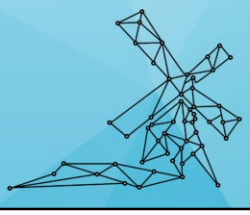
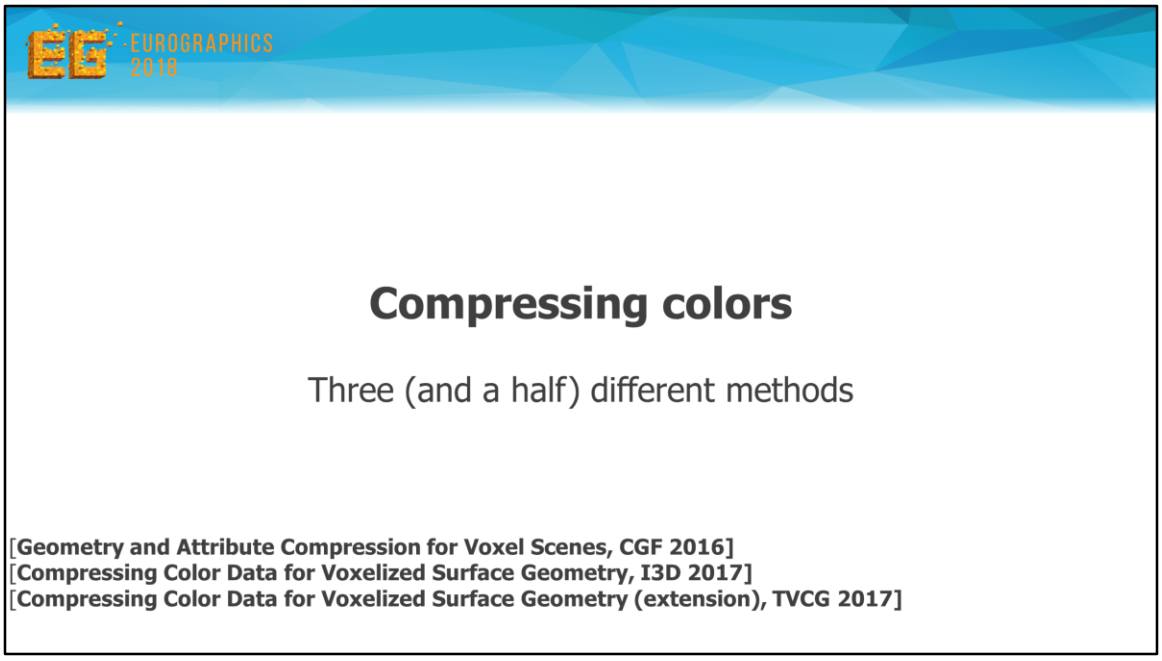
Compressing voxelized surface colors

Dan Dolonius[1]

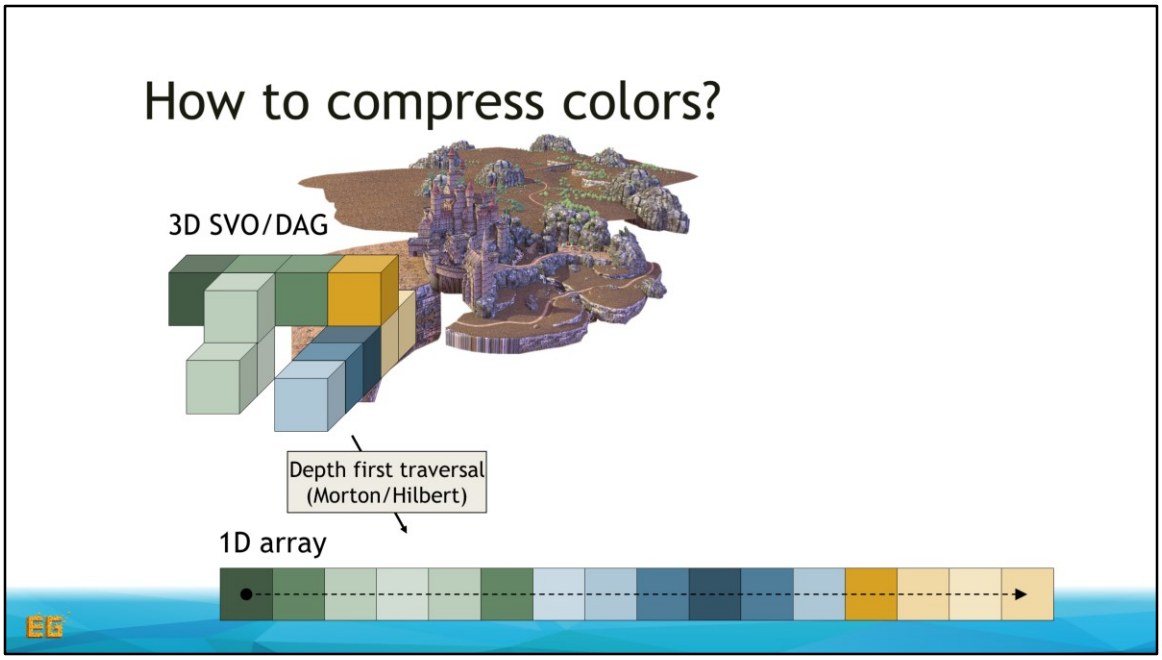
[1] Chalmers University of Technology, Sweden



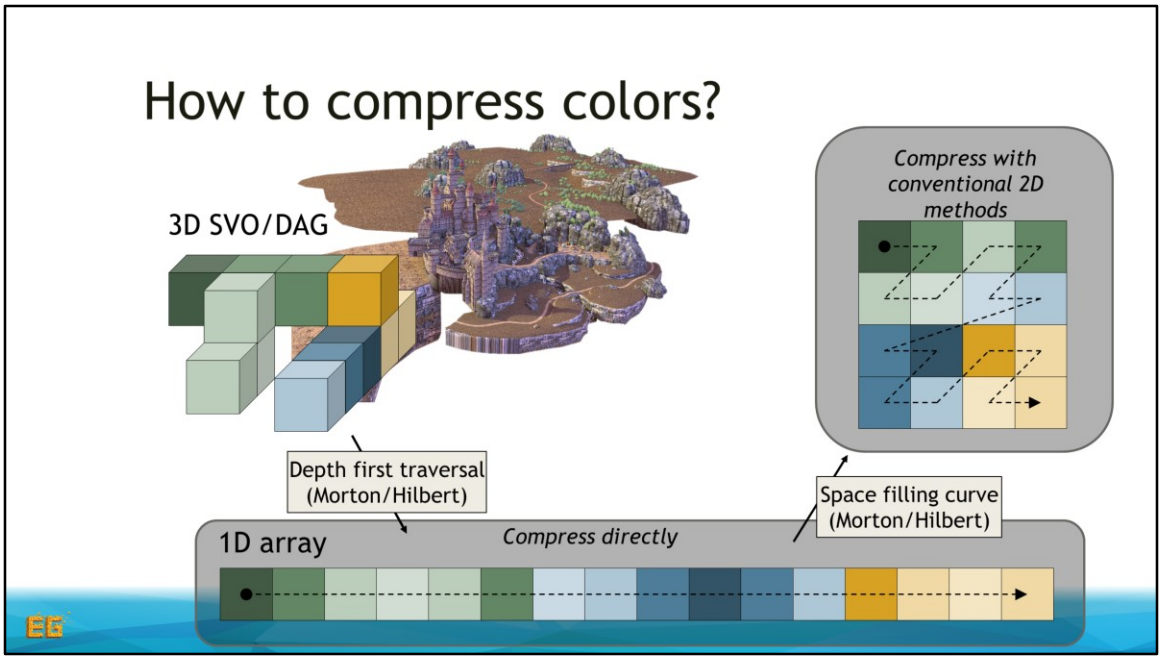


So my name is Dan Dolonius, and I'm a PhD. Student of Ulf.
I will now talk about how we can compress the colors which Ulf so kindly mapped for us.

I will present three different ways of compression, as well as an improvement which can be made to one of those methods.



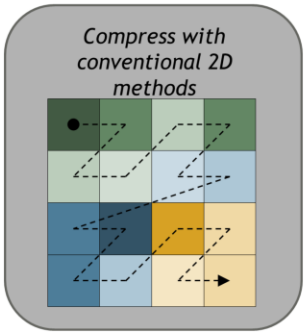
So… Here we have the color array from a Morton traversal Ulf spoke of earlier.



We could compress this array directly, or
<click>
We could again use a space-filling curve to map it to a 2D texture.
<click>
Then we can utilize conventional compression formats for images and textures.
So let's look at this case first.

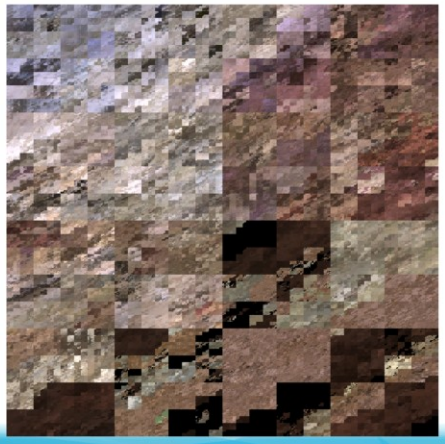
2D Compression

- Hardware accelerated
  - *ASTC / BC7 / BC1*
  - *Decent compression (3%-33% / 33% / 16%)*
  - *Decompression for free*
- Offline
  - *JPG / JPG2K / PNG*
  - *Great compression (5%-20%)*
  - *Suitable for storage / streaming*



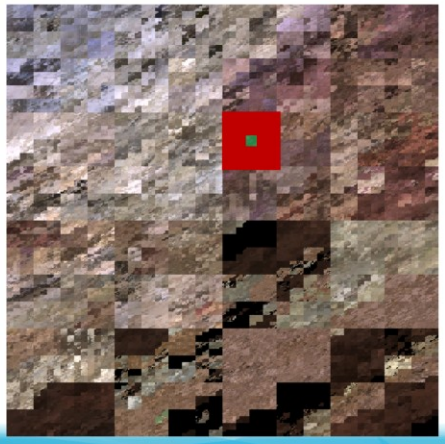
There are many formats out there, and this is a few we have tried.
Real time formats such as ASTC, BC7 and BC1 which gave a high quality result at decent compression ratios.
<click>
And also offline formats such as JPEG, JP2K and PNG which offers great compression at the cost of less quality, which might be suitable to use for storage or streaming.
I also want to make a quick note here, that PNG in itself is not a lossy format, however it can be used as a lossy format by first quantizing the colors, with tools such as pngquant for a lossy compression and optimization for the PNG format.

Example of actual data

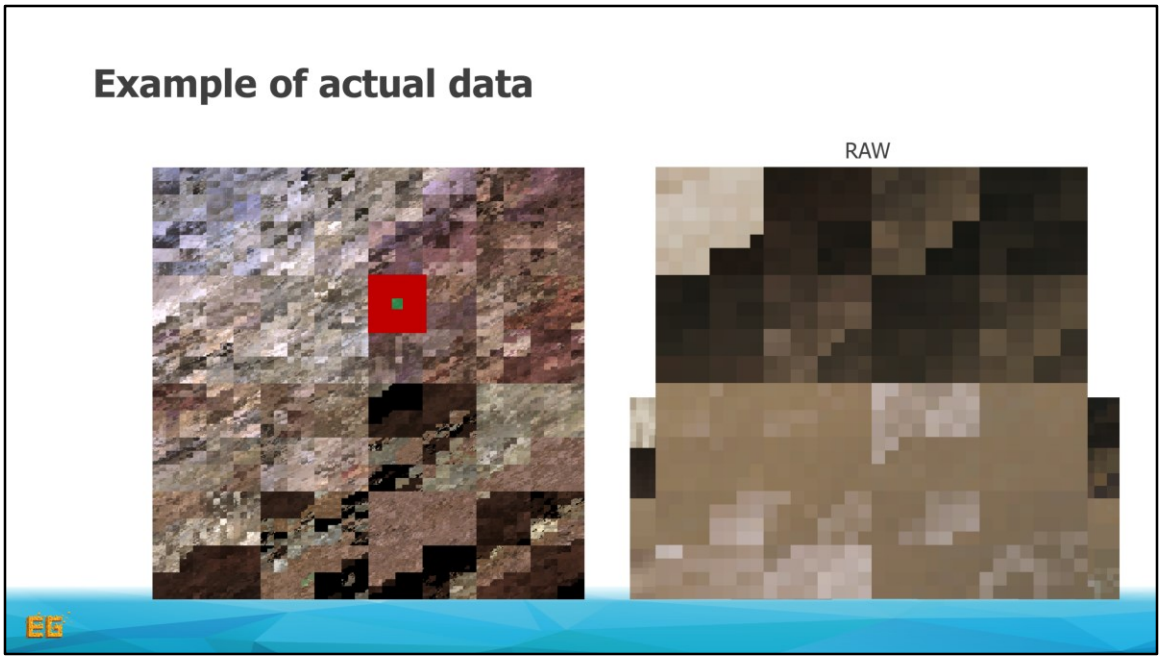


So here is a subset of an image generated from the epic citadel scene, by using morton ordering from 3D to 1D, and then 1D to 2D.
I want to show you why this is a good candidate for compression, and also why this, in some cases, can generate artifacts, which we will highlight later

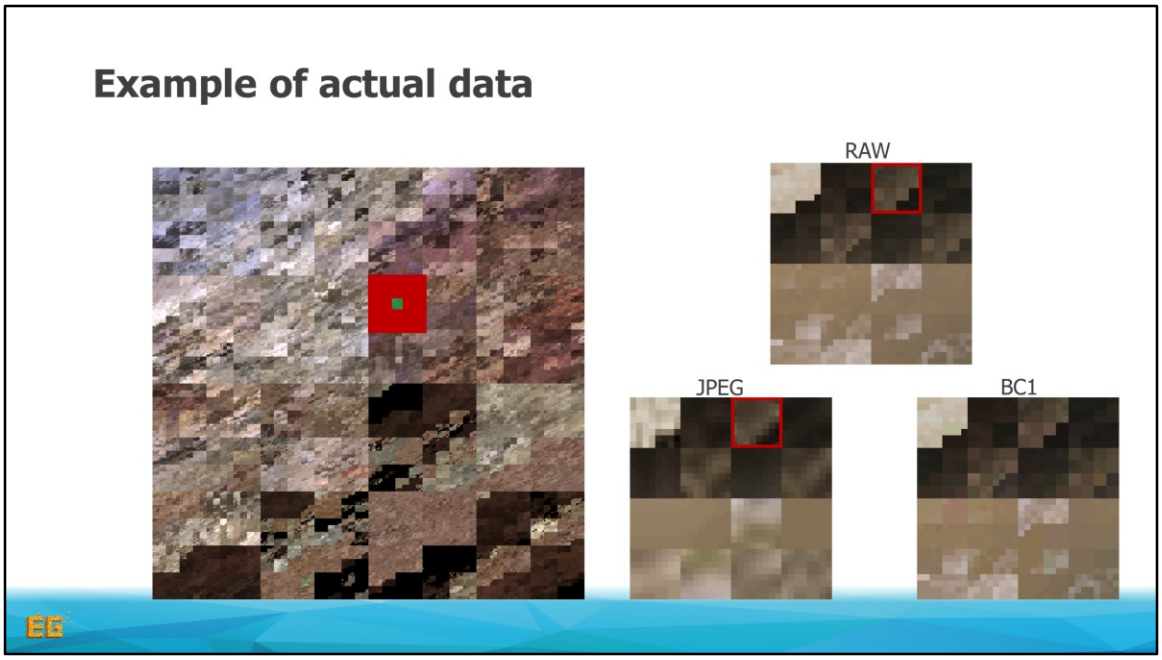
Example of actual data



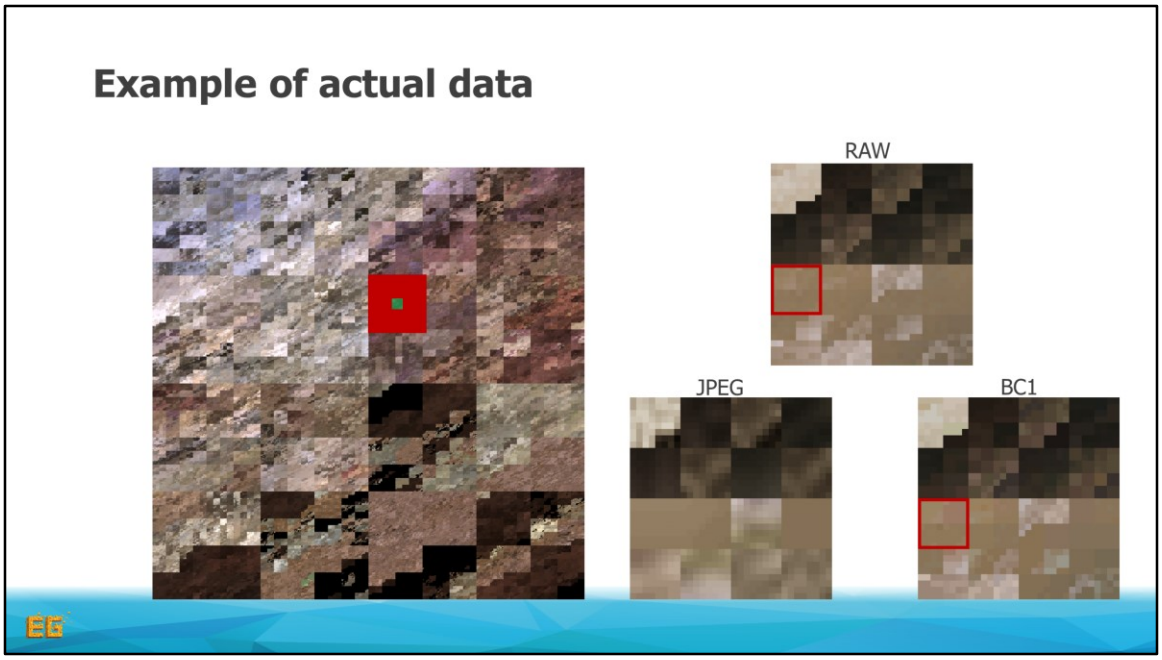
If we zoom in to this region



We see that we do have some coherency.
But we also see that we have some sharp gradients.
This is when we jump from one surface to another in 3D due to our space filling curve.
This is also where things can get a bit problematic.
<click>
So here I show a agressively compressed jpeg as well as the bc1 encoding.
(As you probably know jpeg do a discrete cosine compression in 8x8 blocks.)



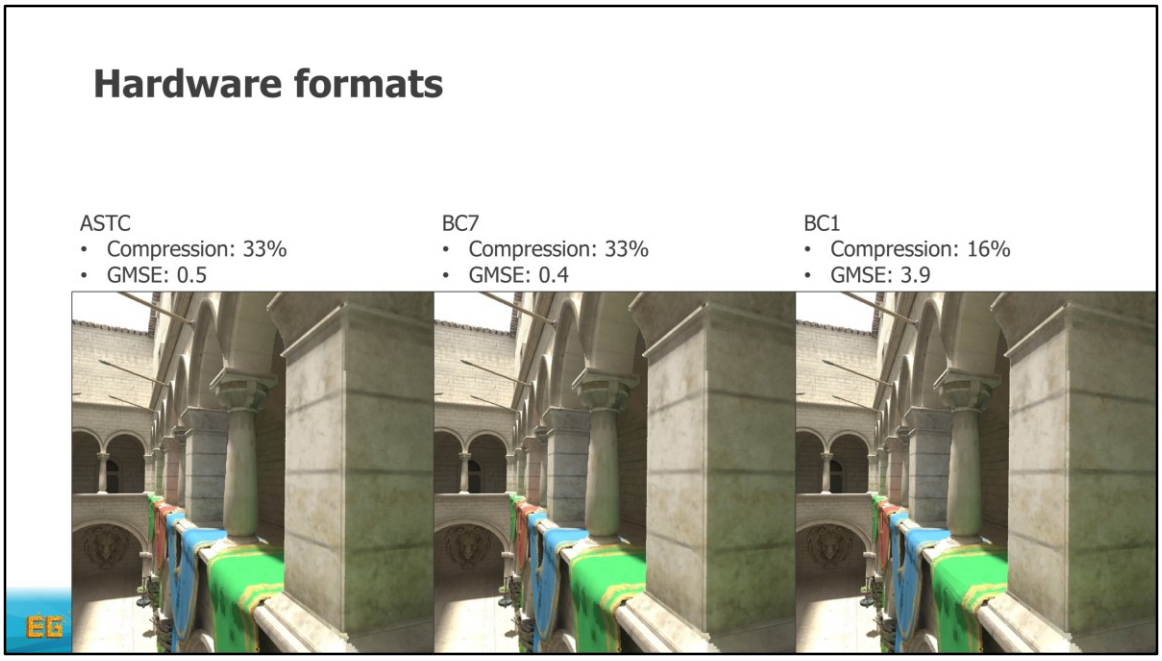
, and you can see that in this highlighted region the sharp gradient between the brown and black is significantly blurred.
This is bad, because now the brown and black surfaces will bleed over to each other. And as I said before, these two are not neccesarily close to each other in 3D space, and so we will get some voxels wich end up having the wrong color.



Now, bc1 also compresses in blocks.
In this case it's 4x4 blocks, where each block has two colors and some weights in order to interpolate between them.
As you *might* see here this region has some colors which get the wrong hue.

This problem is much less present with bc7 and astc as they partition each block in to one or more different sets of colors and is thus much more likely to preserve the problematic sharp gradients.

So now we move on to see some actual results!



Here is three images form a ray traced DAG **in real time.**
We start by looking at the hardware formats for the Sponza scene where each voxel has baked illumination, so the colors should be fairly unique.
The measures we present here is the compression ratio as well as the global mse, meaning the mse of all voxels.
So we have ASTC, as well as, BC7, which compresses to 33% of the original size with a very good mse.
In fact you can hardly see any difference form the original data.
We also have bc1 at half the size but at the cost of a higher mse.
Looking at this it might not be obvious that BC1 has some quality issues, so
<click>

Hardware formats

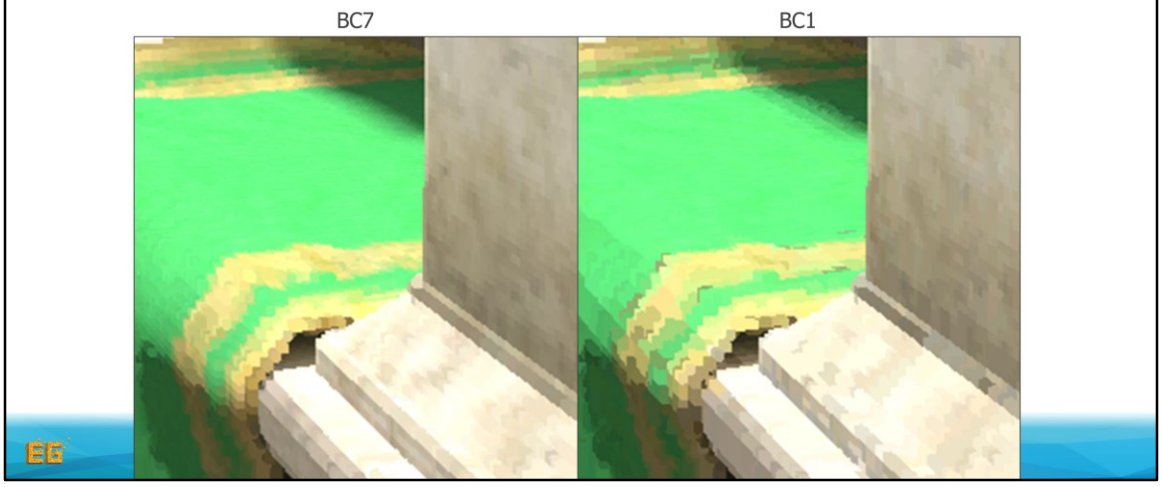
ASTC
- Compression: 33%
- GMSE: 0.5

BC7
- Compression: 33%
- GMSE: 0.4

BC1
- Compression: 16%
- GMSE: 3.9

If we look closer at this region

Hardware formats



We clearly see that some of the brown colors has bled over to the green drape which introduces some obvious artifacts.
But still, depending on your needs this might be an acceptable tradeoff.

Offline formats

JPEG
- Compression: 5%
- GMSE: 8.3

JP2K
- Compression: 5%
- GMSE: 11.1

PNG
- Compression: 12%
- GMSE: 2.4

Now to compare the the offline formats, we now look at the epic citadel, again with per voxel baked illumination.
Here we chose to be quite aggressive with the compression to show that we are able to reduce the size considerably, at the expense of visual quality

Offline formats

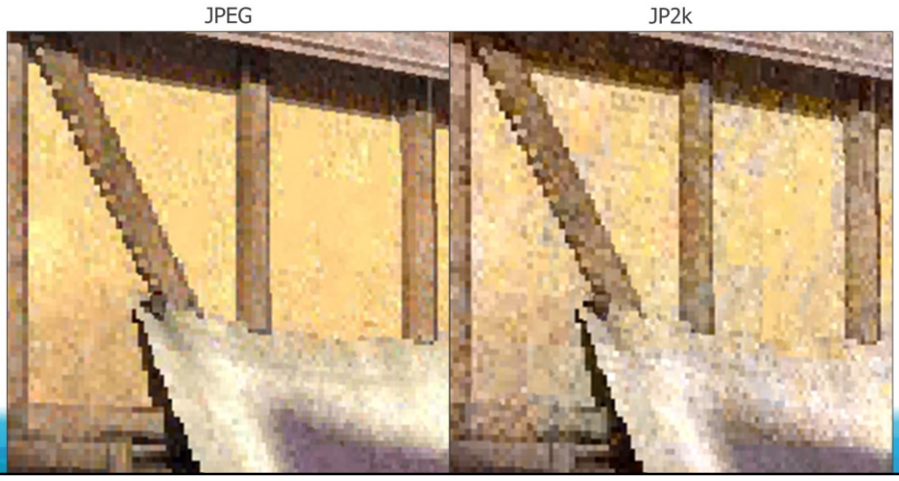
JPEG
- Compression: 5%
- GMSE: 8.3

JP2K
- Compression: 5%
- GMSE: 11.1

PNG
- Compression: 12%
- GMSE: 2.4

We start by comparing jpeg and jp2k by zooming in to this region

Offline formats



As you can see both formats are quite noisy due to the color bleeding in the 2D texture.

We were a bit disappointed by jp2k as we expected it to preserve the sharp gradients sort of “adaptively” du to the hirearchcal nature of wavelets.. but it turns out while jpeg might blur more compared to jp2k, this blur is still restricted to 8x8 blocks while the blur generated from jp2k might span over larger distances and thus introducing more noise. We did some quick tests by doing a naïve wavelet compression using haar wavelets instead of the smooth one used in jp2k, and it seemed more promising, but unfortunately we did not have time to pursue this idea.

Offline formats

JPEG
- Compression: 5%
- GMSE: 8.3

JP2K
- Compression: 5%
- GMSE: 11.1

PNG
- Compression: 12%
- GMSE: 2.4

If we now instead focus on the png format, we note that, while it does seem to have less noise, we have some other troublesome artifacts instead

Offline formats

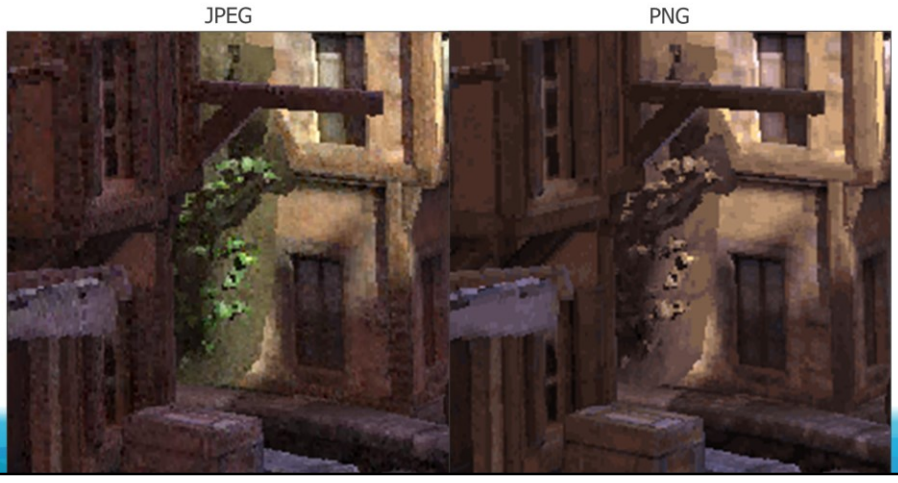
JPEG
- Compression: 5%
- GMSE: 8.3

JP2K
- Compression: 5%
- GMSE: 11.1

PNG
- Compression: 12%
- GMSE: 2.4

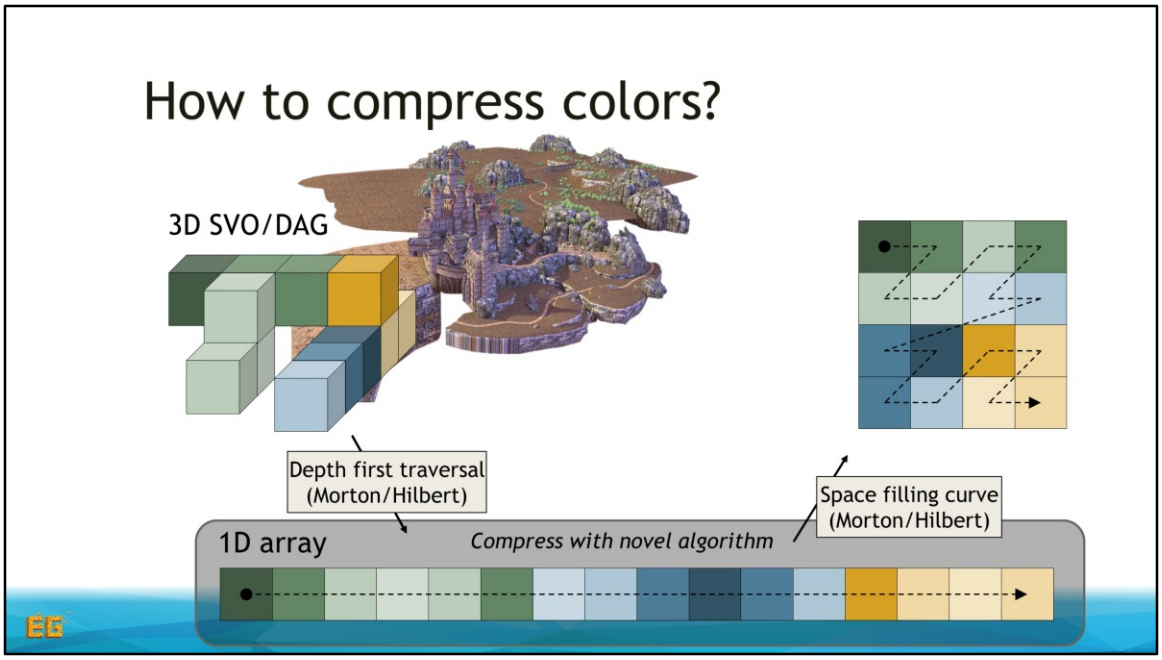
So we zoom into this region

Offline formats

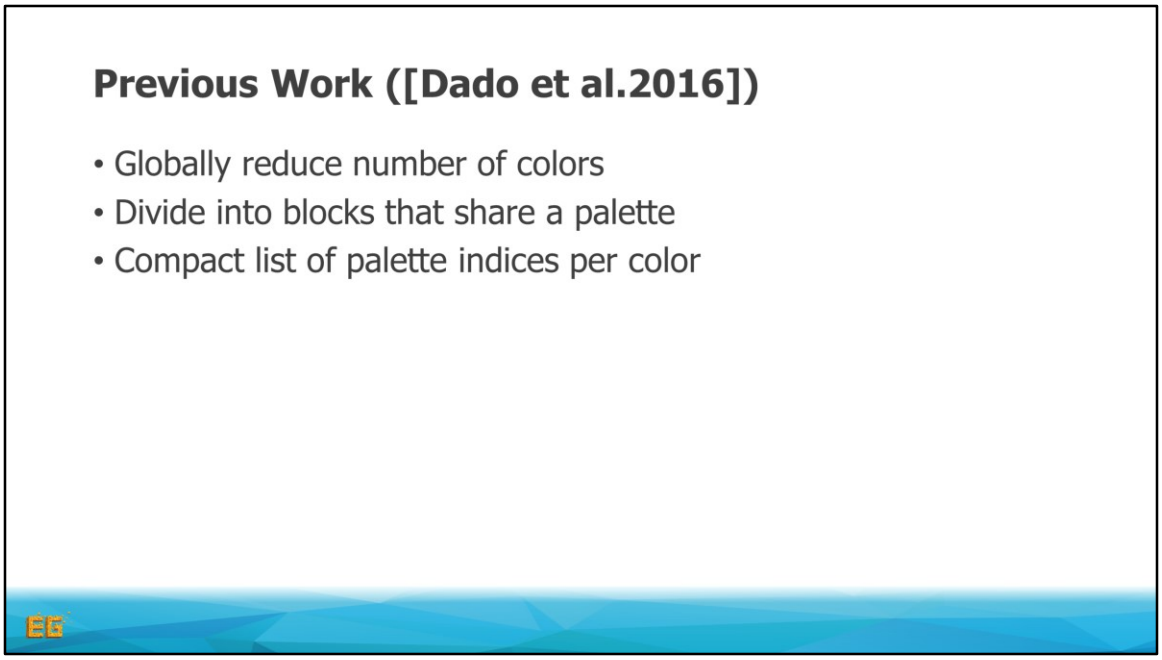


And see that it is indeed less noisy, but this green vine got a completely different hue, as well as you can see some palettization going on.
This is not that strange as png compression utilizes a quantized pallete.

With this said I will now move on to two other different approaches of compressing colors



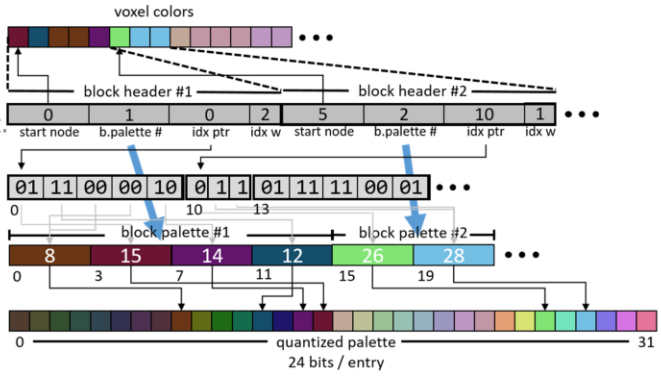
I we look at this slide again,
<click>
we will now focus on this 1D array instead
<click>



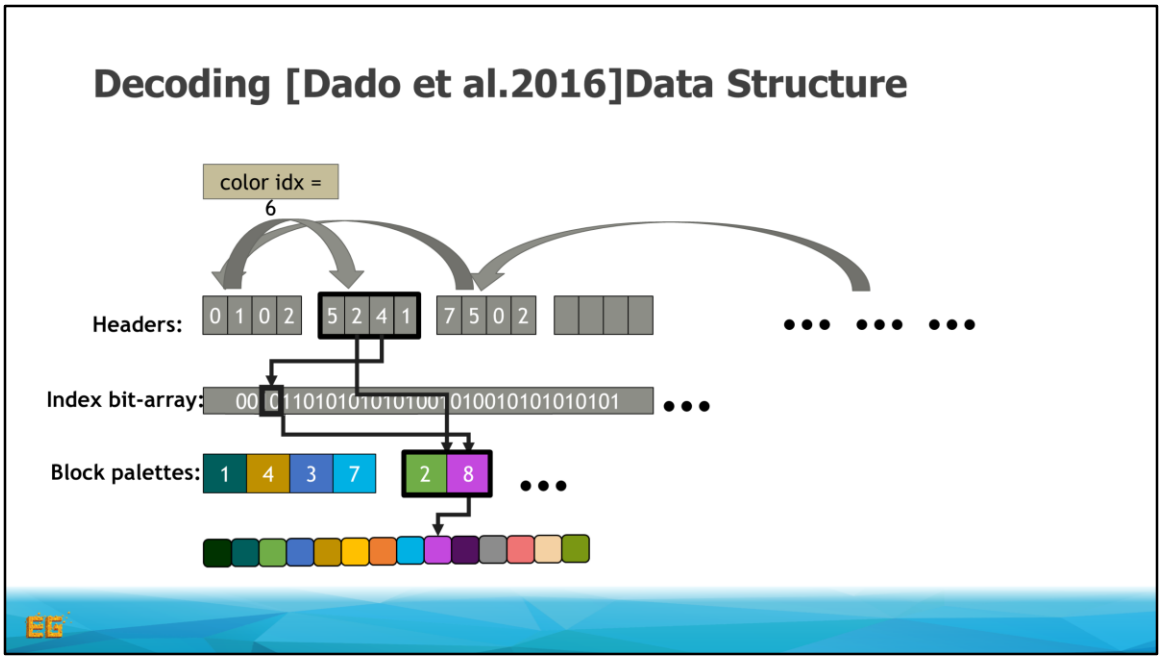
- Let's start by looking at the method proposed by Dado et al.
- Their approach is to first globally reduce the number of colors to, for example, 4K, through a clustering in color space,
- Then they divide all colors into blocks that can share a small palette.
- And then for each color within that block they store an index into that palette.

Previous Work ([Dado et al.2016])

- Block components:
  - Header
    - Start index in color array
    - Block-palette index
    - Bit offset sub-palette index.
    - Bits per index
  - Sub-palette ids
    - Up to 4 bits
  - Block palette
    - Indices to quantized global palette

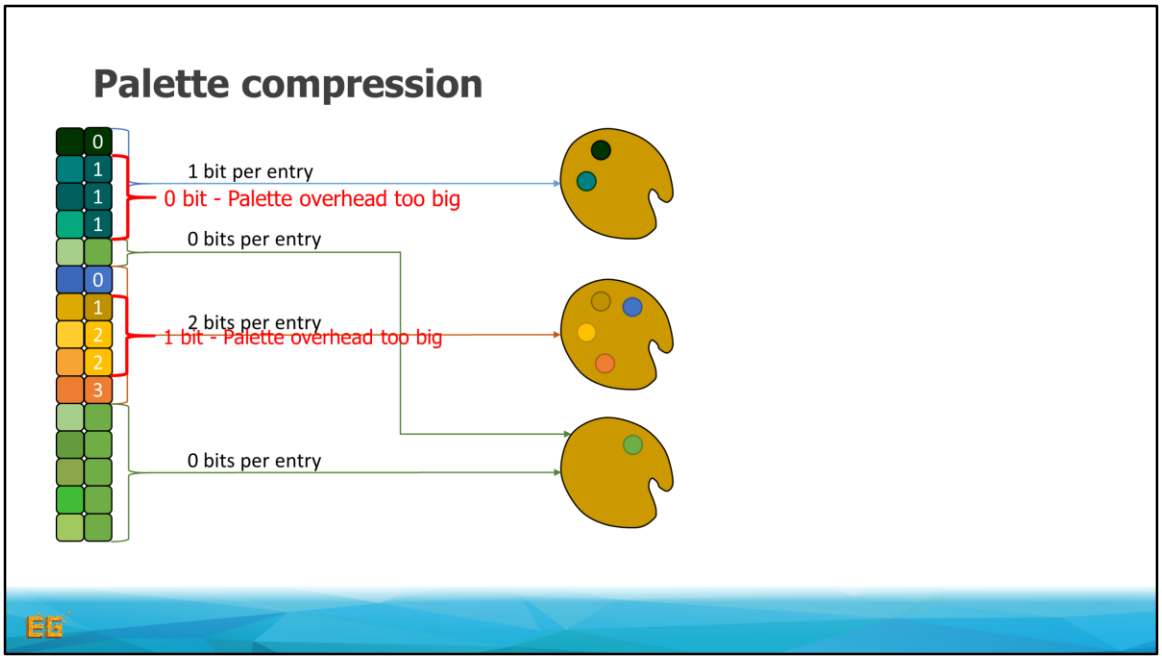


- Their data structure looks like this
- We have an array of headers containing the start index to the color array, a block-palette index, a start index to a bit array used for the block-pallete, and finally a value specifying the number of bits used for the block-pallete indexes, as it would be wasteful to store indices to a pallete with only one color.
- The block palettes contains indices to a global quantized pallete.
- In this next slide I will try to explain how this data structure is used.



So, say that our color index was 6
We first do a binary search on the start indexes of the headers to find the header describing that set of colors
<click> … <click>
We then use the block palette index to find the block palette for this color index
<click>
We then use the bit index to find the position in the bit array for the block-palette indices.
<click>
We use the bit width information and the offset of the color index from the start color index to find the desired index in the block palette.
<click>
And finally with this index we do a look up in the global palette

So let's move on to how we compress the palettes

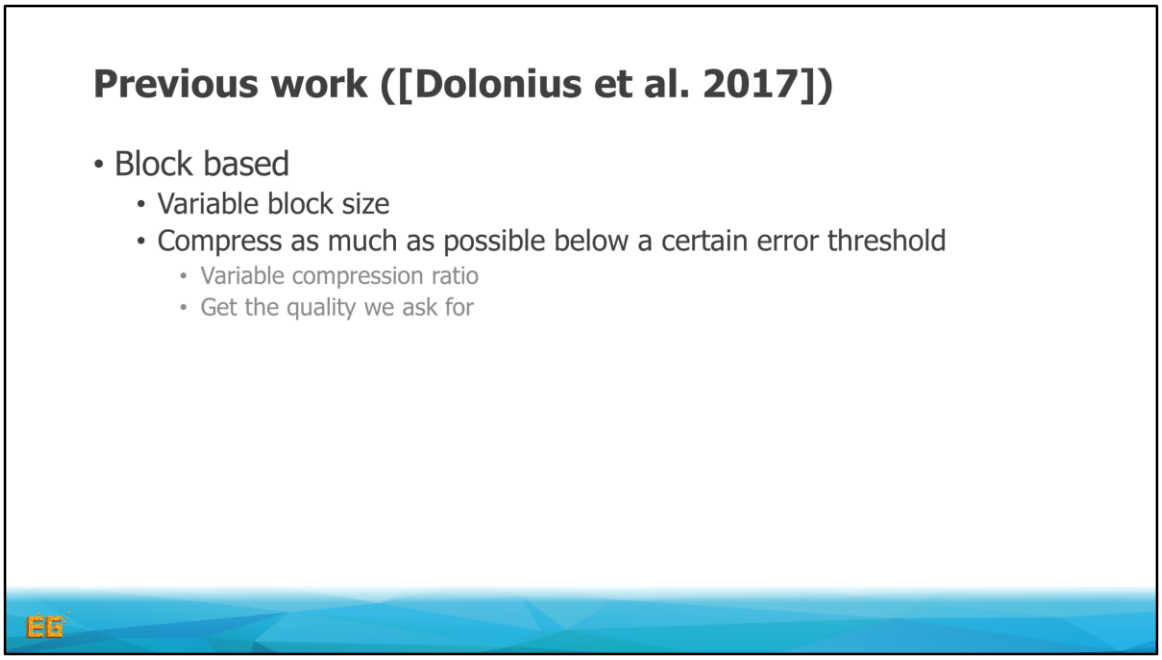


To encode the data structure we first quantize the colors
<click>
We then start with zero bits per entry, and try to find the largest ranges of colors which can be represented by a palette.
<click>
If the memory overhead is too big
<click>
We increase the number of bits per entry, and try to find a palette as before

We repeat this until we have processed all number of bits per entry.

Finally, we resolve palettes for the remaining colors.
The green color here, for example, is directed to the existing palette.

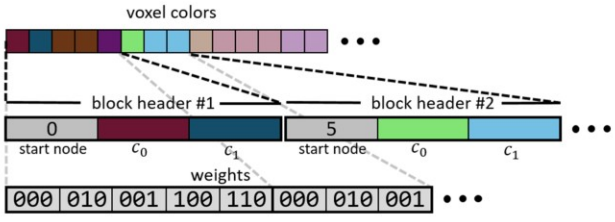
This concludes the section about this particular format, and I will now explain a different approach.



This is another format by Dolonius et al. which is inspired by the block encodings, such as the BC family amd ASTC.
It is block based, but with a variable block size.
This means that we can compress as much as possible under a certain error threshold, which in this case is the largest distance between an approximated color and the original.
So, instead of choosing a compression ratio, we choose the largest error we are willing to tolerate and can then try to compress as much as possible as long we stay under the error.

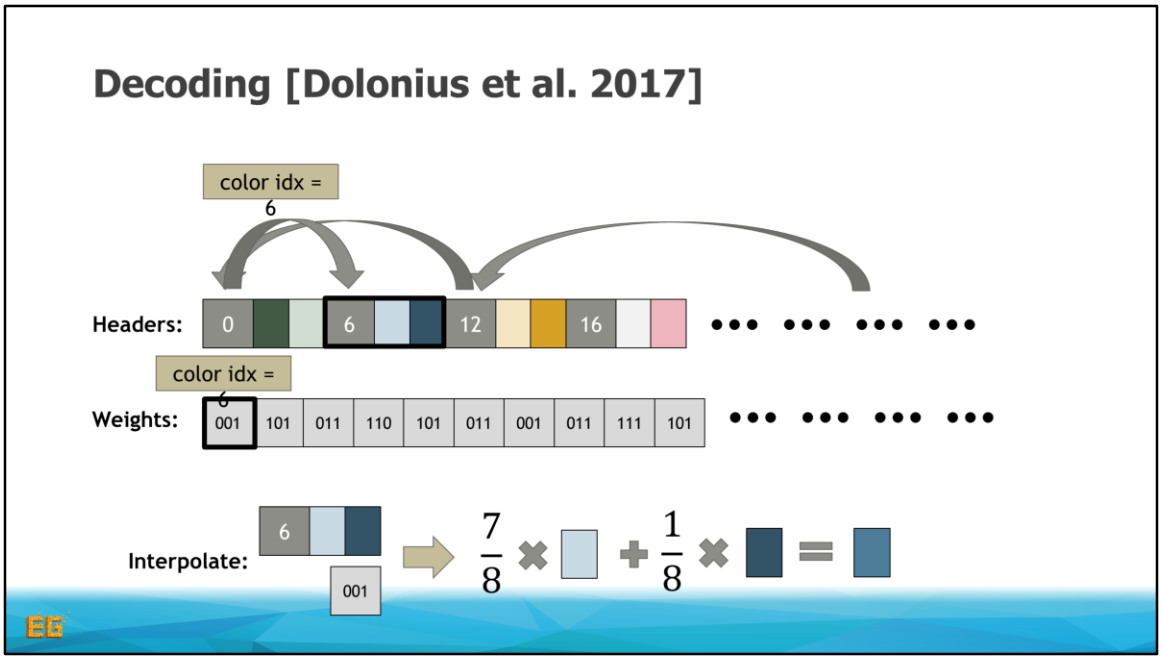
Previous work ([Dolonius et al. 2017])

- Block components
  - Header
    - Start index in color array
    - Two colors
  - Weights
    - ~3 bits per weight
- Interpolate between colors

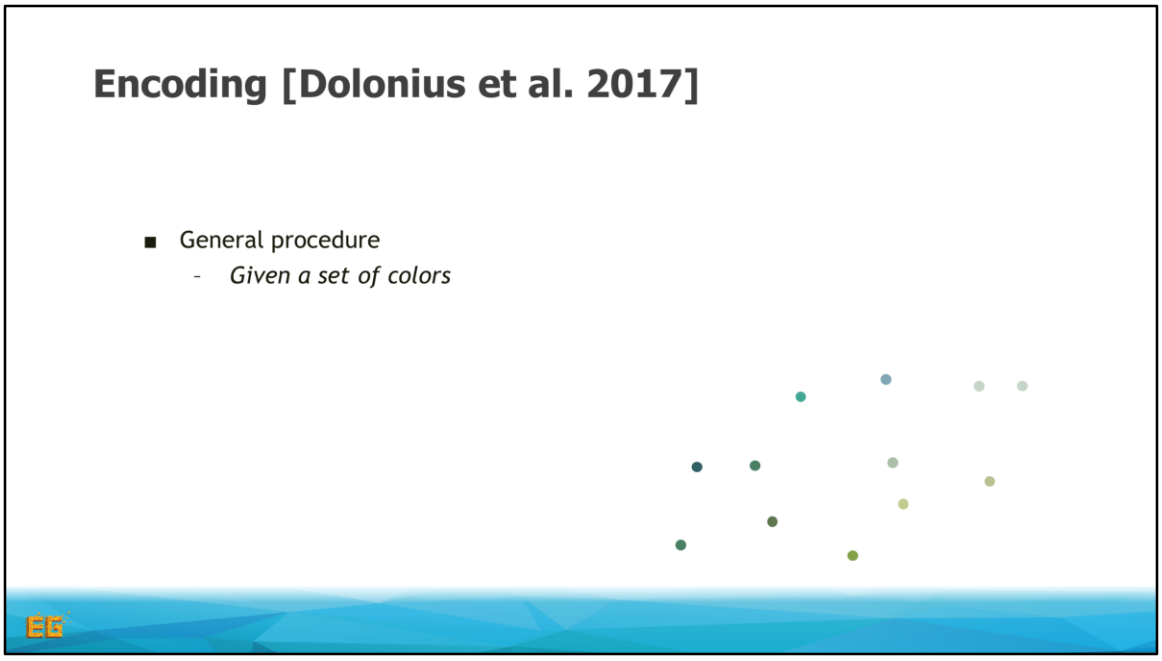


The data structure we use looks like this.
We have an array of headers containing the start index to the color array and two colors for interpolation
We also have a separate array of individual weights which we use to interpolate between the two colors in a block

So lets move on to how we can find our color through this structure



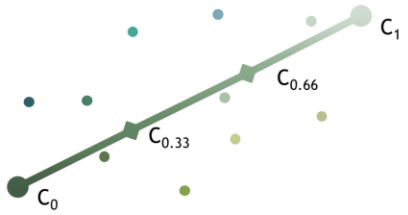
As before, we do a binary search to find the correct block header.
<click> … <click>
We then use our index to find which weight we have for that color by a direct lookup in the array of weights
<click>
The final color then is a simple linear interpolation between the two colors in the header
<click>
And that's it.. So let's move on to the encoding.



We want to end up with a set of blocks where each block will contain a set of approximated colors.
So if we would make a block of these colors.

Encoding [Dolonius et al. 2017]

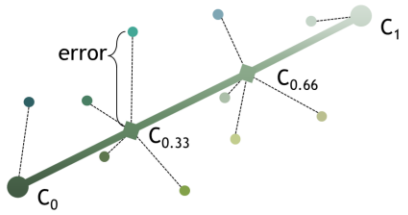
- General procedure
  - *Given a set of colors*
  - *Fit a line segment*
  - *Store end points and index and in header*



We fit a line segment, where we will store the end points and start index in the header

Encoding [Dolonius et al. 2017]

- General procedure
  - *Given a set of colors*
  - *Fit a line segment*
  - *Store end points and index and in header*
  - *Calculate weights*



We will also calculate and store the weights to approximate the colors along this line. The block is considered valid if the error is below the threshold I described before, which is the largest distance to the interpolated color

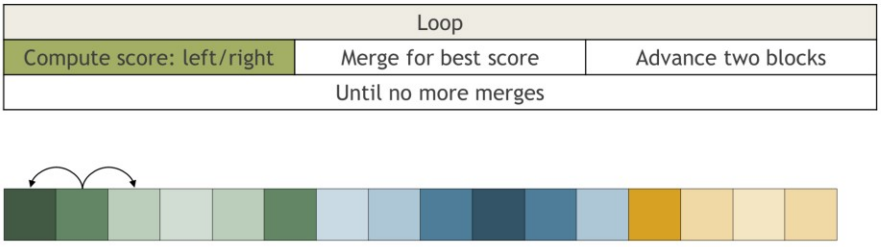
Since the weights are constant, the number of headers is what we want minimize. In other words, we want to construct as large blocks as possible.

Encoding [Dolonius et al. 2017]

Loop		
Compute score: left/right	Merge for best score	Advance two blocks
Until no more merges		

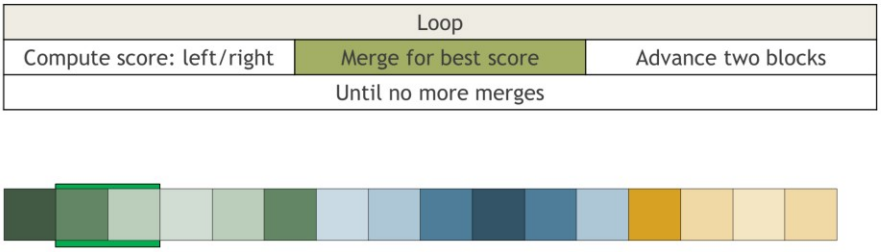
We do this by startig of with each color being its own block

Encoding [Dolonius et al. 2017]



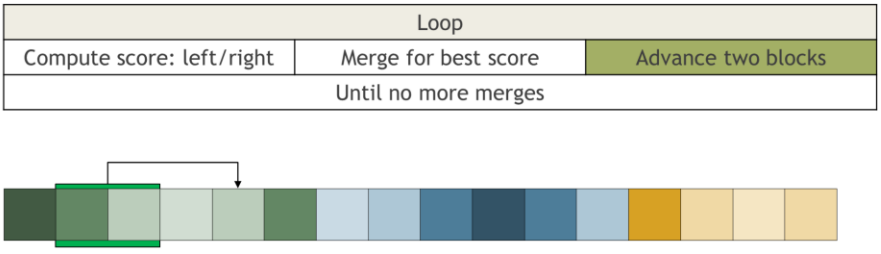
We then start at the second block and try to merge that block to the left or the right
We then calculate a score for both those merges.

Encoding [Dolonius et al. 2017]



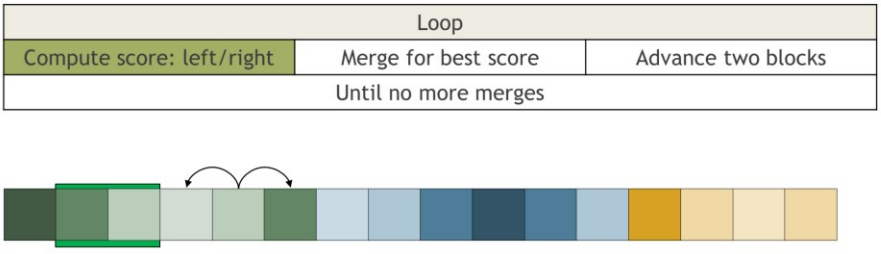
And merge with the block which gave the best score

Encoding [Dolonius et al. 2017]



We then advance two steps

Encoding [Dolonius et al. 2017]

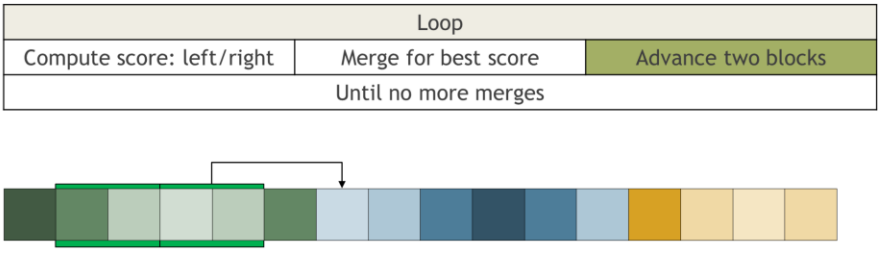


And again look at the left and right block, and do the same thing until we have processed all blocks

Encoding [Dolonius et al. 2017]

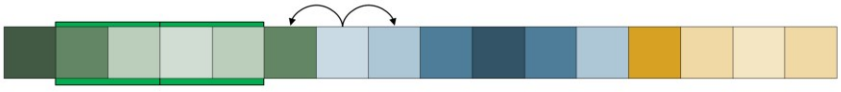
Loop		
Compute score: left/right	Merge for best score	Advance two blocks
Until no more merges		

Encoding [Dolonius et al. 2017]



Encoding [Dolonius et al. 2017]

Loop		
Compute score: left/right	Merge for best score	Advance two blocks
Until no more merges		



Encoding [Dolonius et al. 2017]

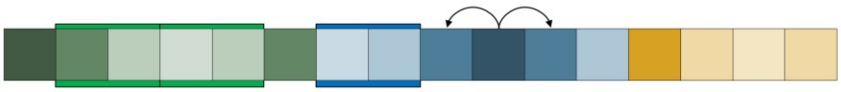
Loop		
Compute score: left/right	Merge for best score	Advance two blocks
Until no more merges		

Encoding [Dolonius et al. 2017]

Loop		
Compute score: left/right	Merge for best score	Advance two blocks
Until no more merges		

Encoding [Dolonius et al. 2017]

Loop		
Compute score: left/right	Merge for best score	Advance two blocks
Until no more merges		

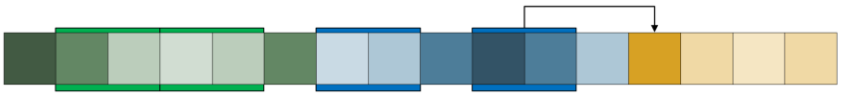


Encoding [Dolonius et al. 2017]

Loop		
Compute score: left/right	Merge for best score	Advance two blocks
Until no more merges		

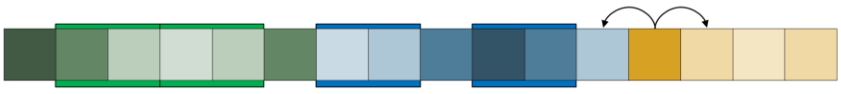
Encoding [Dolonius et al. 2017]

Loop		
Compute score: left/right	Merge for best score	Advance two blocks
Until no more merges		



Encoding [Dolonius et al. 2017]

Loop		
Compute score: left/right	Merge for best score	Advance two blocks
Until no more merges		

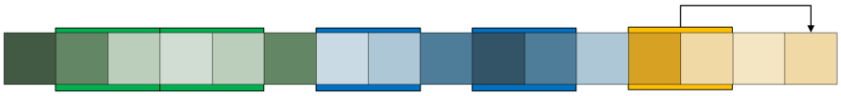


Encoding [Dolonius et al. 2017]

Loop		
Compute score: left/right	Merge for best score	Advance two blocks
Until no more merges		

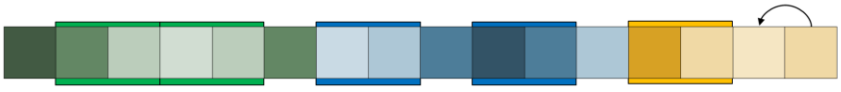
Encoding [Dolonius et al. 2017]

Loop		
Compute score: left/right	Merge for best score	Advance two blocks
Until no more merges		



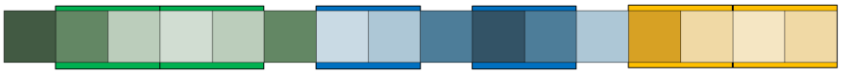
Encoding [Dolonius et al. 2017]

Loop		
Compute score: left/right	Merge for best score	Advance two blocks
Until no more merges		



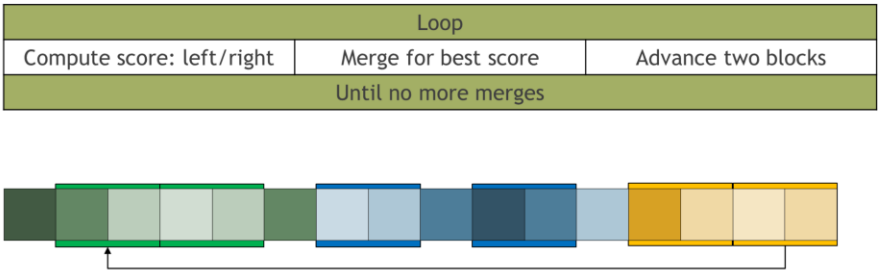
Encoding [Dolonius et al. 2017]

Loop		
Compute score: left/right	Merge for best score	Advance two blocks
Until no more merges		



And now we are done processing all blocks

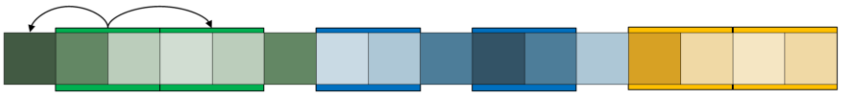
Encoding [Dolonius et al. 2017]



And we start over at the beginning using our new merged blocks instead

Encoding [Dolonius et al. 2017]

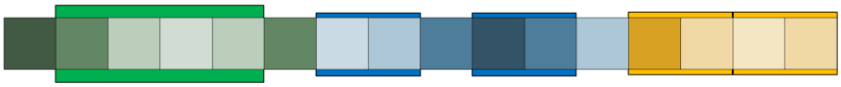
Loop		
Compute score: left/right	Merge for best score	Advance two blocks
Until no more merges		



Look left and right

Encoding [Dolonius et al. 2017]

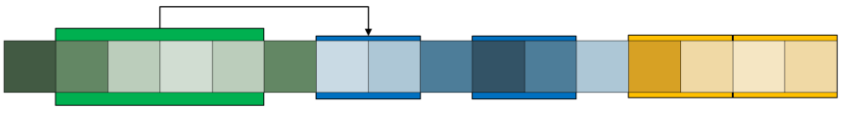
Loop		
Compute score: left/right	Merge for best score	Advance two blocks
Until no more merges		



Merge

Encoding [Dolonius et al. 2017]

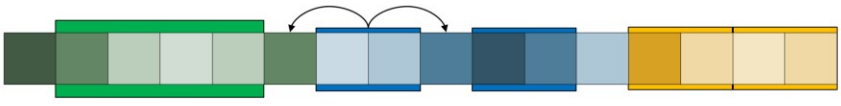
Loop		
Compute score: left/right	Merge for best score	Advance two blocks
Until no more merges		



Advance

Encoding [Dolonius et al. 2017]

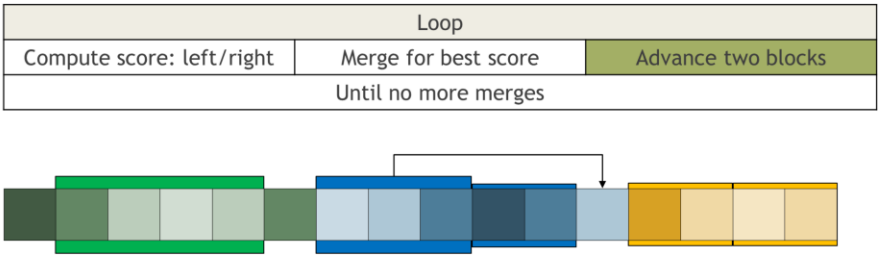
Loop		
Compute score: left/right	Merge for best score	Advance two blocks
Until no more merges		



Encoding [Dolonius et al. 2017]

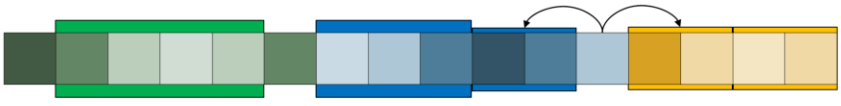
Loop		
Compute score: left/right	Merge for best score	Advance two blocks
Until no more merges		

Encoding [Dolonius et al. 2017]



Encoding [Dolonius et al. 2017]

Loop		
Compute score: left/right	Merge for best score	Advance two blocks
Until no more merges		

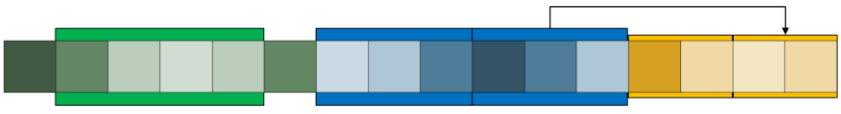


Encoding [Dolonius et al. 2017]

Loop		
Compute score: left/right	Merge for best score	Advance two blocks
Until no more merges		

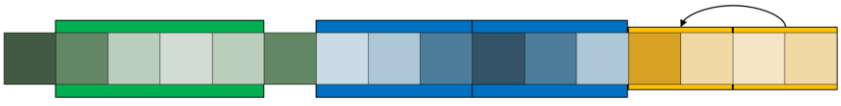
Encoding [Dolonius et al. 2017]

Loop		
Compute score: left/right	Merge for best score	Advance two blocks
Until no more merges		



Encoding [Dolonius et al. 2017]

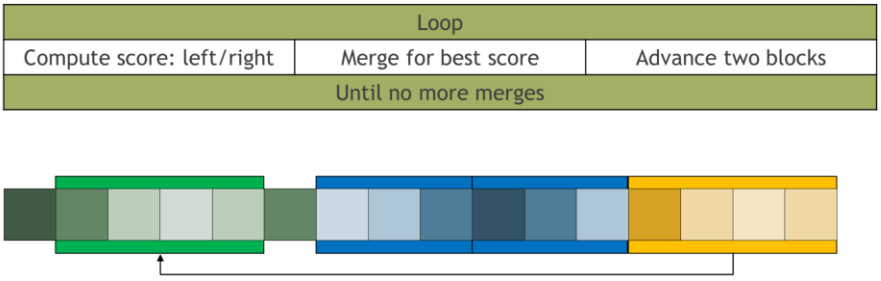
Loop		
Compute score: left/right	Merge for best score	Advance two blocks
Until no more merges		



Encoding [Dolonius et al. 2017]

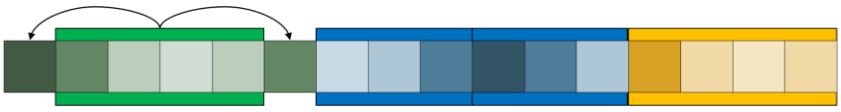
Loop		
Compute score: left/right	Merge for best score	Advance two blocks
Until no more merges		

Encoding [Dolonius et al. 2017]



Encoding [Dolonius et al. 2017]

Loop		
Compute score: left/right	Merge for best score	Advance two blocks
Until no more merges		

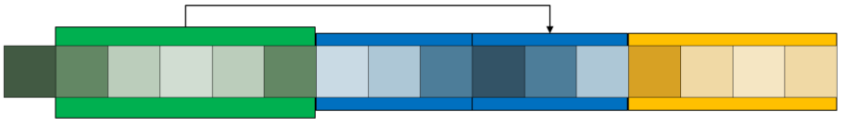


Encoding [Dolonius et al. 2017]

Loop		
Compute score: left/right	Merge for best score	Advance two blocks
Until no more merges		

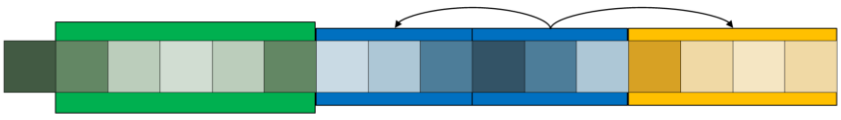
Encoding [Dolonius et al. 2017]

Loop		
Compute score: left/right	Merge for best score	Advance two blocks
Until no more merges		



Encoding [Dolonius et al. 2017]

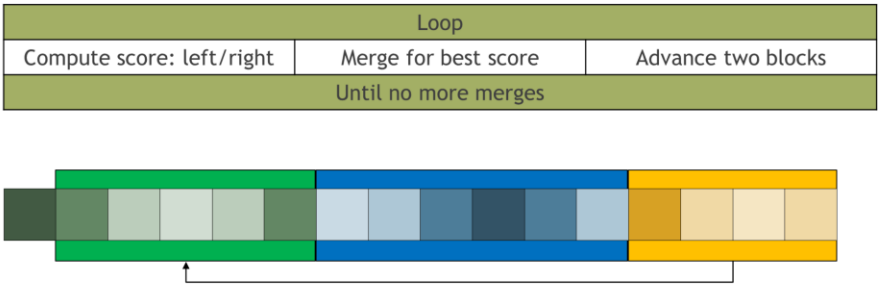
Loop		
Compute score: left/right	Merge for best score	Advance two blocks
Until no more merges		



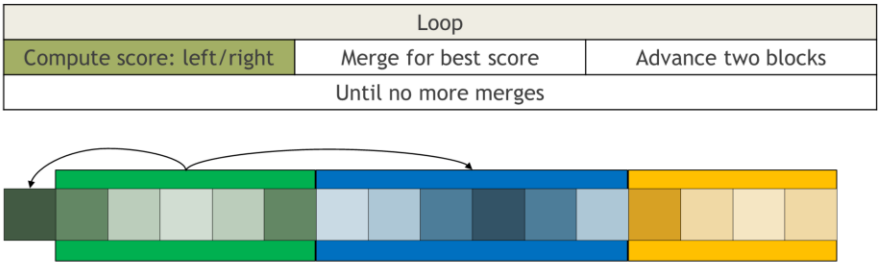
Encoding [Dolonius et al. 2017]

Loop		
Compute score: left/right	Merge for best score	Advance two blocks
Until no more merges		

Encoding [Dolonius et al. 2017]

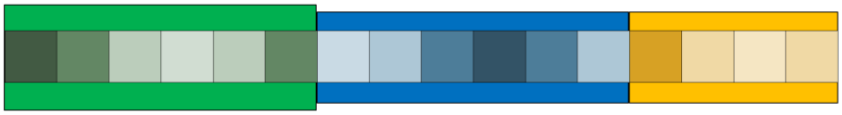


Encoding [Dolonius et al. 2017]

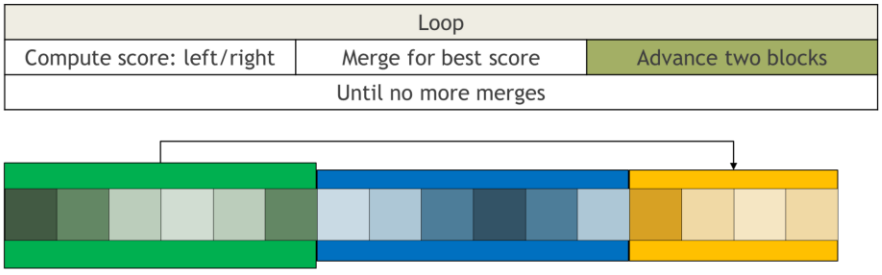


Encoding [Dolonius et al. 2017]

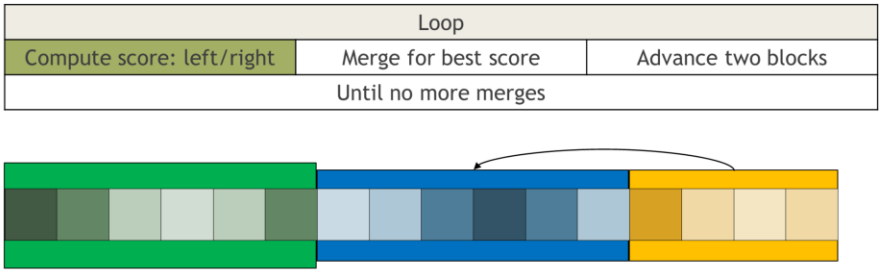
Loop		
Compute score: left/right	Merge for best score	Advance two blocks
Until no more merges		



Encoding [Dolonius et al. 2017]

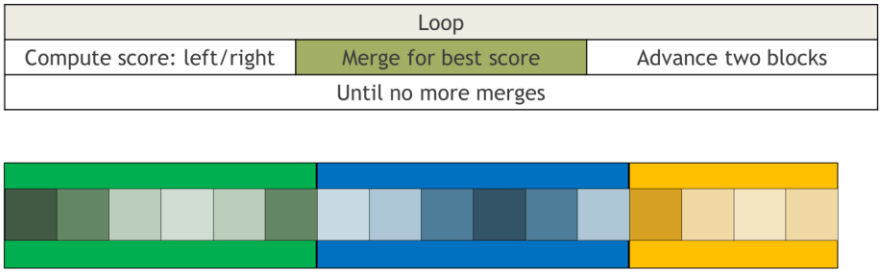


Encoding [Dolonius et al. 2017]



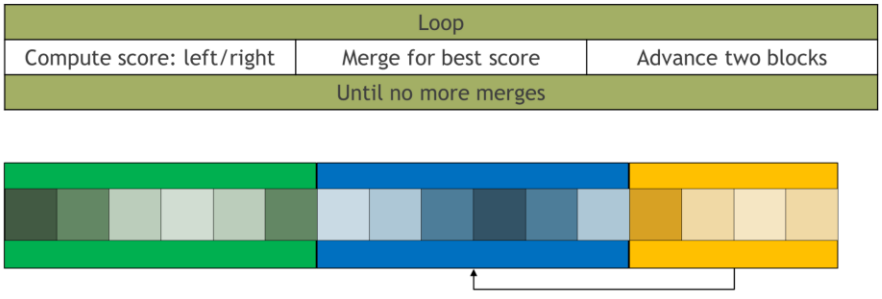
Now, when we got here the only possible merge is to the left, and this would result in a too large error, so we can not perform any more merges with this block, so we simply move on.

Encoding [Dolonius et al. 2017]



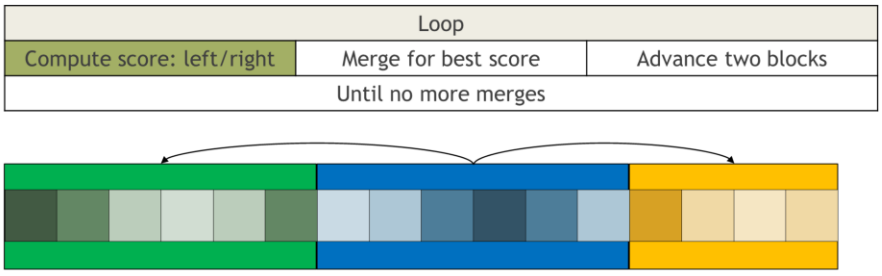
No merge

Encoding [Dolonius et al. 2017]



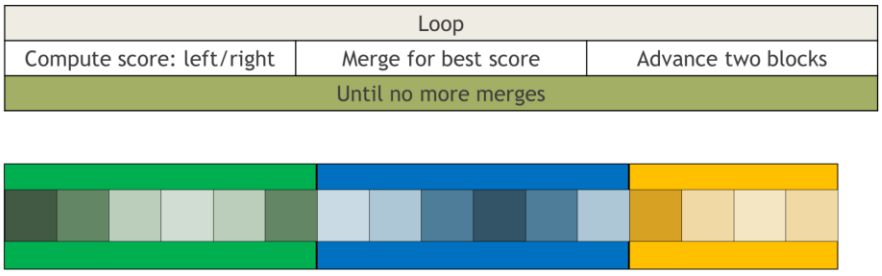
And since this was the last block, we again start over.

Encoding [Dolonius et al. 2017]



Finally, in this pass neither the left or right cold me merged

Encoding [Dolonius et al. 2017]



And when we can't merge any blocks in a pass, we will not be able to merge any in the next pass either.
In other words, there's nothing more we can do to increase the size of our blocks.
So we consider the encoding to be complete.

In the next, final section, I will describe how we improved this format, as one of the drawbacks is that, if we have 3 bits per weight, there is a theoretical minimum of 12% compression.

Improving the format

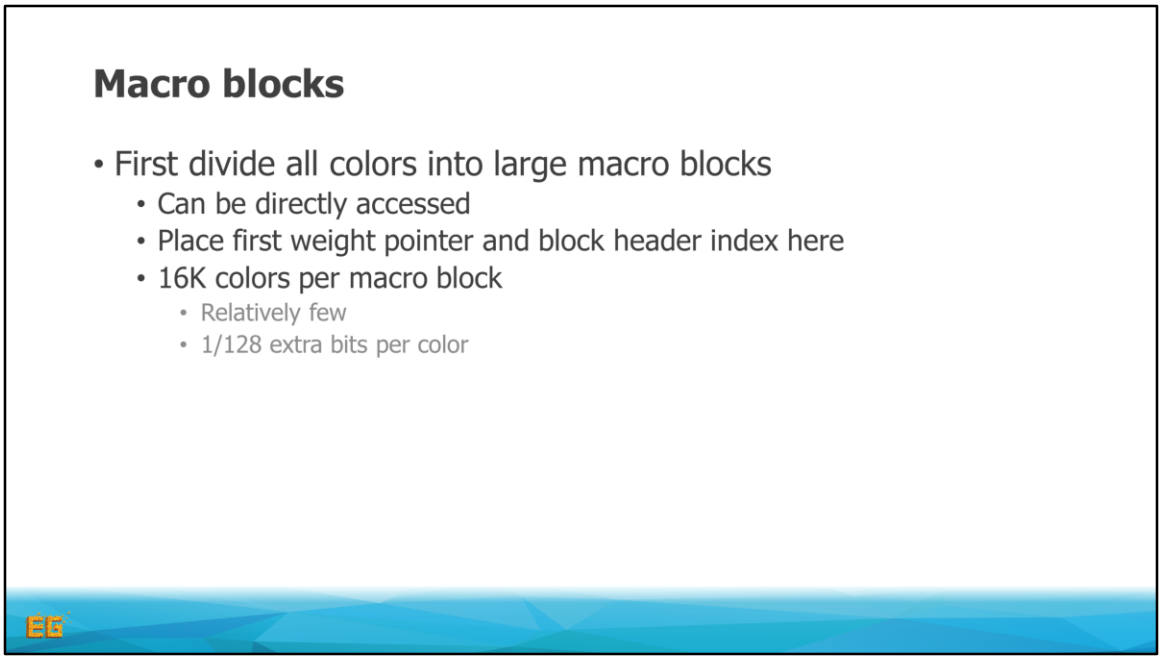
- 3 bits per weight
  - 12% theoretical minimum
- Variable number of bits per weight
  - Each block has an individual bitwidth
  - Problematic
    - Weight array can no longer be directly accessed

If we instead use a variable bit width for each block, we could potentially compress better, as, for example, a large range of equal colors cold be represented with one block and no weights.
Introducing this functionality is however a bit problematic, as we can no longer directly access the weight array as before

Solution

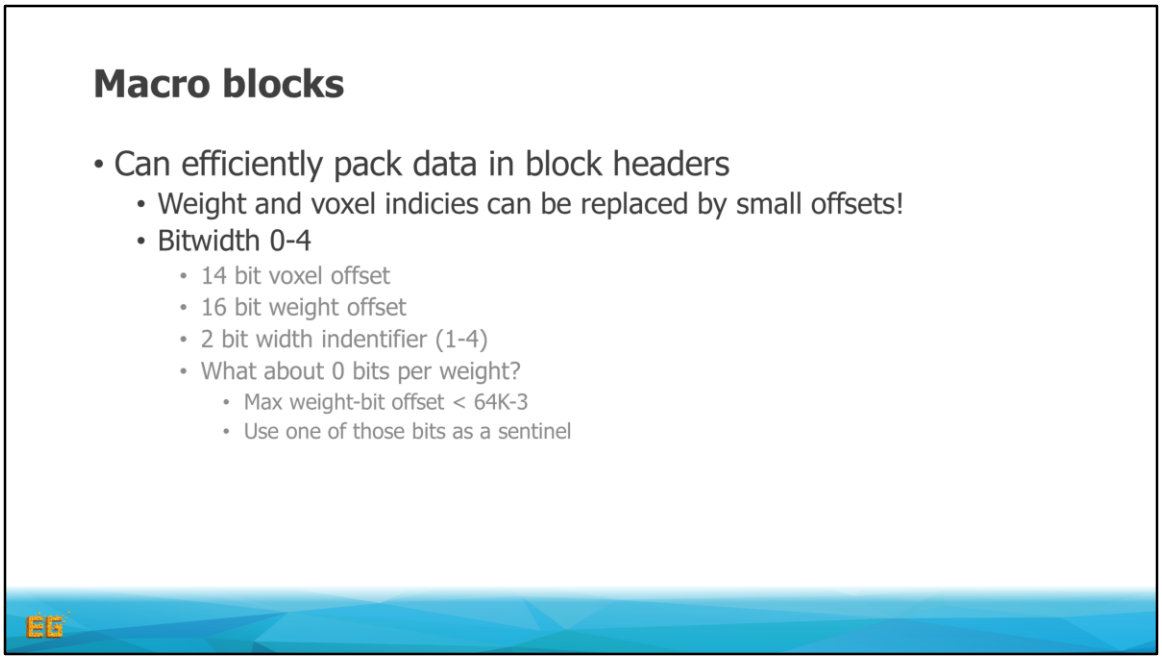
- Store a bit index to the weight array for each block
  - Requires large index
  - Naively implemented increases sizes of blocks
- So let's add more blocks!

So, now we need to store a bit index in the block headers.
Since we will have many, many bits, we need a large index, and if we implement it naively it would considerably increase the size of the blocks.
<click>
One way we could solve this is to introduce another kind of blocks.



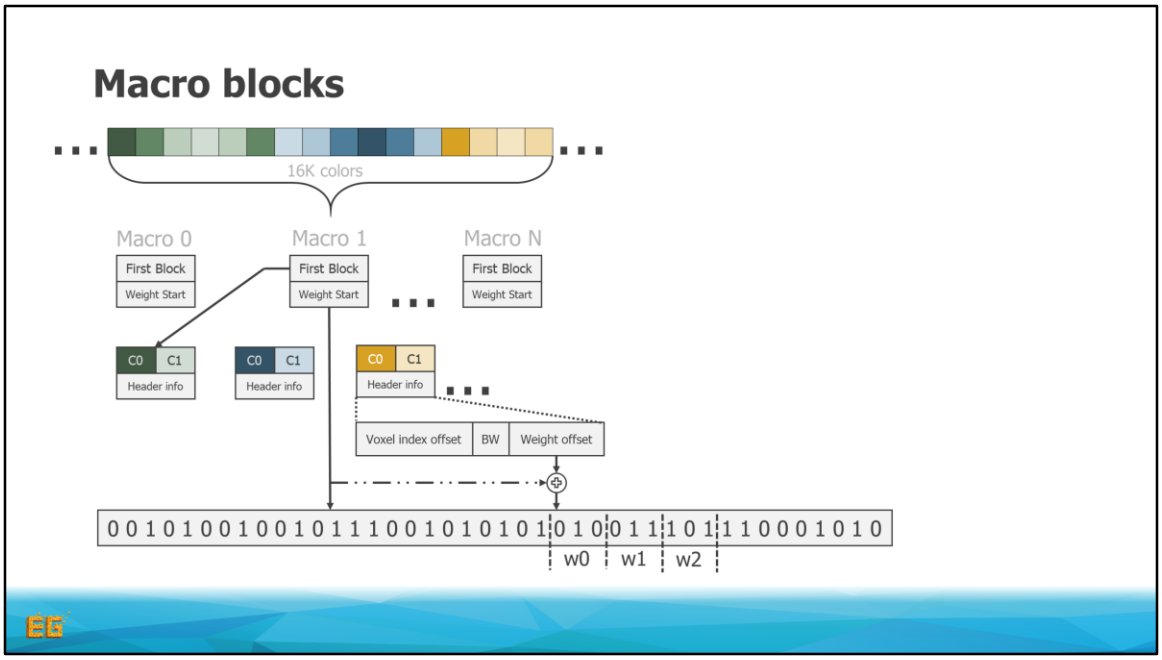
If we first divide the colors in to large blocks of a fixed size, which we call macro blocks.
These can then be directly accessed.
We place the first weight pointer and block header index here (which is similar to Dados implementation).

In this case the blocks are about sixteen thousand colors.
So they will be relatively few, and thus only add about a one over one hundred twenty eight bits memory overhead to each color



With this we can now efficiently pack the data in the block headers, as we can replace the weight and voxel indices by small offsets.
With a bit width of zero to four bits we can use
14 bits as the voxel offset
16 bits as the weight offset
2 bits as the width identifier for one to four bits

However, this only solves all but the zero bit case…
But fortunately, as the max weight-bit offset will be less than 64K-3, we can use one unused bit in the weight offset as a zero bit sentinel.



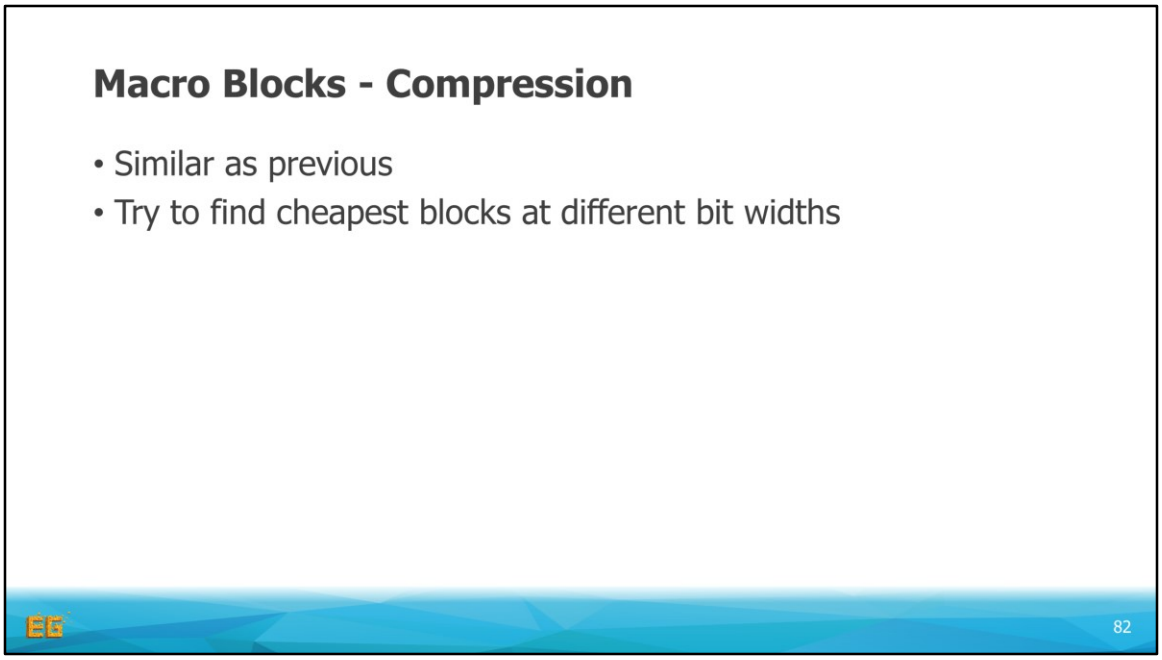
So this is what this data structure looks like. And I will now explain how we can decode this structure.

Macro blocks - Decompression

- Almost same as previous
- Do a direct lookup in macro blocks
- Calculate range of block headers
- Binary search to find header
  - Acually cheaper!
  - Range is smaller than for previous algorithm
- Use start and offsets to find weights
- Interpolate

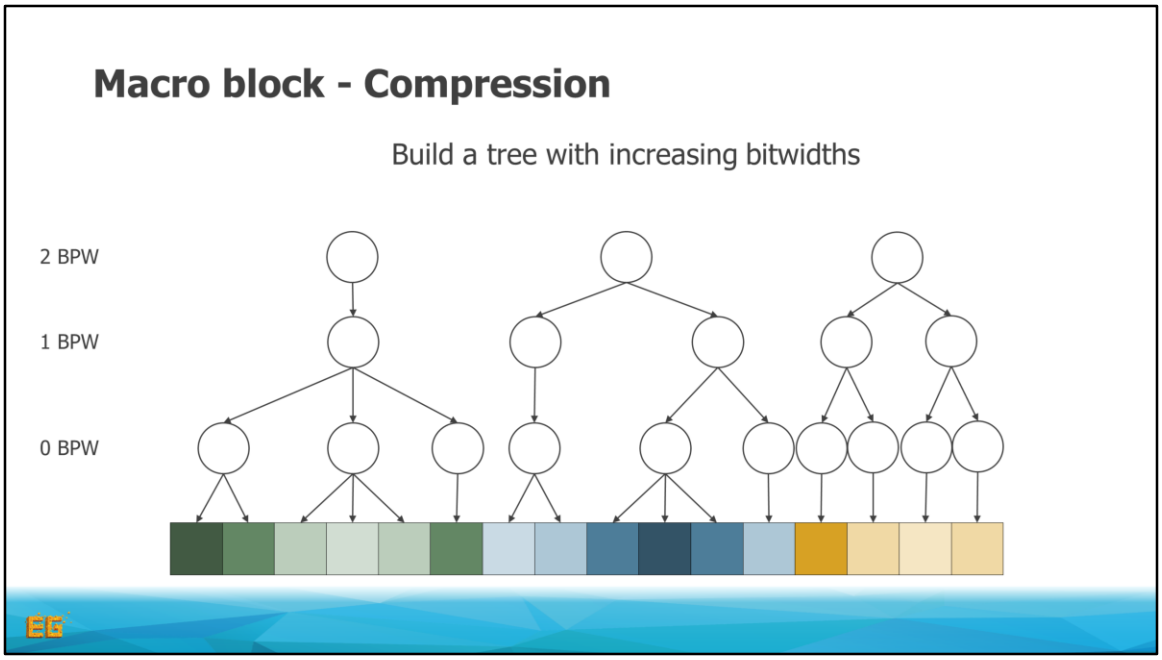
It's almost the same as the previous method.
But we first do a direct lookup in the macro blocks.
We use two macro blocks to calculate the range of block headers where the color resides.
Then we do a binary search, to find the correct block.
This is actually cheaper since the range is reduced as compared to the previous method.

We then use the start offsets to find the weight, and finally interpolate, as before.

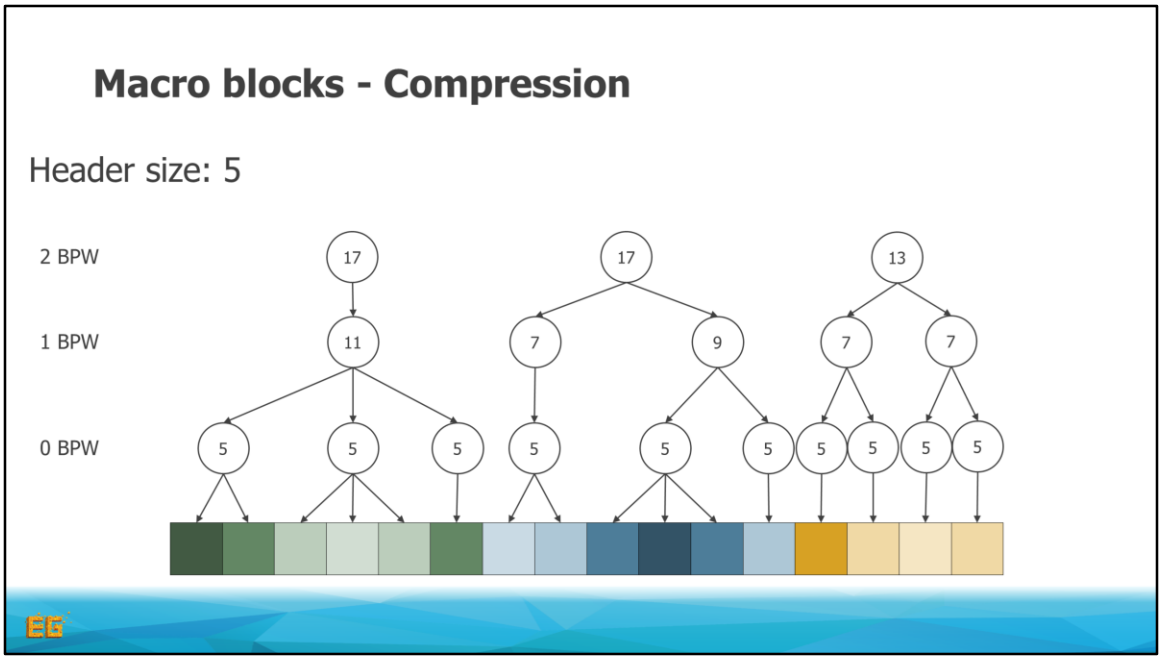


The compression is also similar to the previous algorithm, where we will merge blocks as before.

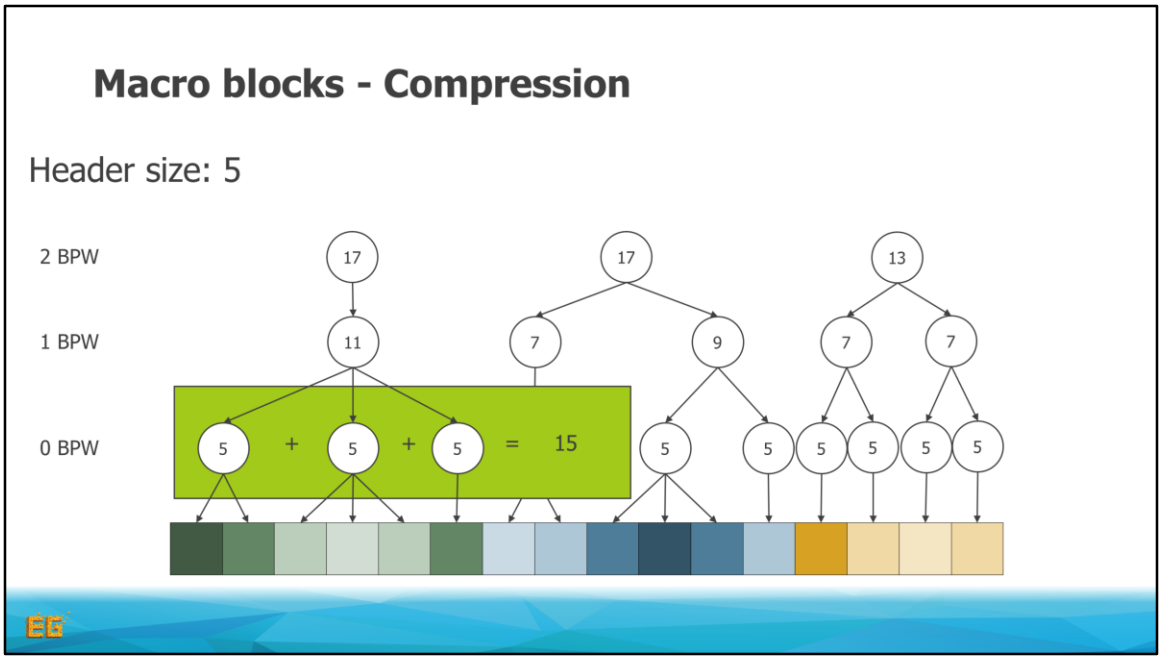
But the extra detail is that we will build a tree of merged blocks, with increasing bit widths, and then do a cut through this tree to find the most memory efficient approximation



To demonstrate.
We have the input colors at the bottom. And merge them using zero bits per weight.
<click>
And use them for the next level.
<click>
And the second.
<click>



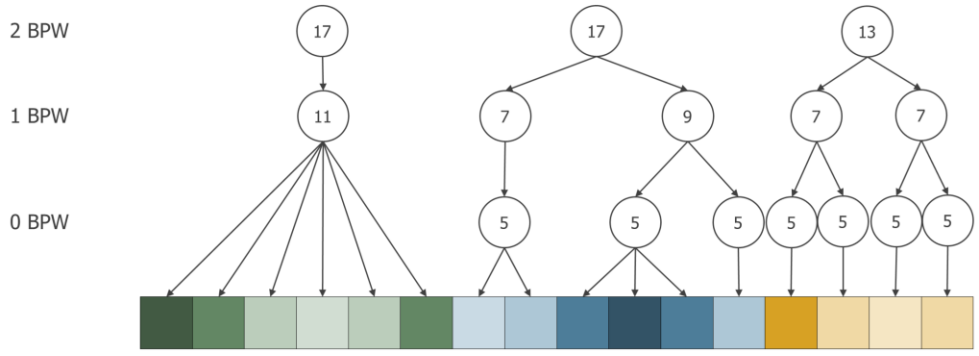
To find the cut, we first calculate the memory requirement for each node.
This is the constant header size plus the size of the weights. Here I have chosen a header size of 5, just to illustrate better.
As we see at the lowest level
<click>
Each block has zero weight cost, and the only cost is the header.



The cut is performed by, bottom up, comparing the sum of the cost of the children, by their parent. And removing the largest.
<click>
So here the sum of the children is 15, which is larger than 11
<click>
So we will remove these nodes
<click>
So here it was cheper having this one larger one bit per weight block, instead of three zero blocks.

Macro blocks - Compression

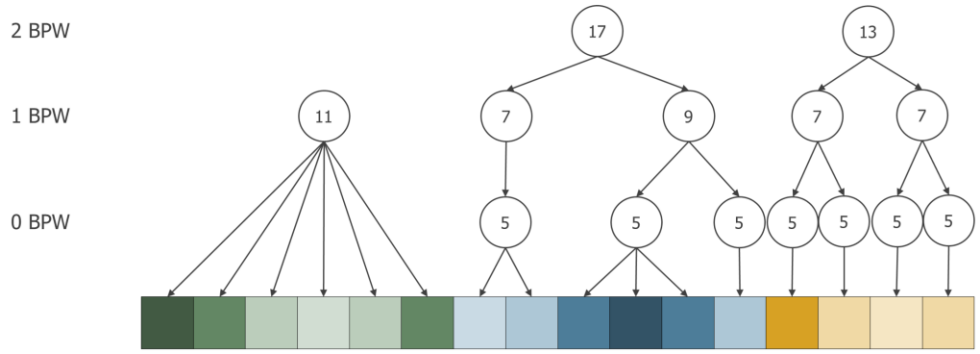
Header size: 5



Similarly, we drop the largest node here

Macro blocks - Compression

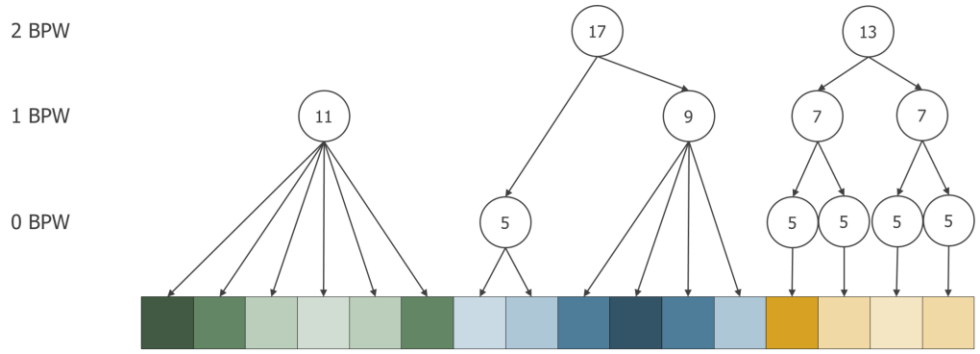
Header size: 5



And here

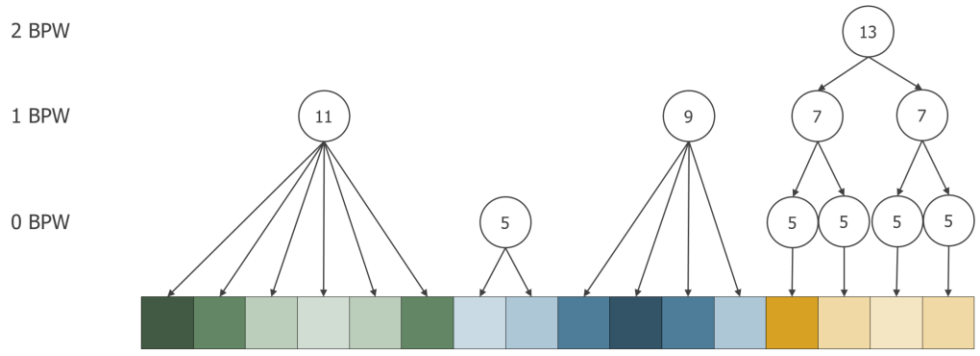
Macro blocks - Compression

Header size: 5



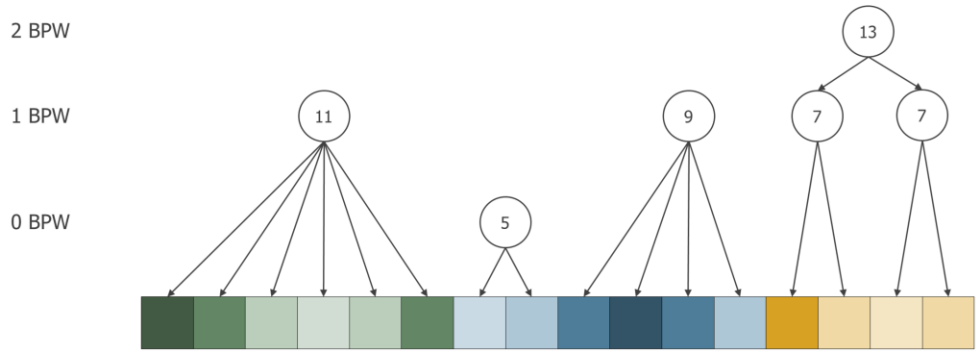
Macro blocks - Compression

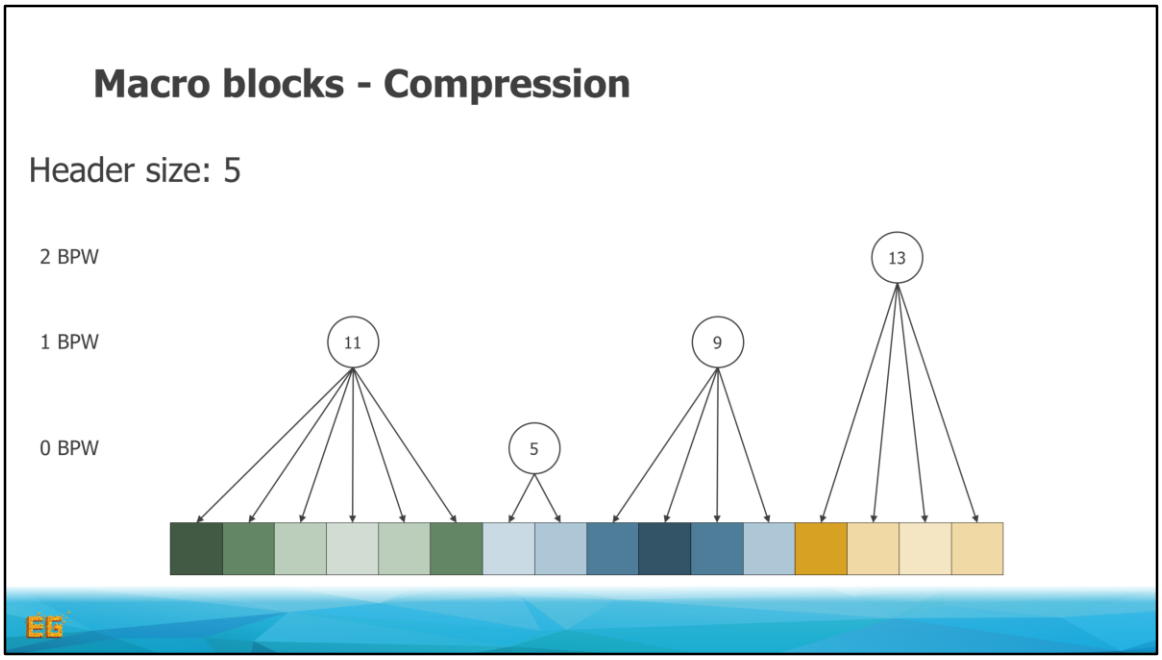
Header size: 5



Macro blocks - Compression

Header size: 5





And with this we are done.

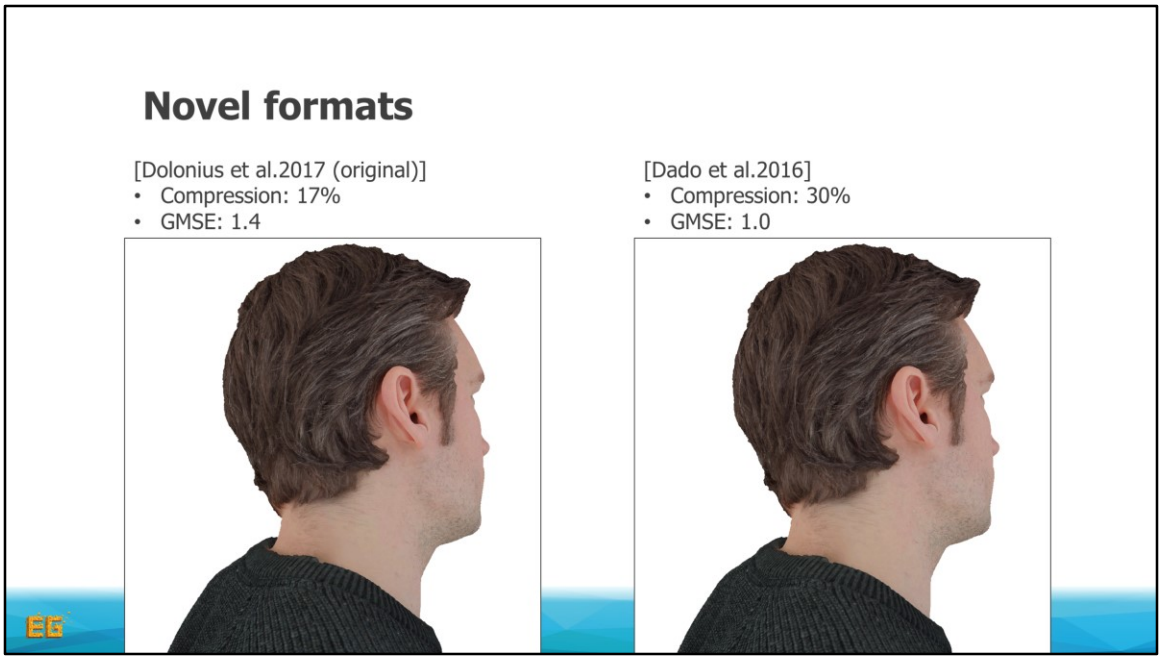
Aftermath

- E.g. Sponza contained much hidded geometry
  - Large black regions when baked
  - Beneficial to Dado and improved Dolonius
  - Tried removing those regions
    - 17% -> 26% GRMSE: 3.0 -> 5.5 (Dado)
    - 8% -> 12% GRMSE: 3.3 -> 5.9 (Improved Dolonius)
    - 16% -> 16% GRMSE: 2.7 -> 5.3 (Original Dolonius)
- Dado benefits from scenes with a limited colorspace
- Dado lossless compression is possible

As a final remark I want to say that the sponza scene contains a lot of hidden geometry.
When baking lighting information, this will result in large blocks being completely black.
And while it degraded the quality for the original method by Dolonius, it is in a way handled well by the improved method, or Dado et al,
Because we can find large cheap blocks, or small palettes representing many colors.

So when removing those regions we see that both Dado and the improved Dolonius format compresses less, with a wors quality.
And the original Dolonius format does not change in compression ratio, as it's sub optimal to begin with, but does reduce in quality.

As a final remark we note that the palette based version by Dado et al. benefits from scenes with a limited colorspace, and can handle lossless compression.



And here I will present the final compression results in this presentation..
In this case we chose a high resolution body scan which was freely available from ten24.
To the left we have the original format by Dolonius et al. and to the right, the method used by Dado et al.
We chose the compression ratios to match a similar error. As we can see, both have very good quality with a compression at 17% for our case, and 30% for dado et al.
At this compression it practically impossible so see any difference compared to the original data.

Novel formats

[Dolonius et al.2017 (original)]
- Compression: 13%
- GMSE: 3.7

[Dado et al.2016]
- Compression: 24 %
- GMSE: 4.6

If we are a bit more agressive with the compression we see that we are able to get down to 13% for our method with a comparable mse to that of dado et al at 24%

Novel formats

[Dolonius et al.2017 (original)]
- Compression: 13%
- GMSE: 3.7

[Dado et al.2016]
- Compression: 24 %
- GMSE: 4.6

To highlight the artifacts at this compression we zoom in at this region

Novel formats

[Dolonius et al.2017 (original)] [Dado et al.2016]

Here we can see the palletization artifacts by Dado et al. and maybe some blocky artifacts by Dolonius et al.
But in the next slide I will show a comparison with the improved format and Dado et al., to highlight the different artifacts from the two methods under extreme compression

Novel formats

[Dolonius et al.2017 (improved)]
- Compression: 11%
- GMSE: 1.6

[Dado et al.2016]
- Compression: 21 %
- GMSE: 0.9

Here we have the sponza scene with a quite decent compression, and no conceivable artifacts for the two methods.

But if we crank up the compression…
<click>

Novel formats

[Dolonius et al.2017 (improved)]
- Compression: 3.7% (0.9 bits per color)
- GMSE: 13.3

[Dado et al.2016]
- Compression: 11 %
- GMSE: 25.0

We start to see the difference.
The method by Dado et al., to the right, preserves detail, at an expense of colors at a compression of 11%.
While the improved compression by Dolonius et al. at 3.7% preserves the overall colors better, but at a loss of detail.
Worth mentioning here is that, at 3.7%, we use less than one bit per color.

And with this I end my part of the course, thank you.