COMPRESSED OPACITY MAPS FOR RAY TRACING

Simon Fenney & Alper Ozkan. *Imagination Technologies*†

OVERVIEW

We propose a method of compressing three-level opacity maps for accelerating alpha-tested triangles in ray-tracing (or rasterisation) that provides random access and low cost decompression. Each map, however, relates to an adjacent *pair of triangles* allowing simpler map look up, taking advantage of correlation across the shared edge, and matching *likely* underlying hardware primitive models.

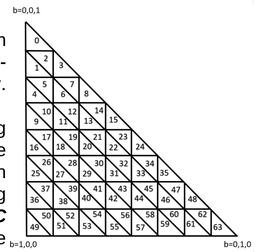
BACKGROUND

Contemporary Graphics APIs, such as DirectX® and Vulkan®, support the ray tracing of alpha-tested geometry by requiring any detected ray-triangle intersections to be subsequently refined by running a shader. Given that the initial intersection may have been performed by dedicated, pipelined hardware, interrupting that process by executing a shader is likely to result in a significant performance penalty.

A related problem occurs in rasterisation where alpha testing needs to be 'logically' done prior to the depth buffer update. This test can interfere with the efficiency of a TBDR or early Z-test system. [Howson 2014] proposes an 'Opacity State Map', storing 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***.

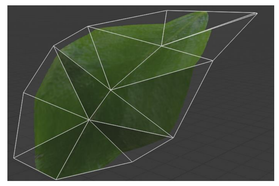
Although the Howson's "Opacity State Map" can be shared between multiple triangles, accessing it requires doing an upfront perspective-correct texture mapping operation, which is relatively costly.

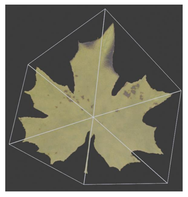
With respect to ray tracing, [Gruen et al 2020] instead propose storing a 'custom' per-triangle opacity map which is addressed via the barycentric coordinates determined by the ray intersection. Each triangular sub-region in the map has one of three states corresponding to Howson's ***O***, ***T***, and combined ***C.*** Akin to rasterisation, only the ***C*** state requires the shader execution. Gruen et al found their technique provided a 19~86% reduction in the expensive shader invocation with higher resolutions providing the greater savings.



Gruen et al's triangular Opacity Map

SAMPLE ALPHA-TESTED FOLIAGE





INITIAL OBSERVATIONS

Gruen et al's approach creates maps for individual triangles but we note:

1) In scenes such as "San Miguel", Unreal Engine's "Downtown West", or "Lumberyard-Bistro", plant models rarely appear to use singular triangles - leaves & flowers are usually modelled by at least a pair of contiguous primitives. Similarly, one would expect, say, a chain link fence or dilapidated window to be formed from rectangular sections.

2) At least two hardware ray tracing systems, Intel's (Gruen et al 2022) & Imagination's, store and/or test rays against triangle *pairs*. There are 3 reasons for doing so:

i) Bounding volume hierarchies tend to use AABBs. 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.

ii) When performing the ray-triangle-pair tests, the maths relating to the common edge can be shared in the evaluation.

iii) A triangle pair is more compact than 2 individual triangles, which can save memory bandwidth and increase cache utilisation.

†name.surname@imgtec.com

Gruen et al's encoding scheme uses 2 bits per sub-region but they note that 3 states could use 'less than 2 bits'. We assume they mean using $\approx \log_2 3$ bits per state, e.g., encoding 5 regions in every 8 bits. Although such an encoding would be optimal for equiprobable, random data, the opacity maps are rarely 'noise'.

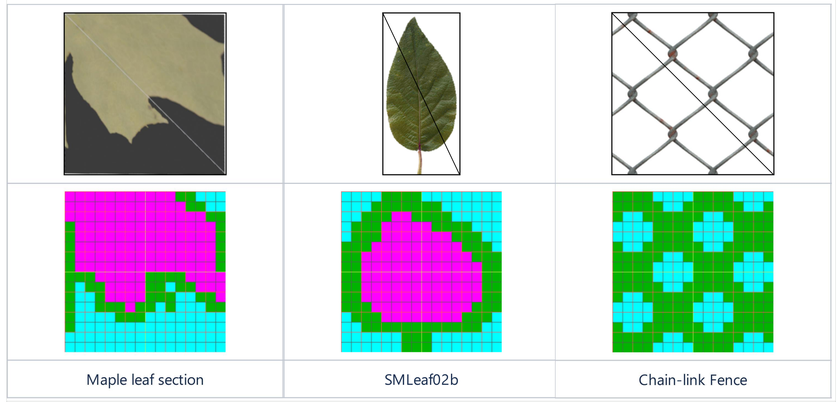
When accessing the source texture map, the sampling process in the shader is likely to use, at least, bilinear filtering. This implies that sampled alpha values will be continuous across the triangle and thus it should be impossible, in practice, to change instantly from fully opaque to fully transparent. (Note that, even if point-sampling were used in the shader, due to the likely difficulty of matching floating-point results between map creation and sampling, a conservative approach is advised)

Given memory latency costs, it would be advantageous for the opacity map to be read at the same time as the triangle vertex data, and be preferably of comparable size.

OUR PROPOSAL

We have investigated storing a *square* opacity map (e.g. 16x16) for each pair of adjacent triangles and using a lossy compression technique based on Vector Quantisation to keep the storage costs to around 1 bit per region for a 16x16 . This would be less than a typical cache line in size.

EXAMPLE 16*16 OPACITY MAPS FOR TRIANGLE PAIRS



In these examples, cyan represents 'fully transparent', magenta, 'fully opaque', and green, "check texture". Sub regions are represented as squares rather than triangles.

OUR METHOD

Many compression methods, e.g. Huffman, do not permit the random decode of an arbitrary value. One scheme that does, however, is Vector Quantisation. VQ has been used in graphics for compression of texture data ([Beers et al 1996], Dreamcast [Sega 1998]), but fell out of favour because of the expense of overcoming the memory latency of the codebook access. If, however, both the indices and codebook are loaded together in 'local memory', this is no longer an issue.

For a 16x16 resolution with a pre-determined budget of 256 bits, we subdivide the map into 64, 2x2 'vectors'. Initially, it may appear there are 3^4 such vectors but, by the continuity assumption, it is impossible for ***O*** and ***T*** states to co-exist in the same vector. This reduces the possibilities to just 31, suggesting the following 5-bit encoding for each vector stored in the VQ codebook:

Palette Mode	Top Left	Top Right	Bottom Left	Bottom Right

The 'Palette Mode' indicates if the per-region modes are chosen from either the palette, {***O***,***C***}, or {***T***,***C***}, and the remaining fields are 1-bit indices into the selected palette.

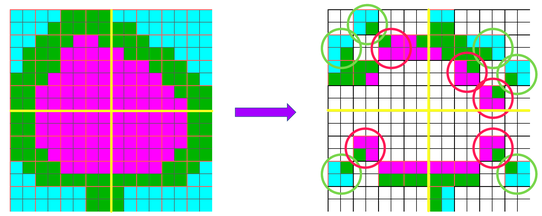
The VQ encoding also requires 64 indices. To fit the 256-bit budget, each index must therefore be less than 4 bits in size. As 2-bit indices are too limiting, 3 bit indices are used. This leaves a budget of 64 bits for codebook and sundry data.

Example data demonstrated that the 2x2 vectors that are uniformly ***O***, ***T***, or ***C*** are particularly common – each typically occurring with probabilities of around 11~29%. Rather than store these explicitly in a code book, it seemed prudent to dedicate 3 indices, {0,1,2}, to implicitly represent {***T***, ***C***, ***O***} respectively, leaving 5 indices to access codebook data.

UTILISING SYMMETRY

From observation of the test data, it became apparent that the natural symmetries, both reflections and rotations, of the data being represented could be utilised to improve compression.

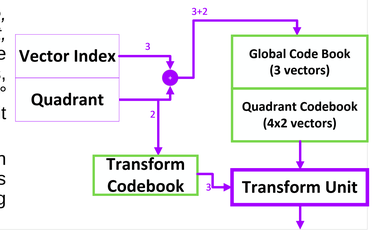
To illustrate, we take the *smleaf02* example, remove the 'uniform' vectors, and divide the map into quadrants:



Using the Top Left quadrant as a reference, we note that the vector circled in red can also be seen twice in the Top Right quadrant but rotated 90° clockwise, and again in the Bottom Right and Bottom Left except rotated by 180° and 270° respectively. Identical behaviour can be seen with the vectors circled in green. In other samples, reflections about the X axis, Y axis, and the lines X=Y, X=-Y were also observed.

To utilise this, the compression scheme 'stores' a transformation 'matrix' for the ***TR***, ***BL***, and ***BR*** quadrants. The ***TL*** quadrant's transform is always the identity matrix. When a stored codebook vector is accessed, the associated quadrant's transformation is applied.

A set of 8 transforms, {*Identity, Rotate 90° Anticlockwise, Reflect Top/Bottom, Reflect X=-Y, Reflect Left/Right, Reflect X=Y, Rotate 180°, Rotate 270°*} can be constructed by chaining 3 optional base transforms, [*Reflect Left/Right, Reflect Top/Bottom, Rotate 90° anticlockwise*]. In hardware this requires just 24 1-bit MUX units and three 3-bit fields in the compressed data.



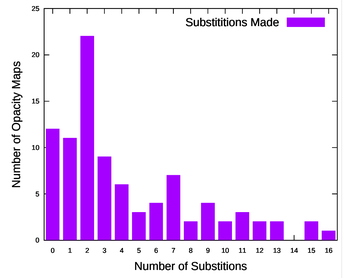
The remaining 55 bits store a total of 11, 5-bit vectors in a code book. Three are shared between all quadrants and are accessed via indices {3,4,5}. The remaining indices {6,7} access per-quadrant codebook entries.

COMPRESSION ALGORITHM

With any fixed-rate compression scheme, there must exist source data that cannot compress losslessly - opacity maps are no different. We note that it is always safe to replace an ***O*** or a ***T*** state in the source map with a ***C*** in the compressed variant. Of course such a substitution comes at performance penalty as it results in the expensive shader invocation that the opacity map is aiming to reduce. It is felt that, assuming a fixed storage budget, the higher resolution achieved by compression, and the subsequent reduced probability of ***C*** states, will offset the occasional substitution.

An initial, naïve compression algorithm that 'brute-forces' the 2^9 transform combinations followed by a greedy algorithm to choose global and local candidates, was used to evaluate the scheme. Testing with a set of 92 (denoised) alpha textures, each directly mapped to two triangles, resulted in 65% of the set compressing with 4 or fewer substitutions.

The test that scored 16 – a 'San Miguel' scene texture of a branch with multiple leaves – really should be modelled with more than just 2 triangles.



Though not presented, the authors are also investigating 0.5bit/region compression using a 32x32 map.

REFERENCES

Andrew **Beers**, Maneesh Agrawala, and Navin Chaddha, 1996
"Rendering from Compressed Textures"
Proceedings of the 23rd Annual Conference on Computer Graphics and Interactive Techniques

*Holger **Gruen**, Carsten Benthin, and Sven Woop, 2020*
"Sub-Triangle Opacity Masks for Faster Ray Tracing of Transparent Objects",
Proceedings of the ACM on Computer Graphics and Interactive Techniques, Volume 3, Issue 2.

*Holger **Gruen**, Joshua Barzack, 2022*
"A Quick Guide to Intel's Ray Tracing Architecture", GDC 2022
https://www.youtube.com/watch?v=SA1yvWs3lHU

John **Howson**, 2014
"Opacity testing for processing primitives in a 3D graphics processing system"
GB2538856

Sega®, 1998
"Guppy/SET5 System Architecture"

HPG 2022