and mflip):
            return rec(4 * i + T, d + 1, mflip, not tflip, 2*u, (v - u - w), 2*w)
        elif u >= 0.5 and not (v == 0.5 and mflip):
            return rec(4 * i + R, d + 1, mflip, tflip, (u - v - w), 2*v, 2*w)
        else:
            return rec(4 * i + M, d + 1, not mflip, tflip,
                      (u + v - w), (w + u - v), (v + w - u))
    return rec(0, 0, False, False, u, v, w)
```

Implemented in Python, the final mapping algorithm could look like this.

```
def uv2index(u, v, level):
    w = (1.0 - (u + v))
    def rec(i, d, mflip, tflip, u, v, w):
        if d == level: return i
        L, M, R, T = (2, 1, 0, 3) if tflip else (0, 1, 2, 3)
        if w > 0.5:
            return rec(4 * i + L, d + 1, mflip, tflip, 2*u, 2*v, (w - u - v))
        elif v >= 0.5 and not (v == 0.5 and mflip):
            return rec(4 * i + T, d + 1, mflip, not tflip, 2*u, (v - u - w), 2*w)
        elif u >= 0.5 and not (v == 0.5 and mflip):
            return rec(4 * i + R, d + 1, mflip, tflip, (u - v - w), 2*v, 2*w)
        else:
            return rec(4 * i + M, d + 1, not mflip, tflip,
                      (u + v - w), (w + u - v), (v + w - u))
    return rec(0, 0, False, False, u, v, w)
```

# Succinct Opacity Micromaps

## Look-up Algorithm — 1

```
uint t = 0, d = 0;
while (true)
{
    bool is_internal = tree_bit(t);
    if (is_internal)
    {
        t += 1;
        uint c = step();
        t, d = bitscan(tree_len, c, t, d);
    }
    else
    {
        return opacity_value(tree_len, d);
    }
}
```

(Extra slide, describe tree-look-up algorithm)

```
uint t = 0, d = 0;
while (true)
{
    bool is_internal = tree_bit(t);
    if (is_internal)
    {
        t += 1;
        uint c = step();
        t, d = bitscan(tree_len, c, t, d);
    }
    else
    {
        return opacity_value(tree_len, d);
    }
}
```

# Succinct Opacity Micromaps




## Look-up Algorithm — 2

```
uvec2 bitscan(uint tree_len, uint child, uint t, uint d)
{
    while (t < tree_len && c > 0)
    {
        bool is_internal = opacity_tree_bit(t);
        if (is_internal)
            child += 4;
        else
            d += 1;
        t += 1;
        child -= 1;
    }
    return uvec2(t, d);
}
```

(Extra slide, describe bitscan algorithm)




```
uvec2 bitscan(uint tree_len, uint child, uint t, uint d)
{
    while (t < tree_len && c > 0)
    {
        bool is_internal = opacity_tree_bit(t);
        if (is_internal)
            child += 4;
        else
            d += 1;
        t += 1;
        child -= 1;
    }
    return uvec2(t, d);
}
```

# References I

-  Oliver Deussen, Pat Hanrahan, Bernd Lintermann, Radomir Mech, 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. ISBN: 0897919998. DOI: [10.1145/280814.280898](https://doi.org/10.1145/280814.280898).
-  Morgan McGuire Frank Meinel Marko Dabrovic. 2011. *CryTek Sponza*. <https://www.cryengine.com/asset-db/product/crytek/sponza-sample-scene>. (2011).
-  Holger Gruen, Carsten Benthin, and Sven Woop. Aug. 2020. "Sub-Triangle Opacity Masks for Faster Ray Tracing of Transparent Objects." *Proc. ACM Comput. Graph. Interact. Tech.*, 3, 2, (Aug. 2020). DOI: [10.1145/3406180](https://doi.org/10.1145/3406180).
-  Timm Dapper Jan-Walter Schliep Burak Kahraman. 2016. *Landscape*. <https://www.laubwerk.com>. (2016).



## References II

-  Guillermo M. Leal Llaguno. 2010. *San Miguel*. <https://www.pbrt.org/scenes-v3/>. (2010).
-  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>. (2022).
-  Eric Werness. Aug. 24, 2022. *VK\_EXT\_opacity\_micromap*. The Khronos Group Inc. (Aug. 24, 2022). Retrieved May 11, 2023 from [https://registry.khronos.org/vulkan/specs/1.3-extensions/man/html/VK\\_EXT\\_opacity\\_micromap.html](https://registry.khronos.org/vulkan/specs/1.3-extensions/man/html/VK_EXT_opacity_micromap.html).