High-Performance Graphics (2023)
J. Bikker and C. Gribble (Editors)

Compressed Opacity Maps for Ray Tracing

S. Fenney[1] and A. Ozkan[1]

[1]Imagination Technologies, UK

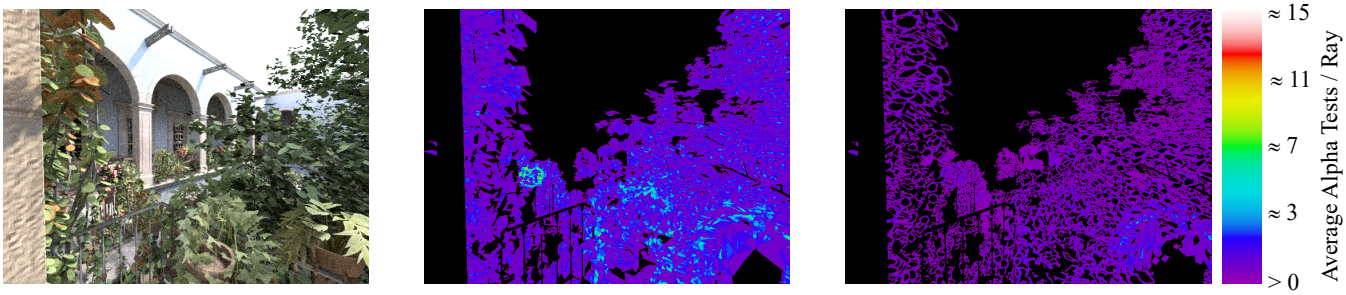

Figure 1: *Left: a view of 'San Miguel' (PBRT model, camera 20, 640x480, 64 primary rays/pixel). Centre: Heat map of 17.5M alpha tests performed on* primary *rays. Right: Heat map of alpha tests with (approximated) opacity maps - 4.6M tests.*

Abstract

Recently, schemes have been proposed for accelerating 'alpha-tested' triangles in ray-tracing through the use of precomputed, three-level Opacity Masks/Maps that can significantly reduce the need for expensive of 'Any-Hit shader' invocations. We propose and compare two related schemes, VQ2 and VQ4, of compressing such maps that provide both random access and low-cost decompression. Each compressed opacity map, however, relates to a pair of adjacent triangles, taking advantage of correlation across the shared edge and matching likely underlying hardware primitive models.

CCS Concepts

• ***Computing methodologies*** → ***Ray tracing; Visibility;***

1. Introduction

Contemporary graphics APIs, such as DirectX® and Vulkan®, support ray tracing of alpha-tested geometry by requiring relevant ray-triangle intersections be subsequently refined by running an 'Any-Hit' shader to determine visibility/presence within the triangle, [Khr23a]. (To avoid potential confusion, the APIs' 'Any-Hit' differs in meaning to the more traditional usage as in, e.g., [Smi98].) Given that the traversal of the Bounding Volume Hierarchy [Cla76], and triangle intersection tests may be performed by dedicated, pipelined hardware, interrupting that process with shader execution generally results in a significant performance penalty. Having a means to reduce these interruptions that i) does not significantly increase the memory bandwidth nor introduce additional memory latency, and ii) has only a small impact on the silicon area of the hardware intersection test units, is very desirable.

A related problem occurs in rasterisation where alpha testing needs to be logically done prior to the depth buffer update. Such a test can interfere with the efficiency of both Tile-based Deferred Renderers (TBDRs), e.g. [Ima], or early Z-test systems, [GKM93]. To mitigate this, Howson [How15] proposes an 'Opacity State Map' that stores a low-resolution, low-precision version of the texture's alpha component, reducing it to just four states. This map is sampled in advance of the normal Z/Alpha test. The first two states are 'fully opaque' and 'fully transparent'. These, which we will refer to as O and T, result in an automatic pass/fail, avoiding a full alpha test. The third and fourth states, 'Partially Transparent' and 'Mixed', both require the full alpha-test process. We shall refer to the union of these latter states as 'Check Texture', C. Although Howson's 'Opacity State Map' can be shared between multiple triangles, accessing it requires performing an upfront perspective-correct texture mapping operation, which is relatively costly.

DOI: 10.2312/hpg.20231133

With respect to ray tracing, [GBW20] instead propose storing a 'custom' per-triangle opacity mask, which is addressed via the triangle's barycentric coordinates as determined by a ray intersection. Figure 2 illustrates an example opacity mask for a portion of an alpha-tested triangle plus leaf texture, using 64 sub-triangles, providing an expected 53% reduction in shader invocation. Each sub-triangle in the mask has one of three states corresponding to Howson's *O*, *T*, and combined *C*. Akin to rasterisation, only the *C* state requires the shader execution. Depending on the resolution of the mask, Gruen et al found their technique provided a 19∼86% reduction in the expensive Any-Hit invocation, with the higher resolutions providing the greater savings. (Note, throughout this paper, 'savings' will refer to the reduction in any-hit execution made compared to a system without opacity masks/maps)

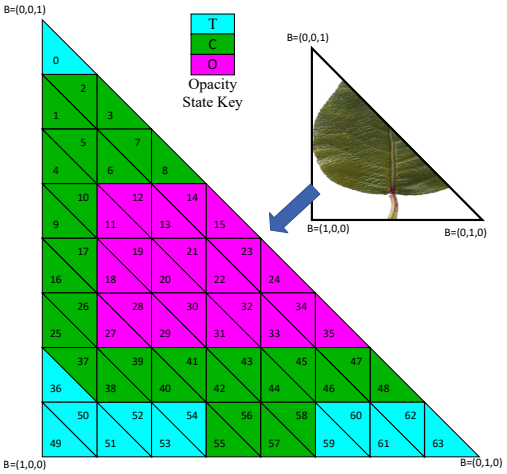

Figure 2: *Example of Gruen et al's Opacity mask with 64 sub-regions.*

To illustrate the potential savings, the left panel of Figure 1 shows a view of the 'San Miguel' model rendered with PBRT-v3, [PJH16], using 64 samples per pixel. The centre panel is a heat map illustrating the average number of alpha tests performed per pixel, i.e. effectively Any-Hit shader invocations, for *just* the primary rays. The right panel shows a simulation of using opacity maps that short-cuts the process. In the original, 17.5 million alpha tests were performed on the primary rays alone - using opacity maps, this reduces to 4.6 million. Note that the secondary rays also require significant alpha testing - approximately 71 million tests in this example - with the costs potentially exacerbated by the likely reduced coherency of those rays. Another example, the PBRT 'landscape' scene, is shown in Figure 12.

The Vulkan® Application Programming Interface has recently introduced an 'Opacity Micromap' extension, [Khr23b], similar to that of Gruen et al, but which instead supports both a four-state mode, comprising *O*, *T*, and two variants of *C*, and a two-state mode, with just *O* and *T*. The latter may, for example, be used for stochastic sampling in global illumination where potential aliasing may not be an issue. The micromap also differs from [GBW20] in that it employs a 'space filling curve' ordering for the sub-triangles rather than the 'raster' order. On some GPUs, [NVI23], the opacity micromap can achieve "a doubling of scene traversal performance in applications with alpha-tested geometry".

2. Initial Observations

The approach of [GBW20] creates maps for individual triangles but we note:

- In scenes such as 'San Miguel', Unreal Engine's 'Downtown West', or Amazon's 'Lumberyard-Bistro', foliage and flowers are usually modelled by a mesh of two or more contiguous primitives with a shared, alpha texture. Further, [NVI23] suggests a leaf "might be described using a couple of triangles". Similarly, one would expect, say, a chain link fence or dilapidated window to be formed from rectangular sections.

- At least two hardware ray tracing systems, Intel's [GB22] and Imagination's [Ima21], store and/or test rays against triangle pairs. There are perhaps three primary advantages for doing so:

  1. Bounding Volume Hierarchies, BVHs, generally use Axis Aligned Bounding Boxes, AABBs, ([MOB*21]). The AABB for a contiguous pair of triangles is rarely much larger than either of the two AABBs of the constituent triangles. Treating them as a pair will thus usually reduce BVH footprint, traversal depth, and AABB testing effort.
  2. When performing the ray vs. triangle-pair intersection tests, the calculations relating to the common edge can be reused in the evaluation.
  3. A triangle pair can be stored more compactly than two individual triangles, which can save memory bandwidth and increase cache utilisation.

The encoding scheme of [GBW20] uses 2 bits per sub-region but they note that three states could use "less than 2 bits". We assume this implies using $\approx log_2 3$ bits per state, as in some Block Truncation Compression schemes, [FNK94], perhaps encoding 5 regions in every 8 bits. Although this would be near optimal for equiprobable, random data, opacity maps are rarely 'noise'.

When performing alpha testing in Z-buffer rasterisation, unless a system has hardware support for automatic 'order independent translucency', [Seg98], or, perhaps, the application implements a multi-pass scheme such as [SML11], developers often just apply a threshold and use the 'discard' in the fragment shader to create a binary 'fully transparent/fully opaque' result. This avoids the need to sort geometry but typically introduces artefacts.

Ray tracing, on the other hand, orders the intersections. We thus expect applications may take advantage of partial transparency particularly for primary, reflection/refraction, and point-light shadow rays. Therefore, when accessing the source texture map, the sampling process in the any-hit shader is likely to at least use bilinear filtering. This implies that sampled alpha values will be continuous across the triangle and thus it should be impossible, in practice, to change instantly from fully opaque to fully transparent. (Due to the likely difficulty of exactly matching the floating-point calculation results of mask/map creation with that of the sampling performed by the GPU, we feel it would be unwise to use point-sampling in

the shader. Taking a conservative approach, i.e. using a continuous filter, seems prudent.)

Finally, given today's memory latencies, it would be advantageous for the opacity map to be stored adjacent to, and read at the same time as, the triangle vertex data and, preferably, to be of comparable size.

3. Compressed Opacity Maps

We propose storing a *square* opacity map for each *pair* of adjacent triangles and using square sub-regions with, e.g. 16x16 (Figure 3) or 32x32 resolution. It is expected that there will be continuity across the shared triangle edges. Further, given the computed barycentric coordinates, square sub-regions should be marginally simpler to index than sub-triangles.

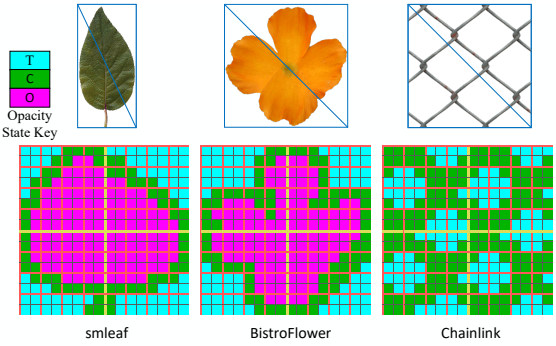

Figure 3: *Examples of textured triangle pairs with 16x16 Opacity Maps.*

A lossy compression scheme is employed to reduce the memory footprint. Fast random access to the opacity map is essential, but many compression methods, e.g. Huffman, do not permit constant-time decoding of an arbitrary value. Several compression schemes were investigated, e.g. a 3-level quad-tree hierarchy and a 'wavelet mod 3', with the aim of providing fast 'random access', but Vector Quantisation, (VQ), [GG91], particularly appealed due to its simplicity to decode in hardware. VQ can be summarised as a system that takes an input array of J, n-dimensional vectors and approximates it using i) a 'codebook' which is an array of K vectors, $K < J$, and ii) J indices, one per input vector, each of which identifies the corresponding'best' representative in the codebook.

VQ has been used in graphics for compression of image data, [Hec82], and textures, [BAC96] which, in turn, inspired the 2bpp and 1bpp texture compression in the Dreamcast console [Seg98]. It fell out of favour mainly because of the expense of overcoming the memory latency of the dependent codebook access. If, however, the indices and codebook are contiguous and both loaded simultaneously in 'local storage', the latency associated with the indirection can be eliminated. Utilising VQ, storage costs of around 1 bit/region for a 16x16 opacity map or even 0.5 bits/region for 32x32 are feasible. Such maps would be similar in size to a cache line, typically 32∼128bytes.

With any fixed-rate compression scheme, however, there must exist source data that cannot be compressed losslessly and opacity maps are no different. We note, though, that it is always safe to replace an O or T state in the source map with a C in the compressed variant. Unfortunately, such a substitution comes at performance penalty since that replacement would result in a shader invocation that the opacity map is aiming to reduce. [GBW20] demonstrated that higher resolution opacity maps provide greater reductions in shader execution due to the higher proportions of O and T states. This raises the question: given an assumption of a fixed-storage budget for each opacity map, will the relative gain in performance that is achieved by compression - thus permitting higher resolutions - significantly outweigh any losses due to conservative substitutions?

The paper will now describe and compare two VQ-based compression schemes: VQ2, storing 16x16 maps at 1 bit/region, and VQ4, which stores 32x32 maps at 0.5 bits/region.

4. VQ2: A 2x2 VQ, 1 bit/region scheme

For the 1 bit/region compression mode, VQ2, a 16x16 map resolution was chosen, giving a storage budget of 256 bits that must at least contain the vector indices and codebook. The map is divided into 64, 2x2 vectors, and the codebook stores a small subset of the set of all possible vectors, PV_{2x2}. Initially, it may *appear* that $|PV_{2x2}| = 3^4$, each thus requiring ≈ 7 bits to encode. However, by the continuity assumption in section 2, O and T states cannot co-exist in the same vector. This implies $|PV_{2x2}| = 31$, suggesting the 5-bit encoding of figure 4 for each entry in the codebook; the 'Palette Mode' flag indicates if the per-region modes are chosen from either $\{O,C\}$ or $\{T,C\}$, and the remaining fields are 1-bit indices into the selected palette.

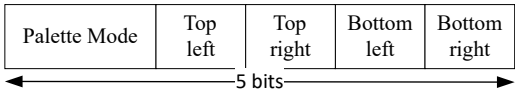

Figure 4: *5-bit codebook vector encoding*

The VQ encoding also requires 64 indices. To meet the 256-bit budget, each index clearly must have fewer than 4 bits. An index size of 3-bits was chosen as a 2-bit index was too restrictive. This leaves 64 bits for the codebook and additional fields.

Example scene data indicated that the three 2x2 vectors which are uniformly O, T, or C, are particularly common, each typically occurring with probabilities of 11∼29%. Rather than explicitly placing these in a codebook, it seemed prudent to dedicate 3 indices, e.g. $\{0,1,2\}$, to implicitly represent $\{T,C,O\}$, leaving 5 indices to access explicitly stored vectors.

4.1. Utilising Symmetry

From observations of model data, it became apparent that the natural symmetries, both reflections and rotations, in the source opacity maps could be utilised to improve compression. To illustrate, we take the smleaf example of figure 3, divide the map into quadrants and, for clarity, remove the 'uniform' vectors, as shown in figure 5.

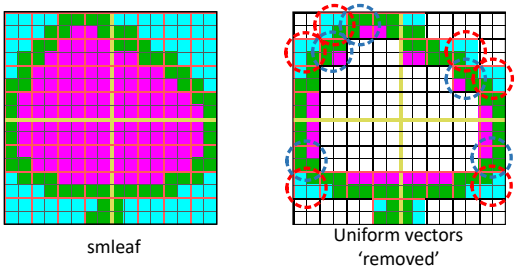

Figure 5: *Examples of symmetry in opacity map*

Using the top left quadrant, *TL*, as a reference, we note that the vectors circled in red also appear in the bottom left quadrant, *BL*, but rotated 90° anticlockwise, and again in the top right, *TR*, and bottom right, *BL*, except rotated by 270° and 180° respectively. Identical behaviour can be seen with the vectors circled in blue. In other samples, reflections about the x-axis, y-axis, and the lines $x = y$, $x = -y$ were observed.

To utilise this, the compression scheme stores a transformation 'matrix' for the *TR*, *BL*, and *BR* quadrants, each chosen from the set, {*Identity, Rotate 90° Anticlockwise, Reflect Top/Bottom, Reflect X=-Y, Reflect Left/Right, Reflect X=Y, Rotate 180°, Rotate 270° Anticlockwise*}. The *TL* quadrant's transform is always the identity. When a stored codebook vector is accessed, the associated quadrant's transformation is applied.

Any of the above set of 8 can be constructed using just three base transforms, [*Reflect Left/Right, Reflect Top/Bottom, Rotate 90° Anticlockwise*], selectively applying these, in turn, to a vector. The encoding is summarised in table 1. In hardware this requires 12, 2-input multiplexor units and a per-quadrant 3-bit field, indicating which of the base transforms to apply. (Alternatively, an equivalent process can be applied to the coordinates accessing the contents of the vector - this latter approach is preferable for the 0.5 bit/region method, VQ4, discussed in Section 5). It should be noted that, although the transformation process was inspired by observed symmetries, it can also be regarded simply as a method of synthesising additional vectors from those stored in the codebook.

Reflect_{LR}	Reflect_{TB}	Rotate_{90}	Result
0	0	0	Identity
0	0	1	Rotate_{90}
0	1	0	Reflect_{TB}
0	1	1	$\text{Reflect}_{X=-Y}$
1	0	0	Reflect_{LR}
1	0	1	$\text{Reflect}_{X=Y}$
1	1	0	Rotate_{180}
1	1	1	Rotate_{270}

Table 1: *Per-quadrant transform encoding/construction. Operations are performed right to left.*

The remaining 55 bits store a total of 11, 5-bit vectors in a code book. Three are shared between all quadrants and are accessed via indices $\{3,4,5\}$. The remaining indices, $\{6,7\}$, access per-quadrant codebook entries. Although the arbitrary ordering of the vectors relating to $\{3,4,5\}$ and the four pairs corresponding to $\{6,7\}$ could *theoretically* provide an additional $\lfloor log_2(3!) \rfloor + 4$ encoding bits, we have not investigated utilising this.

An overview of the stored data and the decode process is shown in figure 11.

4.2. Compression Algorithm

With reference to algorithm 1, we coded a naïve compressor that brute-forces all 2^9 quadrant transform combinations. For each such combination, it applies a greedy algorithm that simply chooses the global and local codebook candidates *directly* from the input vectors, i.e. selecting in turn those that contribute to the fewest total *C* states. The choice of assigning a vector to global or per-quadrant codebooks is a simple heuristic of how many quadrants contain that vector. Once the codebook is determined, each source vector is assigned the best available representative and a score is calculated. The transform and associated codebook with the least score is chosen.

4.3. Compression Results

In evaluating VQ2, a set of 170 alpha textures, drawn from various sources including 'San Miguel', 'Lumberyard-Bistro', Unreal Engine's 'Downtown West', PBRT's 'landscape', as well as textures previously provided by developers for texture compression evaluation, were de-noised of spurious outliers, cropped to rectangles that removed excess transparent borders, and were then assumed to map to triangle pairs. Images that were clearly intended to be 'texture atlases', e.g. a collection of several flowers, were first manually subdivided into smaller images. Note that the assumption of mapping to just a pair of triangles is at times not ideal - a small number of the set would be better mapped to more triangles that more tightly bound the non-transparent areas.

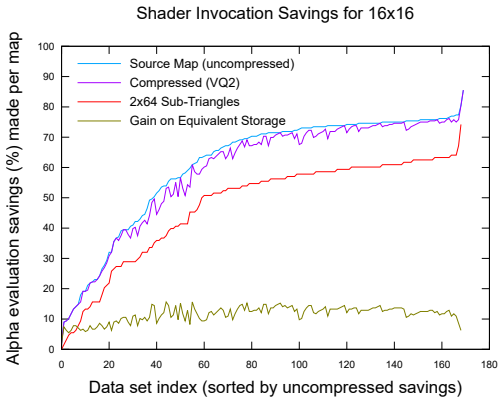

Figure 6: *VQ2: Savings across test set: source, compressed, vs two 64-sub-triangle opacity masks/micromaps*

To quantify the behaviour of the lossy compressor, the reduction in alpha tests using opacity maps was collated across the test set for both the original and compressed 16x16 maps. The results are shown in Figure 6; the maps here being ordered from those

Algorithm 1 VQ2 Compression

```
BestScore ← INT_max
Vecs ← InputVecs without uniform vectors
for all T possible transform combinations do
    LocalVecs ← apply T to Vecs
    OrderedVecs ←sort LocalVecs
    for all V in OrderedVecs do
        V.score ← count number of C states
    end for
    MergedVecs ←combine OrderedVecs duplicates, sum scores
    PrioritisedVecs ←sort MergedVecs by decreasing score
    CreateCodeBook( PrioritisedVecs, LocalCodeBook)
    LocalScore ← EvaluateCodeBook(LocalVecs, LocalCodeBook)
    if LocalScore < BestScore then
        BestScore ← LocalScore
        CodeBook ← LocalCodeBook
        if BestScore = 0 then
            break
        end if
    end if
end for

procedure CREATECODEBOOK(PrioritisedVecs, CB)
    CB.Globals ← []                          ▷ clear codebook entries
    CB.Quadrants ← []
    while (CB not full) ∧ (PrioritisedVecs not empty) do
        V ← Take lowest scoring vector from PrioritisedVecs
        Dest ← unassigned      ▷ Codebook assignment heuristic
        if V only in one quadrant, q then
            if CB.Quadrant_q not full then
                Dest ← AddToQuadrant
            else if CB.Globals not full then
                Dest ← AddToGlobal
            end if
        else                                 ▷ V is in multiple quadrants
            if CB.Globals not full then
                Dest ← AddToGlobal
            else
                Dest ← AddToQuads
            end if
        end if

        if Dest = AddToGlobal then
            CB.Globals+=V
        else if Dest = AddToQuads then
            for all q in V.quadrants do
                if CB.Quadrant_q not full then
                    CB.Quadrant_q+=V
                end if
            end for
        end if
    end while
end procedure
```

with the greater proportion of C states, thus with the least savings, through to those with the fewest C states, i.e. the greatest savings. The relatively small difference in performance between source and compressed data indicates that, despite the simplistic compressor, relatively few replacements of O or T with C are made.

As a benchmark, the graph also plots a 256-bit encoding using the 2bit/region opacity mask/micromap scheme with two sets of 64 sub-triangles. Given the same memory budget, the lossy compression typically achieves a gain in excess of 10%.

Finally, across the 170 images, 80 of the possible 512 transform combinations were used. Of these, 50 were unique, while the most frequently used, selected in 15% of the cases, was [*TR: Rotate 90°, BL: Reflect Top/Bottom, BR: Reflect X=Y*].

5. VQ4: A 4x4 VQ, 0.5 bit/region scheme

Because higher resolutions provide greater shader savings - albeit with diminishing returns (see figure 7) - we wanted to explore using a larger opacity map, i.e. 32x32, yet avoid quadrupling the indexing storage compared to a 16x16. A more ambitious approach, VQ4, featuring 4x4 vectors with a target of 0.5 bits per region was thus evaluated. VQ4 applies the same fundamental principles as VQ2, however the details differ. As with VQ2, the map has 64 vectors but the larger memory footprint allows the 'luxury' of 4-bit indices. VQ4 also re-uses the quadrant transformation encodings of VQ2. Together, these leave a budget of 247 bits for the codebook.

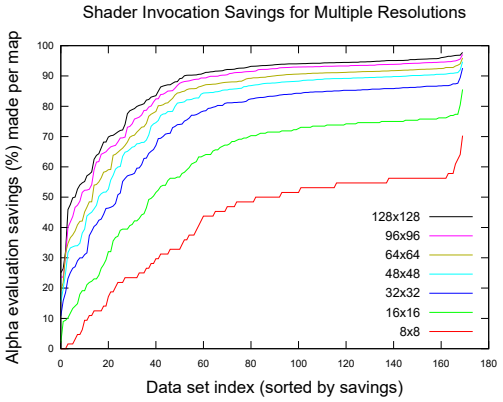

Figure 7: *Uncompressed opacity map savings for multiple resolutions. Note that savings 'per region' soon diminish.*

5.1. Codebook vector representation

Using 4x4 vectors instead of 2x2 implies some new limitations. From the continuity assumption, rather than having 3^{16} (≈4.3e7) possible vectors, one can demonstrate that $|PV_{4x4}| = 231713$, i.e. $< 2^{18}$. Encoding and decoding using the ideal 18 bits, however, seemed impractical. Instead, a completely arbitrary 4x4 vector is considered to comprise four, 2x2 mini vectors, each of which reuses the scheme of figure 4, yielding a 20-bit encoding.

We note that there is a correspondence between each 4x4 and an equivalent 2x2 in a lower resolution, 16x16 map. It therefore

follows that any uniformly O or T 2x2 vector in a 16x16 map, implies the matching 4x4 vector in the 32x32 map must also be, respectively, fully O or T and therefore frequently occurring. As with VQ2, VQ4 has a means to implicitly store such vectors. In the case of uniform C vectors, it is less beneficial; the purpose of a higher resolution map is to reduce the expected probability of C sub-regions, and thus it follows that uniform C vectors occur less often. Nevertheless, in this presentation it is assumed the uniform C vector will also be implicitly represented.

Although in the general case, both T and O states may coexist in a single 4x4 vector, there are frequent occurrences of vectors composed only of $\{O,C\}$ or $\{T,C\}$, thus allowing a more compact 17-bit representation, similar to that of VQ2. Further, by identifying cases with horizontal or vertical partitions of $\{O,C\}$ and $\{T,C\}$ allows 18-bit encodings. The VQ4 scheme evaluated thus uses the codebook encodings of table 2 which meets the 247-bit budget. VQ4 does not use the per-quadrant codebooks of VQ2.

Index Range	Encoding	Storage Budget (bits)
0..2	Uniform T, C, or O	0
3..5	Simple $\{O,C\}$ or $\{T,C\}$	51
6	Vertical partition	18
7	Horizontal partition	18
8..15	Generic vectors	160

Table 2: *VQ4 Index/Codebook Assignment*

5.2. Compression Algorithm

The naïve algorithm of VQ2 is inadequate for the more ambitious 4x4 representation. Some VQ encoding algorithms, e.g. [Wu92], use principal component analysis to identify spatial splitting planes, but given each of the 16-dimensions only has 3 possible 'integer' values, attempting to partition sets of vectors with arbitrary 16-dimensional planes did not seem suitable. Instead, we opted to use an algorithm inspired by [Hec82], where partitioning simply uses the major axes.

As with the VQ2 compressor, VQ4 evaluates all 512 quadrant transform options, and for each determines a codebook. This latter step is summarised in algorithm 2. This maintains a list of partitions of the transformed vector set, beginning with a single partition containing all bar the uniform O, T, and C vectors, since these are 'free'. Nearly all partitions are represented by the best 'compatible' vector, i.e. one in which, for each dimension, if there is more than one of $\{O,T,C\}$ in the vectors of the partition in that dimension, the representative must use a conservative C. An illustration of the compatible vector generation process, for 2x2 vectors, can be seen in figure 8.

Each partition also maintains a 'loss' score which indicates the total number of conservative C substitutions that would be made if all vectors used the representative. A greedy algorithm picks the partition with the expected 'most to gain', i.e, the one with the largest 'loss', and divides it into two smaller partitions, selecting the axis and value which minimises the total loss. The process repeats until the number of partitions reaches the required number of

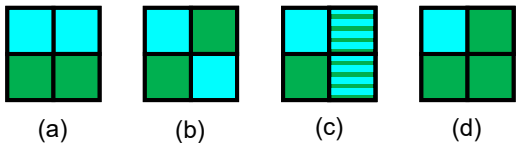

Figure 8: *Calculation of a compatible vector: (a) and (b) represent two source 2x2 vectors, (c) illustrates their overlap, while (d) shows the resulting best compatible vector*

codebook entries. This description has 'glossed over' the handling of the 17- and 18 -bit encodings listed in table 2 but we propose to post-process the codebook should there be, for example, too many 20-bit vectors. In practice, however, we have not yet found a case where this was required.

An example of the results of this process is shown in figure 9. Here, 85 of the 834 source O and T states were replaced with C.

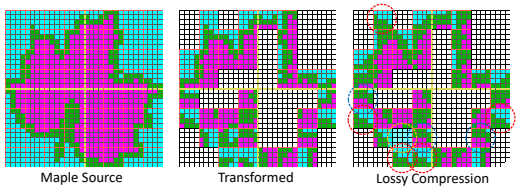

Figure 9: *'Maple Leaf' VQ4 compression example - source, transformed, & lossy vector representation. The codebook entry circled in red is not present in the transformed source but is the most compatible with the source vectors it represents. The example highlighted in blue represents one vector exactly and two others, conservatively. One, circled in orange, has been represented by a uniform* C *vector.*

5.3. Compression Results

As with VQ2, the data set was compressed and the relative savings compared against the source maps. These results are shown in figure 10. The drop in savings for the compressed data is relatively greater in VQ4 than VQ2 but, in general, it still performs around 10% better than an equivalent 512-bit, uncompressed map, i.e. 16x16, as was shown as the source map in figure 6.

6. Future Work

With the existing VQ2 and VQ4 modes, we believe that improvements could be made to the compressors to achieve fewer substitutions. The VQ4 compressor algorithm, for example, though far better than that used to evaluate VQ2, still employs a greedy approach - an alternative, back-tracking search might yield superior results. It may also be possible to find a means of utilising the unused 'implicit' encoding space - i.e. taking advantage of the otherwise arbitrary order of some of the codebook vectors, though this would require additional comparison hardware. One such use may be to offer an alternative to always assuming a uniform C vector.

Algorithm 2 VQ4 Codebook Generation

```
procedure CreateCodeBook(MergedVecs, CB)
    CB ← []                                   ▷ clear codebook entries
    Partitions[0].Members ← all MergedVecs
    Partitions[0].Rep ← AllC                  ▷ Force All Check texture
    Partitions[0].Score ← ComputeNumSubstitutions(Partitions[0])
    NumPartitions ← 1

    while (CB not full) ∧ (NumPartitions > 0) do
        P ← Take highest score partition from Partitions
        if P.Score = 0 then
            Partitions+=P                     ▷ lossless compression
            break
        end if
        BestPartScore ← INT_max
        for all non-constant dimensions, D, in P do
            for all possible partitions of D of P do
                Create partitions P_A and P_B from P using D
                if P.Rep = AllC then          ▷ maintain one AllC
                    Either P_A.Rep ← AllC or P_B.Rep ← AllC
                end if
                LocalScore ← Score(P_A) + Score(P_B)
                if LocalScore < BestScore then
                    BestParts_A ← P_A
                    BestParts_B ← P_B
                    BestScore ← LocalScore
                end if
            end for
        end for
        Partitions+=BestParts_A               ▷ including score and rep
        Partitions+=BestParts_B
    end while
    for all P in Partitions do
        CB+=P.Rep
    end for
end procedure
```

Our BVH builder should also be updated to produce, where possible, tighter AABBs around triangle pairs that have opacity maps, or even to discard some triangles; visual inspection of test scenes certainly suggests that some existing models could be made more efficient.

The 'Opacity Micromap' of [Khr23b] has a two-state mode, which *never* executes the Any-Hit shader, and is intended, say, for stochastic sampling use cases. Further, any four-state micromap can be interpreted as two-state. We plan to investigate a similar approach for interpreting the three-state opacity map as two-state, possibly by using the 2x2 vectors to determine hit/miss probabilities in combination with a hash of the hit's barycentric coordinate LSBs.

Some other questions remain to be answered:

- As the compression is lossy, is there an advantage in favouring one of T or O over the other when deciding whether to replace with C?
- For 'traditional' any hit use cases, [Smi98], such as shadow rays, where the order of intersections is not critical, when an alpha-tested intersection finds a C state, would it be beneficial to defer or queue the any-hit shader evaluation in the expectation that (opaque or) other alpha-tested geometry may find an O with lower overall cost?

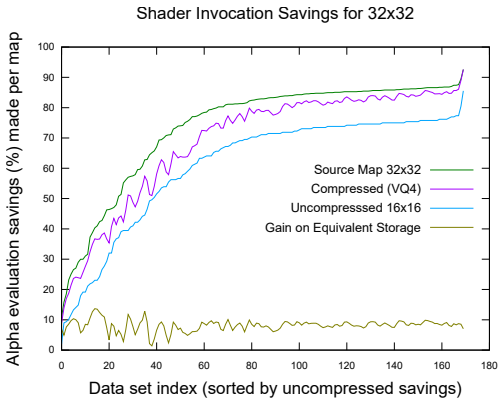

Figure 10: *VQ4: Alpha test Savings: Source vs Compressed vs Same storage cost*

6.1. 'Paint by Brush'

Although VQ-based methods have been presented, a number of different compression schemes were considered, such as wavelets and a hierarchy method - as mentioned in Section 3. However, there is another experimental approach we'd like to explore that we've termed 'Paint by Brush'. This method was inspired by Roger Johansson's Mona Lisa project, [Joh08]. Here, Johansson used an evolutionary algorithm to construct a replica of the Mona Lisa that is made up of translucent polygons, stacked on top of each other.

This approach can be adapted for opacity maps as well. A list of shapes or 'brushes' can be defined and applied on top of each other, similar to layers in image editing software, to approximate a given image. For example, placing an O square on top of a larger C background, would create a square of O state surrounded by a ring of C state. Additional shapes can be added on, similar to 'pressing a brush on paper' controlled by using brush type, size, rotation, centre position, and 'colour', i.e. $\{O, C, T\}$. Assuming a fixed number of brush operations, such a system allows random access for a given location by testing for inclusion in all the 'applied brushes' in parallel. The difficulty is in developing an efficient compressor: This method is likely intuitive to a human, but implementation in a program, without resorting to the slow, random mutation method of Johansson, is a more challenging task.

7. Conclusions

We have presented methods of storing compressed opacity maps that allow reductions in potentially expensive Any-Hit shader invocations which are close to that achieved by the uncompressed data, yet at 50% or even 25% of the memory budget. Alternatively, this can be seen as offering the benefits of a higher resolution for

the same storage footprint. Using a vector quantisation approach means that the decompression process is inexpensive, particularly for implementation in a hardware ray-triangle intersection tester. The data sizes are comparable to the triangle/triangle pair coordinate data and so can be loaded simultaneously to reduce latency without significant additional bandwidth.

Acknowledgements

We wish to thank our colleagues at Imagination Technologies for their support in this research and to the reviewers who provided excellent suggestions on improving the content of this paper.

References

[BAC96] BEERS, ANDREW C., AGRAWALA, MANEESH, and CHADDHA, NAVIN. "Rendering from Compressed Textures". *Proceedings of the 23rd Annual Conference on Computer Graphics and Interactive Techniques*. SIGGRAPH '96. New York, NY, USA: Association for Computing Machinery, 1996, 373–378. ISBN: 0897917464. DOI: 10.1145/237170.237276. URL: https://doi.org/10.1145/237170.237276 3.

[Cla76] CLARK, JAMES H. "Hierarchical Geometric Models for Visible Surface Algorithms". *Commun. ACM* 19.10 (1976), 547–554. ISSN: 0001-0782. DOI: 10.1145/360349.360354. URL: https://doi.org/10.1145/360349.360354 1.

[FNK94] FRÄNTI, P., NEVALAINEN, O., and KAUKORANTA, T. "Compression of Digital Images by Block Truncation Coding: A Survey". *The Computer Journal* 37.4 (Jan. 1994), 308–332. ISSN: 0010-4620. DOI: 10.1093/comjnl/37.4.308. eprint: https://academic.oup.com/comjnl/article-pdf/37/4/308/1067221/370308.pdf. URL: https://doi.org/10.1093/comjnl/37.4.308 2.

[GB22] GRUEN, HOLGER and BARZACK, JOSHUA. *A Quick Guide to Intel's Ray Tracing Architecture*. 2022. URL: https://www.youtube.com/watch?v=SA1yvWs3lHU 2.

[GBW20] GRUEN, HOLGER, BENTHIN, CARSTEN, and WOOP, SVEN. "Sub-Triangle Opacity Masks for Faster Ray Tracing of Transparent Objects". *Proc. ACM Comput. Graph. Interact. Tech.* 3.2 (2020). DOI: 10.1145/3406180. URL: https://doi.org/10.1145/3406180 2, 3.

[GG91] GERSHO, ALLEN and GRAY, ROBERT M. *Vector Quantization and Signal Compression*. USA: Kluwer Academic Publishers, 1991. ISBN: 0792391810 3.

[GKM93] GREENE, NED, KASS, MICHAEL, and MILLER, GAVIN. "Hierarchical Z-Buffer Visibility". *Proceedings of the 20th Annual Conference on Computer Graphics and Interactive Techniques*. SIGGRAPH '93. Anaheim, CA: Association for Computing Machinery, 1993, 231–238. ISBN: 0897916018. DOI: 10.1145/166117.166147. URL: https://doi.org/10.1145/166117.166147 1.

[Hec82] HECKBERT, PAUL. "Color image quantization for frame buffer display". *ACM Siggraph Computer Graphics* 16.3 (1982), 297–307. URL: https://dl.acm.org/doi/10.1145/800064.801294 3, 6.

[How15] HOWSON, JOHN. *Opacity Testing For Processing Primitives In A 3D Graphics Processing System*. 2015. URL: https://www.freepatentsonline.com/y2015/0221127.html 1.

[Ima] IMAGINATION TECHNOLOGIES. *Tile-Based Deferred Rendering (TBDR)*. URL: https://docs.imgtec.com/starter-guides/powervr-architecture/topics/tile-based-deferred-rendering.html 1.

[Ima21] IMAGINATION TECHNOLOGIES. *PowerVR Photon*. 2021. URL: https://www.imaginationtech.com/products/gpu/graphics-architecture/powervr-photon 2.

[Joh08] JOHANSSON, ROGER. *Genetic programming: Evolution of mona lisa*. Dec. 2008. URL: https://rogerjohansson.blog/2008/12/07/genetic-programming-evolution-of-mona-lisa/ 7.

[Khr23a] KHRONOS, VULKAN WORKING GROUP. *Vulkan® 1.3.250 - A Specification: Any-Hit Shaders*. 2023. URL: https://registry.khronos.org/vulkan/specs/1.3-extensions/html/vkspec.html#shaders-any-hit 1.

[Khr23b] KHRONOS, VULKAN WORKING GROUP. *Vulkan® 1.3.250 - A Specification: Opacity Micromap*. 2023. URL: https://registry.khronos.org/vulkan/specs/1.3-extensions/html/vkspec.html#VK_EXT_opacity_micromap 2, 7.

[MOB*21] MEISTER, DANIEL, OGAKI, SHINJI, BENTHIN, CARSTEN, et al. "A Survey on Bounding Volume Hierarchies for Ray Tracing". *Computer Graphics Forum* 40.2 (2021), 683–712. DOI: https://doi.org/10.1111/cgf.142662. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1111/cgf.142662. URL: https://onlinelibrary.wiley.com/doi/abs/10.1111/cgf.142662 2.

[NVI23] NVIDIA. *Ada GPU Architecture*. 2023. URL: https://images.nvidia.com/aem-dam/Solutions/geforce/ada/nvidia-ada-gpu-architecture.pdf 2.

[PJH16] PHARR, MATT, JAKOB, WENZEL, and HUMPHREYS, GREG. *Physically Based Rendering: From Theory to Implementation (3rd ed.)* 3rd. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., Nov. 2016, 1266. ISBN: 9780128006450 2.

[Seg98] SEGA. *Guppy/SET5 System Architecture*. 1998 2, 3.

[Smi98] SMITS, BRIAN. "Efficiency Issues for Ray Tracing". *J. Graph. Tools* 3.2 (1998), 1–14. ISSN: 1086-7651. DOI: 10.1080/10867651.1998.10487488. URL: https://doi.org/10.1080/10867651.1998.10487488 1, 7.

[SML11] SALVI, MARCO, MONTGOMERY, JEFFERSON, and LEFOHN, AARON. "Adaptive Transparency". *Proceedings of the ACM SIGGRAPH Symposium on High Performance Graphics*. HPG '11. Vancouver, British Columbia, Canada: Association for Computing Machinery, 2011, 119–126. ISBN: 9781450308960. DOI: 10.1145/2018323.2018342. URL: https://doi.org/10.1145/2018323.2018342 2.

[Wu92] WU, XIAOLIN. "Color Quantization by Dynamic Programming and Principal Analysis". *ACM Trans. Graph.* 11.4 (1992), 348–372. ISSN: 0730-0301. DOI: 10.1145/146443.146475. URL: https://doi.org/10.1145/146443.146475 6.

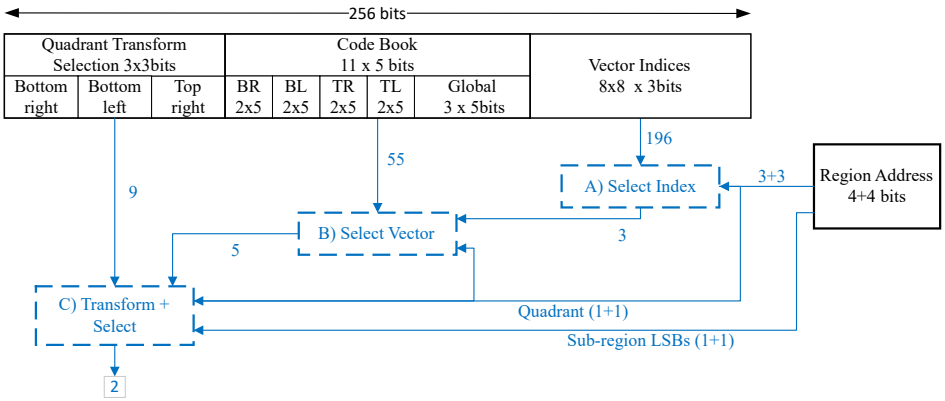

Figure 11: *1bit/region opacity map data and decode process: i) MSBs of input coordinates select the index, ii) which in conjunction with quadrant identifies the representative vector and iii) vector or coordinate LSBs are transformed and the final result selected.*

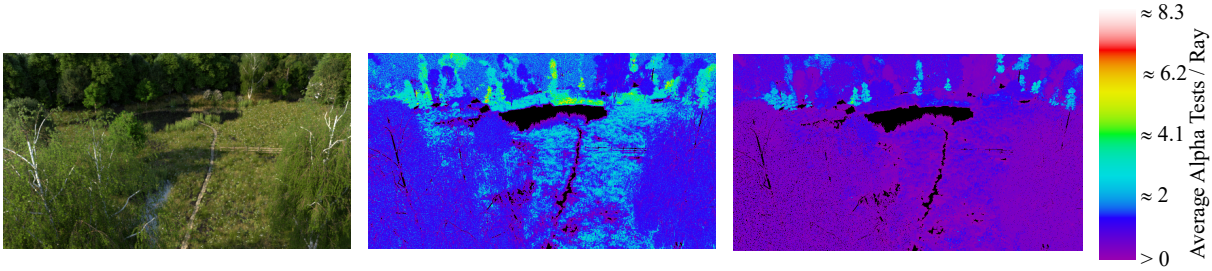

Figure 12: *PBRT Landscape model: Left: Render with 256 primary rays/pixel. Centre: Heat map of primary ray alpha tests - average 1.5/ray. Right: Heat map with (approximated) opacity maps - average 0.6/ray.*