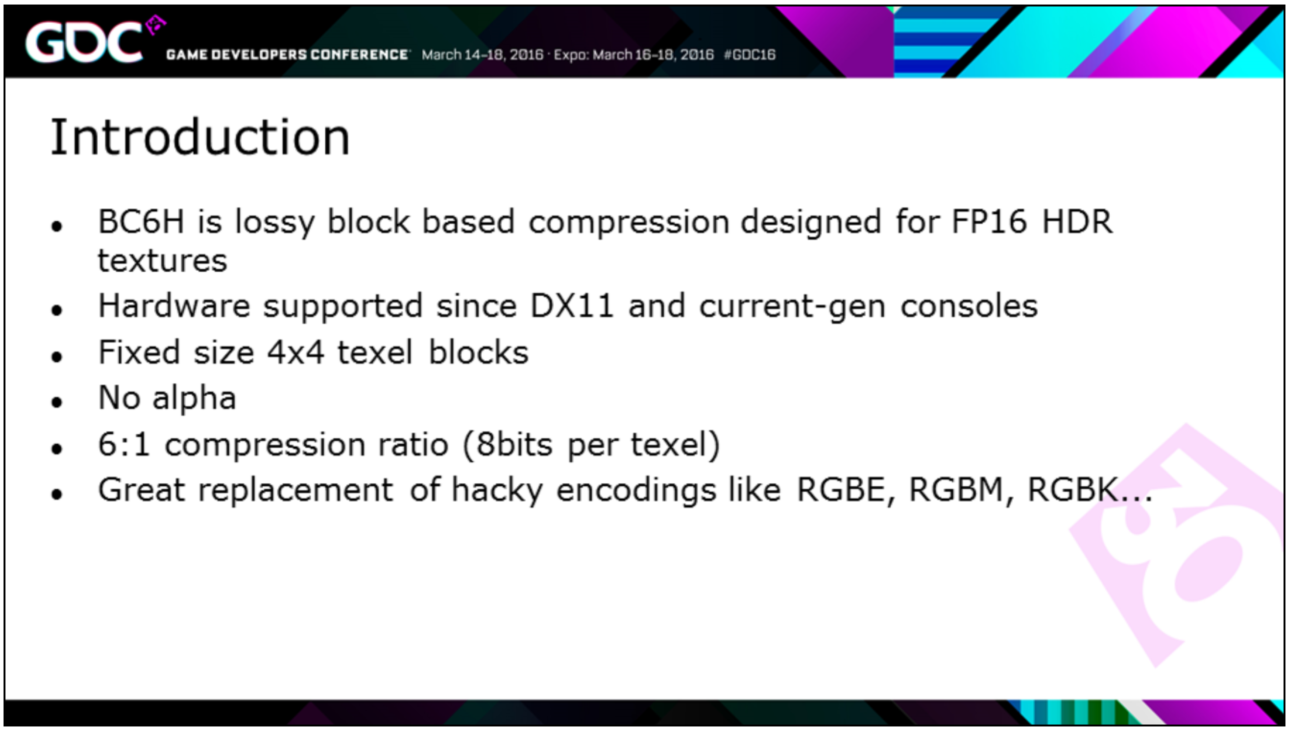
Real-Time BC6H Compression on GPU

Krzysztof Narkowicz
Lead Engine Programmer
Flying Wild Hog

GAME DEVELOPERS CONFERENCE March 14–18, 2016 · Expo: March 16–18, 2016 #GDC16

Hi everyone, my name is Krzysztof Narkowicz. I'm the Lead Engine Programmer at Flying Wild Hog. It's a small company based in Poland and we are mainly making some old-school first person shooter games like Hard Reset or Shadow Warrior's remake called Shadow Warrior.

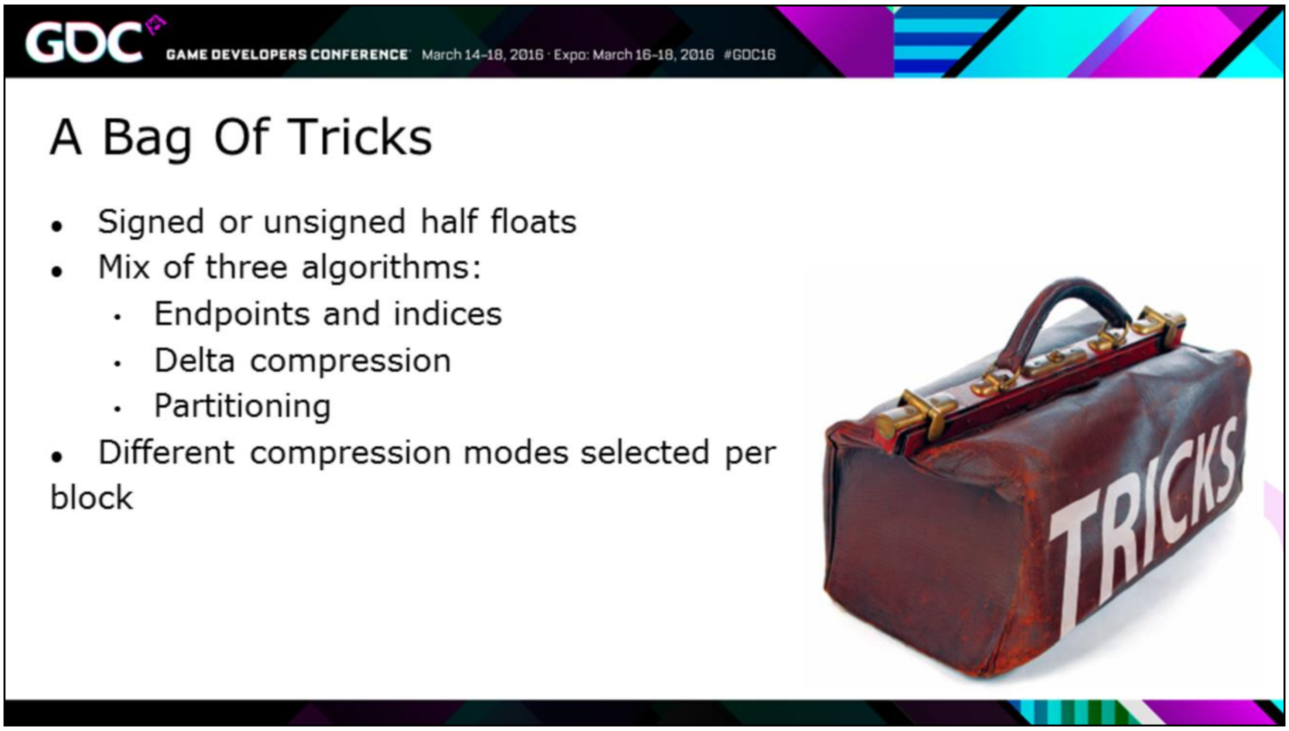
I'll be speaking today about BC6H compression, how it works, how to compress it in real-time and what's the motivation behind doing that.


GAME DEVELOPERS CONFERENCE March 14-18, 2016 · Expo: March 16-18, 2016 #GDC16

Introduction

- BC6H is lossy block based compression designed for FP16 HDR textures
- Hardware supported since DX11 and current-gen consoles
- Fixed size 4x4 texel blocks
- No alpha
- 6:1 compression ratio (8bits per texel)
- Great replacement of hacky encodings like RGBE, RGBM, RGBK…

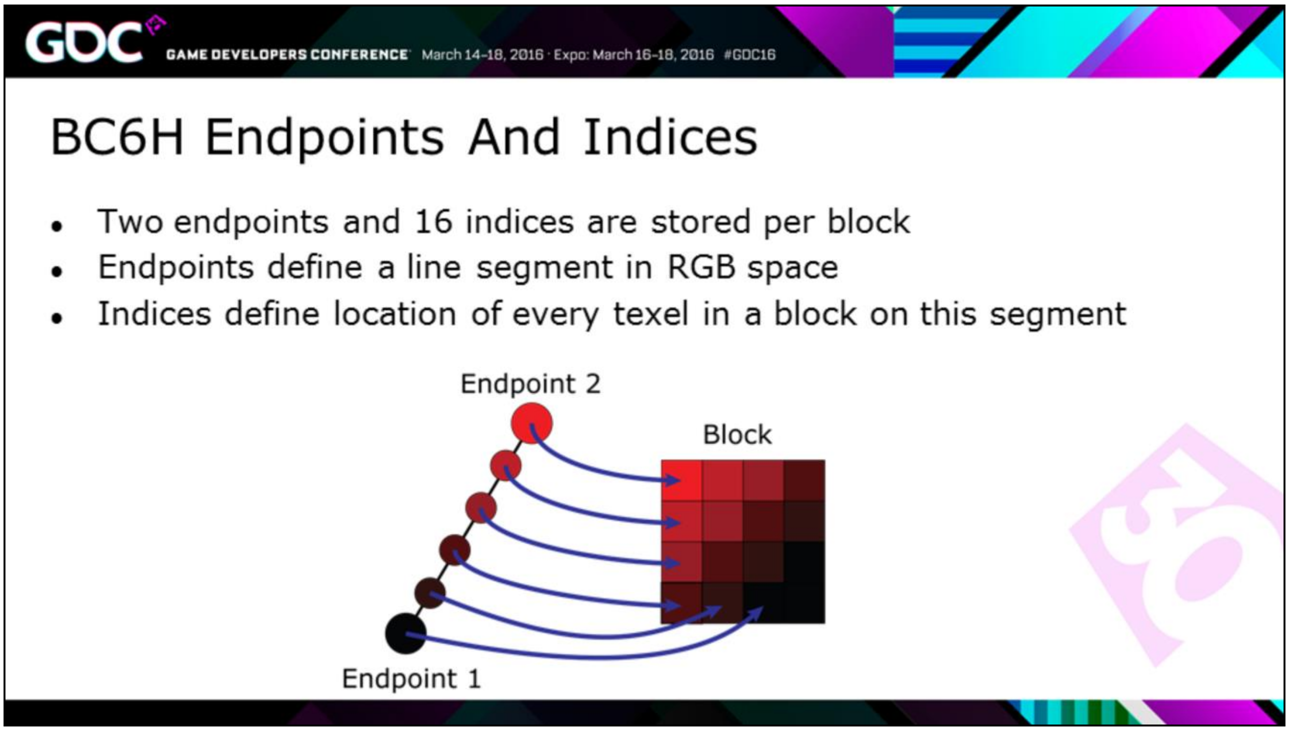
BC6H is a lossy block based compression format designed for compressing half floating point textures. It's fully hardware supported starting from DX11 and current-gen consoles. It uses fixed size 4x4 texel blocks, which is very convenient for native hardware decompression. It doesn't support alpha and sampling alpha is required to return one. It has 6:1 compression ratio or in other words it uses 8 bits per texel. All these properties make it a very convenient replacement of hacky encodings like RGBE, RGBM or RGBK, which we used in previous generation games for storing HDR textures.


GAME DEVELOPERS CONFERENCE March 14-18, 2016 · Expo: March 16-18, 2016 #GDC16

A Bag Of Tricks

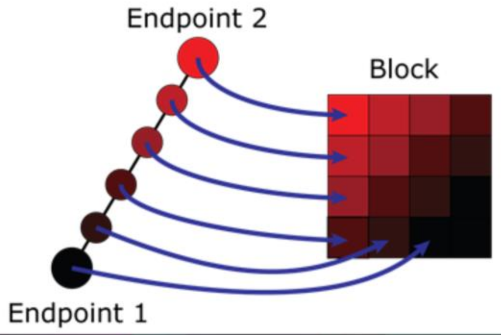
- Signed or unsigned half floats
- Mix of three algorithms:
  - Endpoints and indices
  - Delta compression
  - Partitioning
- Different compression modes selected per block

BC6H is much more complicated than previous BC formats. It’s more like a bag of tricks than a single compression algorithm. It has many options. It can compress signed or unsigned half floating point HDR textures. It uses three different algorithms: endpoints and indices, delta compression and partitioning. It has different compression modes, which can be selected per block. Those modes define which algorithms will be used for the current block and additionally define some compression tradeoffs per block. Effectively, each block can have a different compression format.

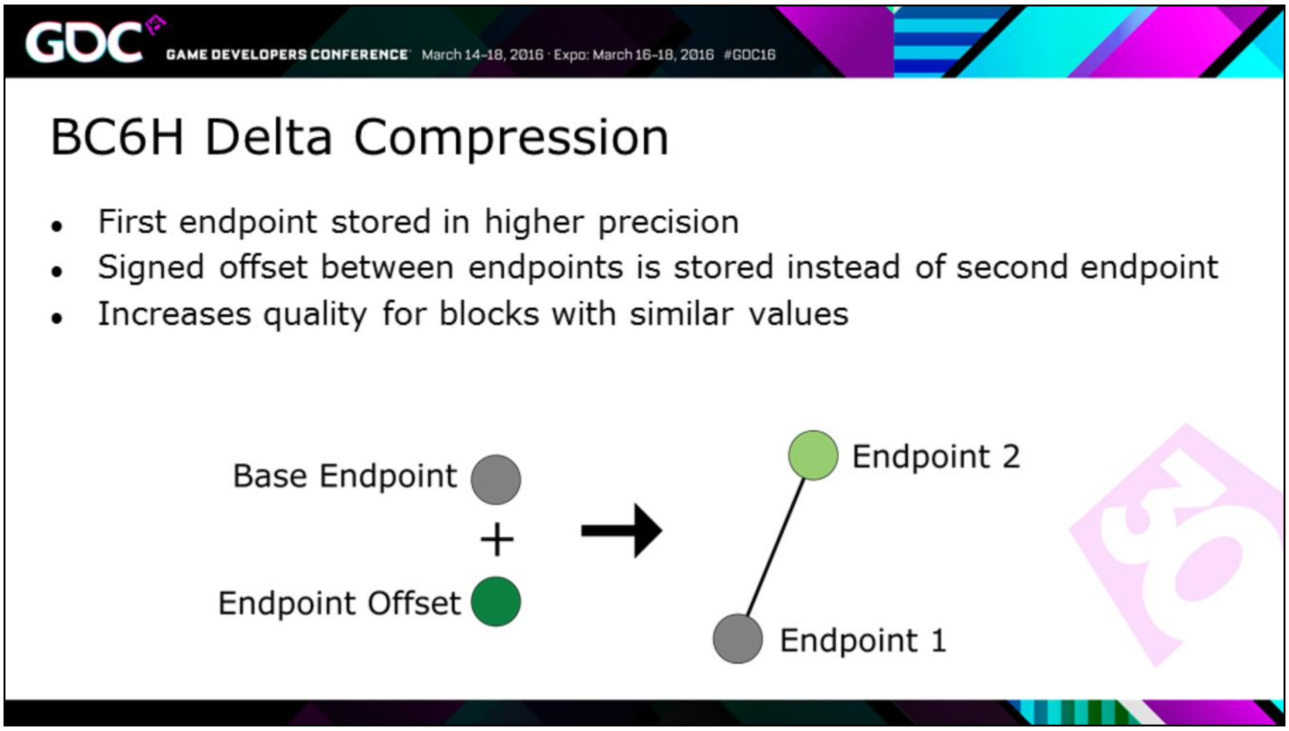


BC6H Endpoints And Indices

- Two endpoints and 16 indices are stored per block
- Endpoints define a line segment in RGB space
- Indices define location of every texel in a block on this segment

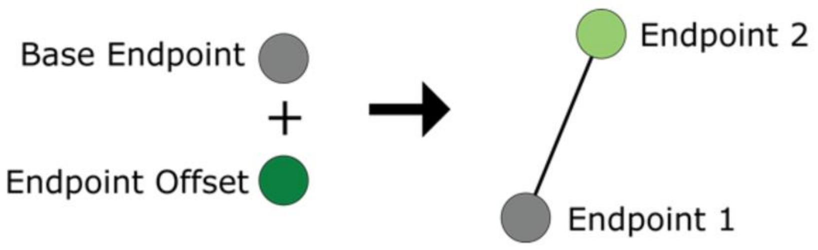


Endpoints and indices were the only algorithm in previous BC formats. 2 endpoints and 16 indices are stored per block. Endpoints define a line segment in RGB space and indices define a location of every texel in a block on this segment. In other words endpoints form a color palette and indices assign every texel in a block to a single entry of this color palette.

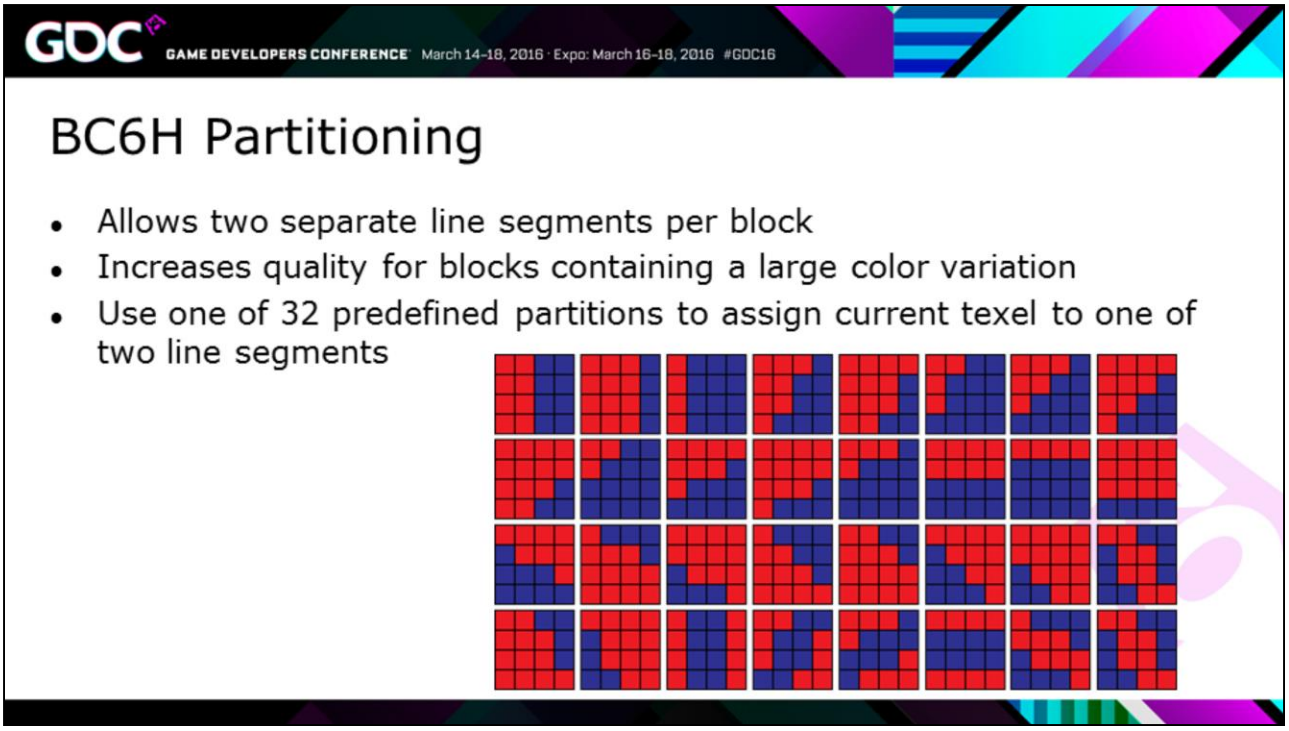

GAME DEVELOPERS CONFERENCE March 14-18, 2016 · Expo: March 16-18, 2016 #GDC16

BC6H Delta Compression

- First endpoint stored in higher precision
- Signed offset between endpoints is stored instead of second endpoint
- Increases quality for blocks with similar values



There is also delta compression. It’s designed for cases when all the colors in a block are similar. Instead of storing two endpoints directly, we store one base endpoint using higher precision and an offset vector. This way we can set line segment’s absolute position more precisely, but we also limit it’s length.

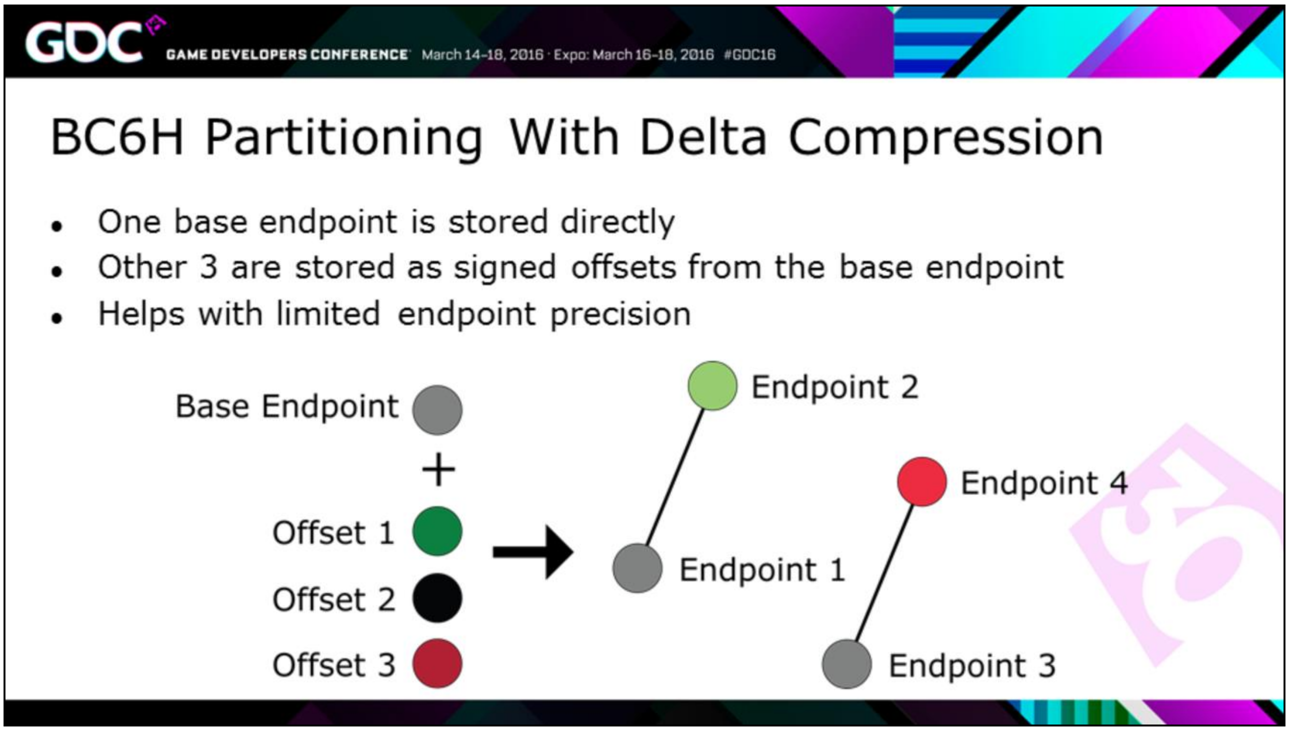


GDC GAME DEVELOPERS CONFERENCE March 14-18, 2016 · Expo: March 16-18, 2016 #GDC16

BC6H Partitioning

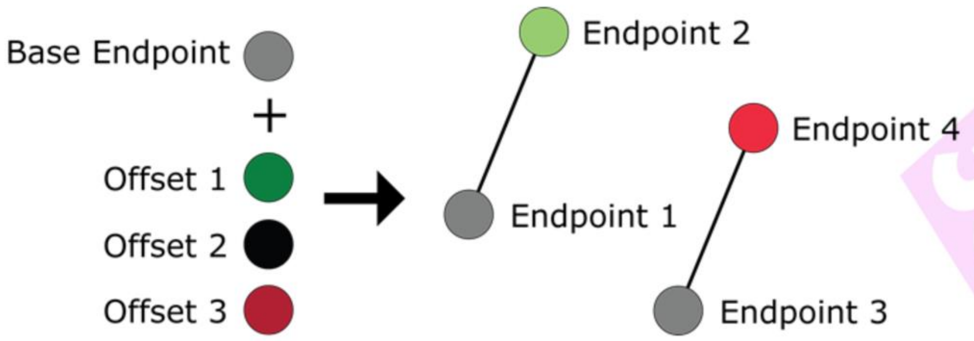
- Allows two separate line segments per block
- Increases quality for blocks containing a large color variation
- Use one of 32 predefined partitions to assign current texel to one of two line segments

There is also partitioning. It was designed for cases when block contains more than 2 distinct colors and it’s hard to approximate this block using just a single line segment. Partitioning allows to use two line segments per block and there are 32 predefined partitions, which assign every texel to one of those two line segments.

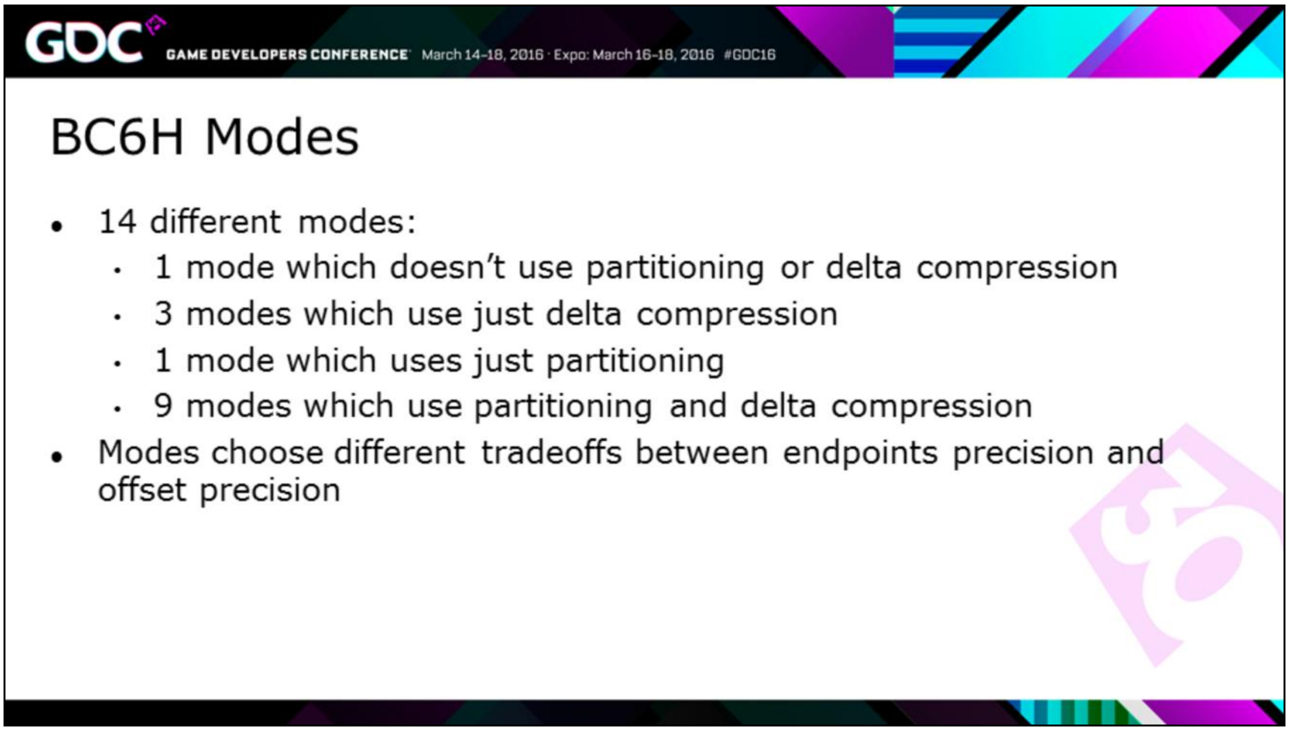


BC6H Partitioning With Delta Compression

- One base endpoint is stored directly
- Other 3 are stored as signed offsets from the base endpoint
- Helps with limited endpoint precision



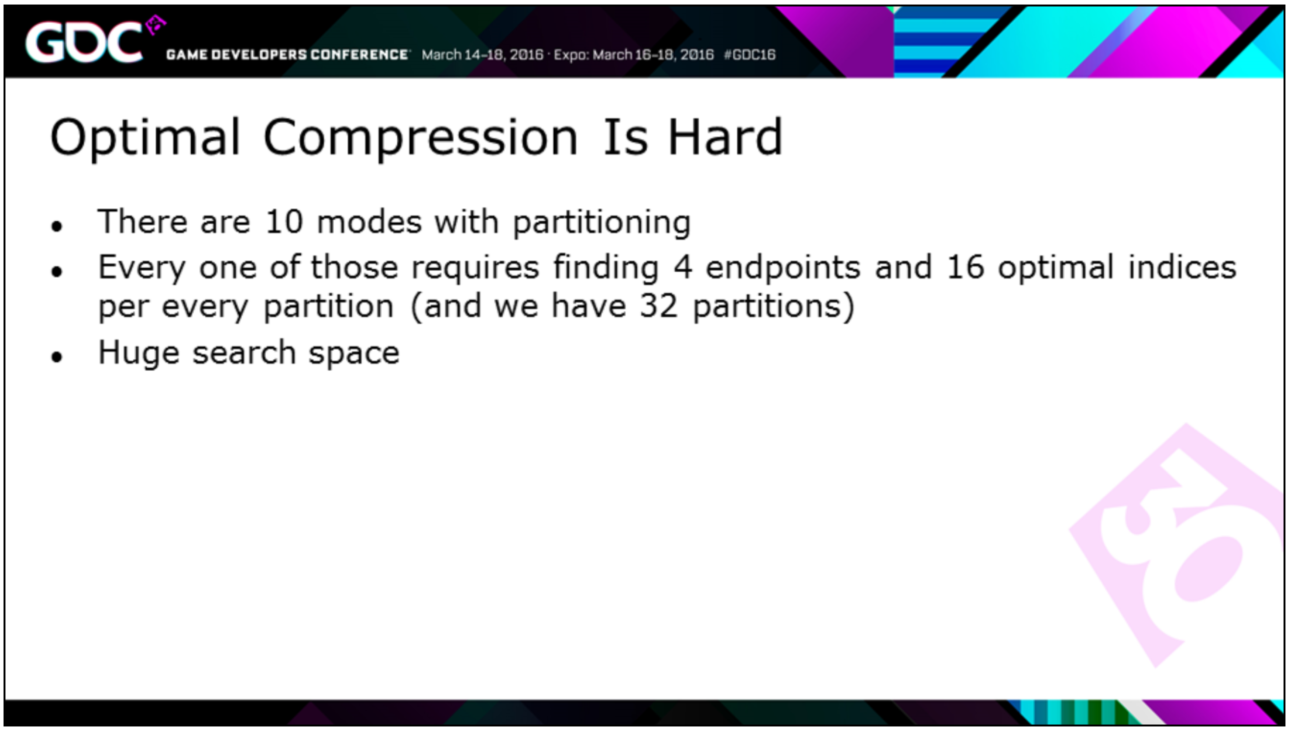
Storing an additional line segment in a fixed size block lowers the precision of endpoints to just 6 bits per channel. This isn’t a lot for storing 16 bit half floating points. Thankfully, we can mix partitioning with delta compression. Then instead of storing 4 endpoints directly, we store one base endpoint using higher precision and a offset vector. This greatly helps with limited endpoint precision.


GDC GAME DEVELOPERS CONFERENCE March 14-18, 2016 · Expo: March 16-18, 2016 #GDC16

BC6H Modes

- 14 different modes:
  - 1 mode which doesn't use partitioning or delta compression
  - 3 modes which use just delta compression
  - 1 mode which uses just partitioning
  - 9 modes which use partitioning and delta compression
- Modes choose different tradeoffs between endpoints precision and offset precision

Finally, we have 14 different modes. Modes define which algorithms will be used for the current block and for the same set of algorithms they define a tradeoff between base endpoint precision and offset vector precision. We can either set line segement's absolute position more precisely or have a longer offset vector.

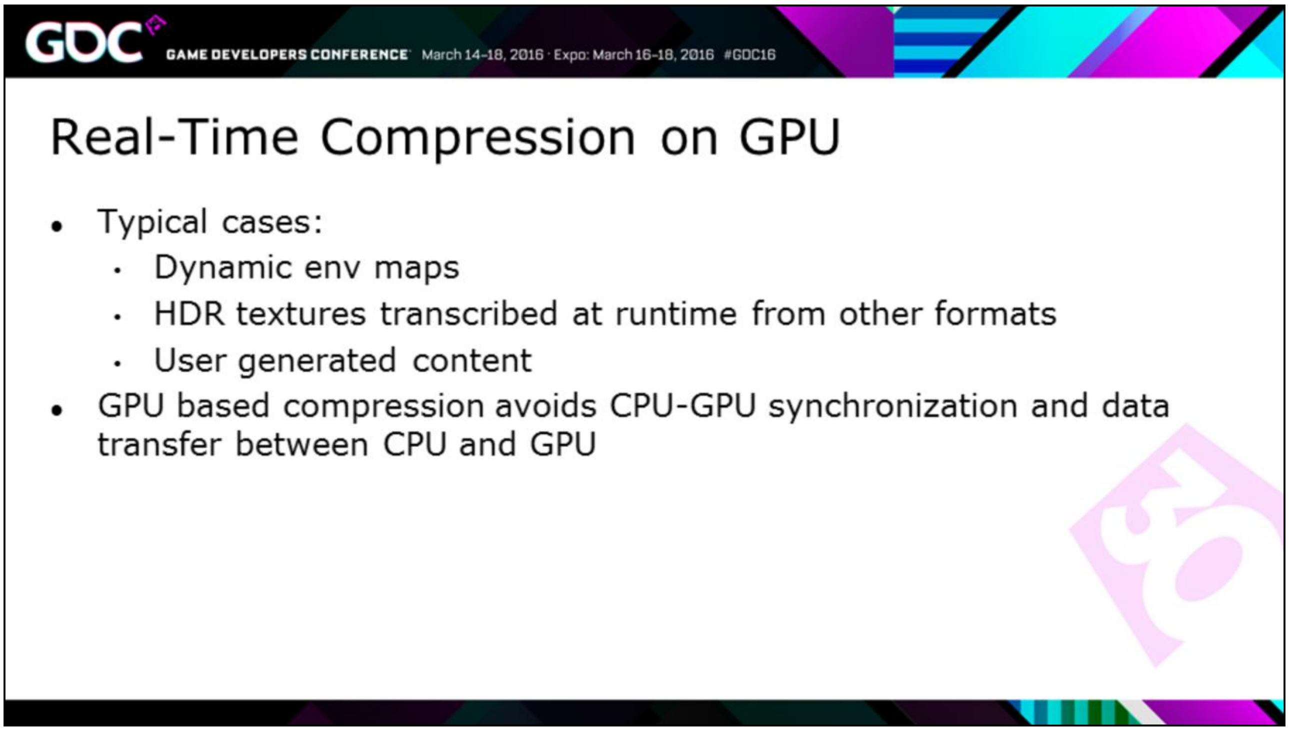


GDC GAME DEVELOPERS CONFERENCE March 14-18, 2016 · Expo: March 16-18, 2016 #GDC16

Optimal Compression Is Hard

- There are 10 modes with partitioning
- Every one of those requires finding 4 endpoints and 16 optimal indices per every partition (and we have 32 partitions)
- Huge search space

BC6H optimal compression is pretty hard. This is why BC6H or BC7, which is a very similar compression format, aren't very widespread among developers. If we just look at the modes with partitioning – we have 10 such modes and every one has 32 partitions and for each partition we need to compute 4 optimal endpoints and 16 indices. Basically, we have to encode every block 320 times. This is really a huge search space and even offline compressors cheat here. For example Intel BC6H compressor's "very slow preset", skips almost the half of modes with partitioning.


GDC GAME DEVELOPERS CONFERENCE March 14-18, 2016 · Expo: March 16-18, 2016 #GDC16

Real-Time Compression on GPU

- Typical cases:
  - Dynamic env maps
  - HDR textures transcribed at runtime from other formats
  - User generated content
- GPU based compression avoids CPU-GPU synchronization and data transfer between CPU and GPU

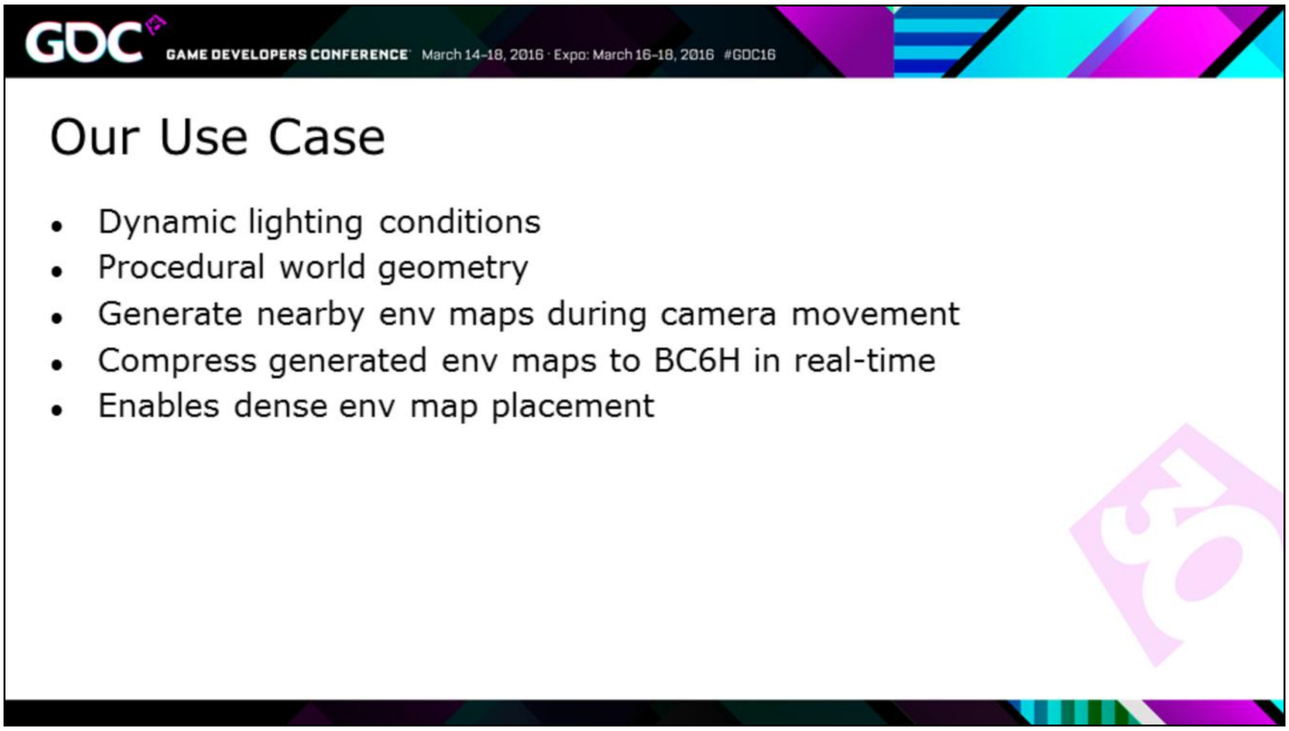
What's the point of compressing HDR textures at runtime?

A typical use case is compressing generated environment maps, which are commonly used for capturing indirect specular. Some games can just generate them offline and store on disk, but it’s not possible for open world games with dynamic lighting conditions, like changing time of a day or changing weather conditions. Usually, open world games store a GBuffer for every environment map and relight it at runtime in order to generate environment maps. It would be nice to compress those generated environment maps using BC6H in order to save some memory and bandwidth.

Another possible use case is virtual texturing. With virtual texturing we have some highly compressed HDR data with some kind of JPEG like codec and want to transcribe it at runtime to a more GPU friendly compression format like BC6H.

Another possible use is user generated content.

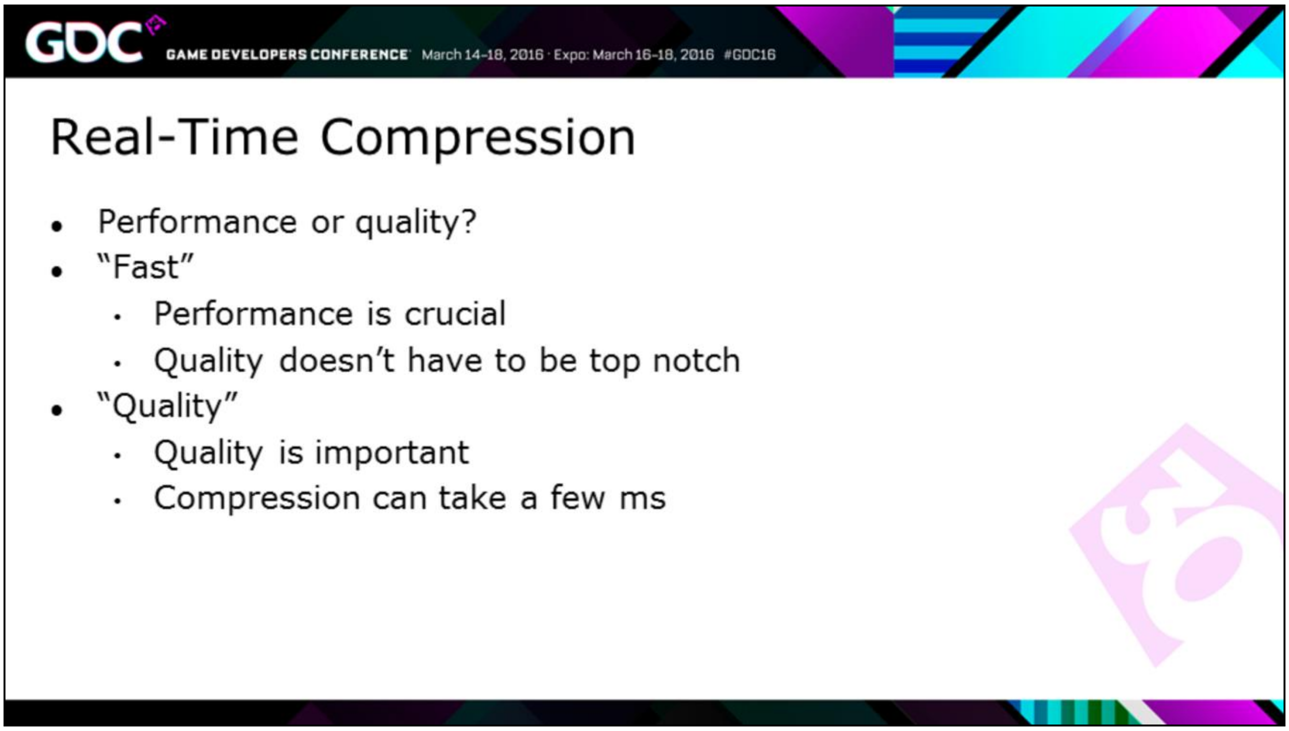
Anyway, in all these cases it's preferable to compress on GPU for performance and simplicity. This way we can skip CPU-GPU synchronization and data transfer between these two units isn’t required.


GDC GAME DEVELOPERS CONFERENCE March 14-18, 2016 · Expo: March 16-18, 2016 #GDC16

Our Use Case

- Dynamic lighting conditions
- Procedural world geometry
- Generate nearby env maps during camera movement
- Compress generated env maps to BC6H in real-time
- Enables dense env map placement

Our use case is quite similar. We have dynamic lighting conditions, so we can’t generate environment maps offline and store them on disk. Additionally, we have dynamic geometry as our levels are procedurally generated at loading time, so we can’t even store GBuffers for environment maps. Our solution is very simple. We just generate environment maps on the fly by rendering simplified scene’s representation. Then we compress generated environment maps using BC6H and store them in a small cache. This works pretty nicely and enables dense environment map placement, as at any moment of the time we have to store only a small subset of them.


GDC GAME DEVELOPERS CONFERENCE March 14-18, 2016 · Expo: March 16-18, 2016 #GDC16

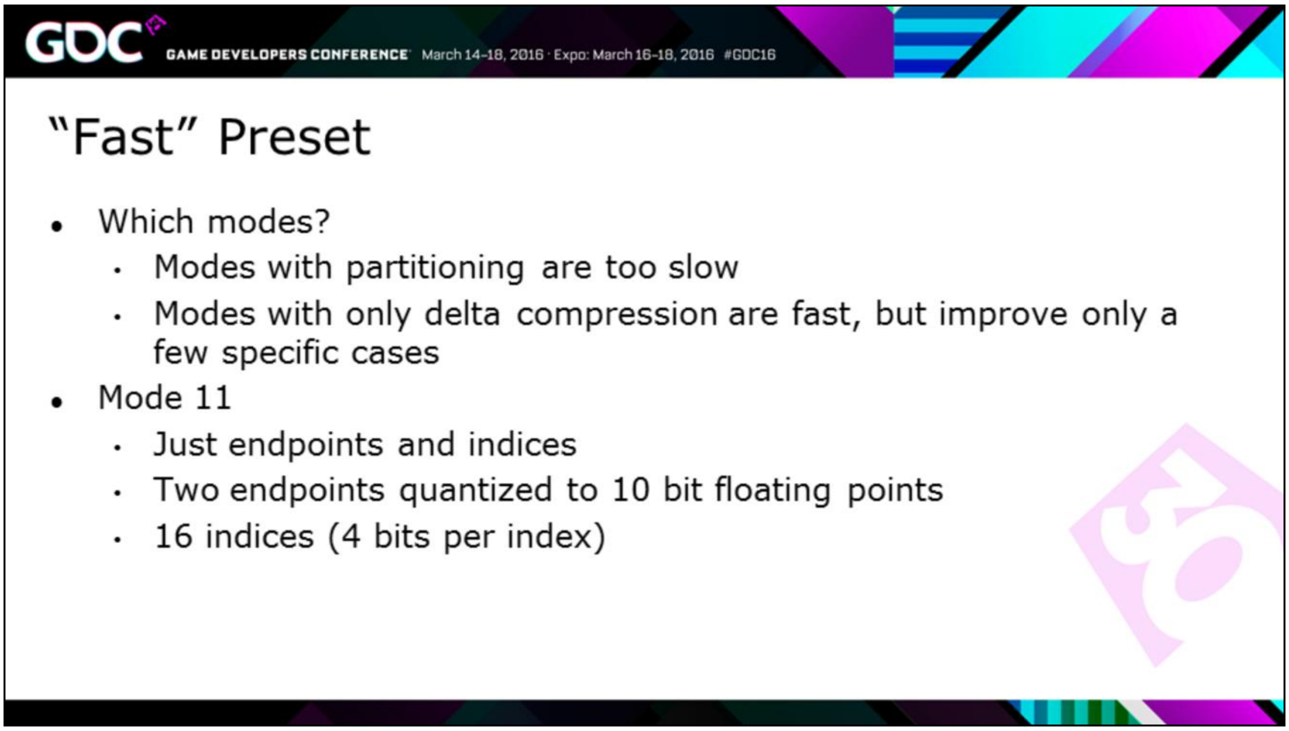
Real-Time Compression

- Performance or quality?
- "Fast"
  - Performance is crucial
  - Quality doesn't have to be top notch
- "Quality"
  - Quality is important
  - Compression can take a few ms

Now I'm going to preset two real-time compression algorithms with different tradeoffs between quality and performance.

The first one it's the "Fast" preset. "Fast" preset is designed for cases where performance is crucial, but the quality doesn't have to be top notch. It's good for things like environment maps, which we won't be looking at directly.

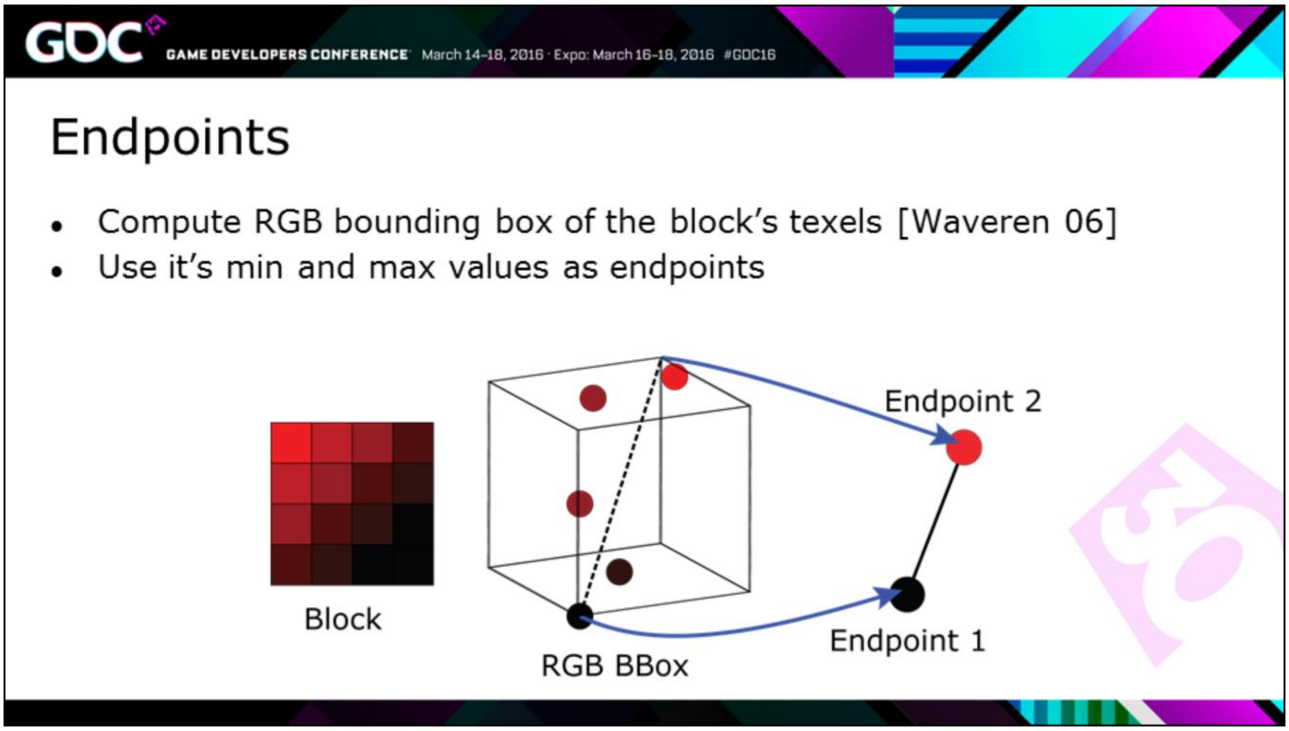
The second preset is the "Quality preset". "Quality" preset is designed for things like a skybox texture, where we want good quality and we can wait a few ms for the compression.


GDC GAME DEVELOPERS CONFERENCE March 14-18, 2016 · Expo: March 16-18, 2016 #GDC16

"Fast" Preset

- Which modes?
  - Modes with partitioning are too slow
  - Modes with only delta compression are fast, but improve only a few specific cases
- Mode 11
  - Just endpoints and indices
  - Two endpoints quantized to 10 bit floating points
  - 16 indices (4 bits per index)

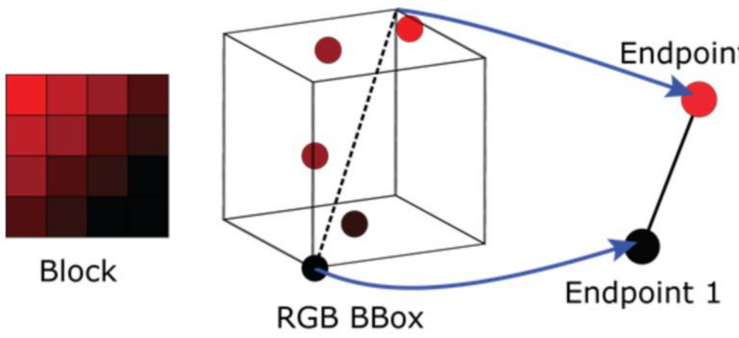
We will start from the "fast" preset. The first question is - which modes do we want to use for implementing this preset? Obviously modes with partitioning are too complicated and slow. We tried using delta compression modes. Those modes are designed for a specific use case – when colors in a block are similar. Delta compression won't improve the worst case and overall quality improvement wasn't worth the extra performance cost. We are left with only one mode – mode 11. Mode 11 uses only endpoints and indices. It has two endpoints stored as 10 bit quantized floating points and 16 indices stored using 4 bits per index.



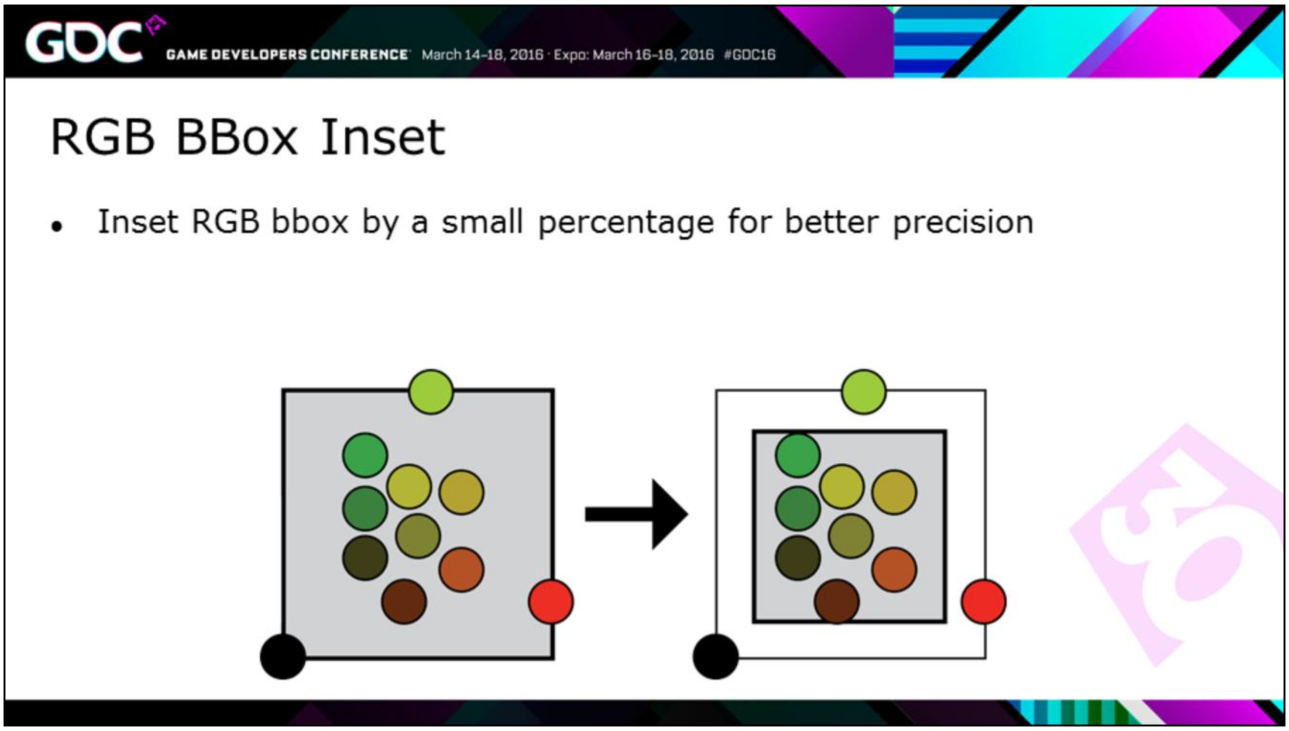
GDC GAME DEVELOPERS CONFERENCE March 14-18, 2016 · Expo: March 16-18, 2016 #GDC16

Endpoints

- Compute RGB bounding box of the block's texels [Waveren 06]
- Use it's min and max values as endpoints



Mode 11 is similar to previous BC formats, so we can use the classic BC real time compression algorithm by Van Waveren. It's very simple. We compute a RGB space bounding box of all colors in a block and use it's min and max corners as endpoints. It's also blazing fast, as it boils down to a few min/max instructions.

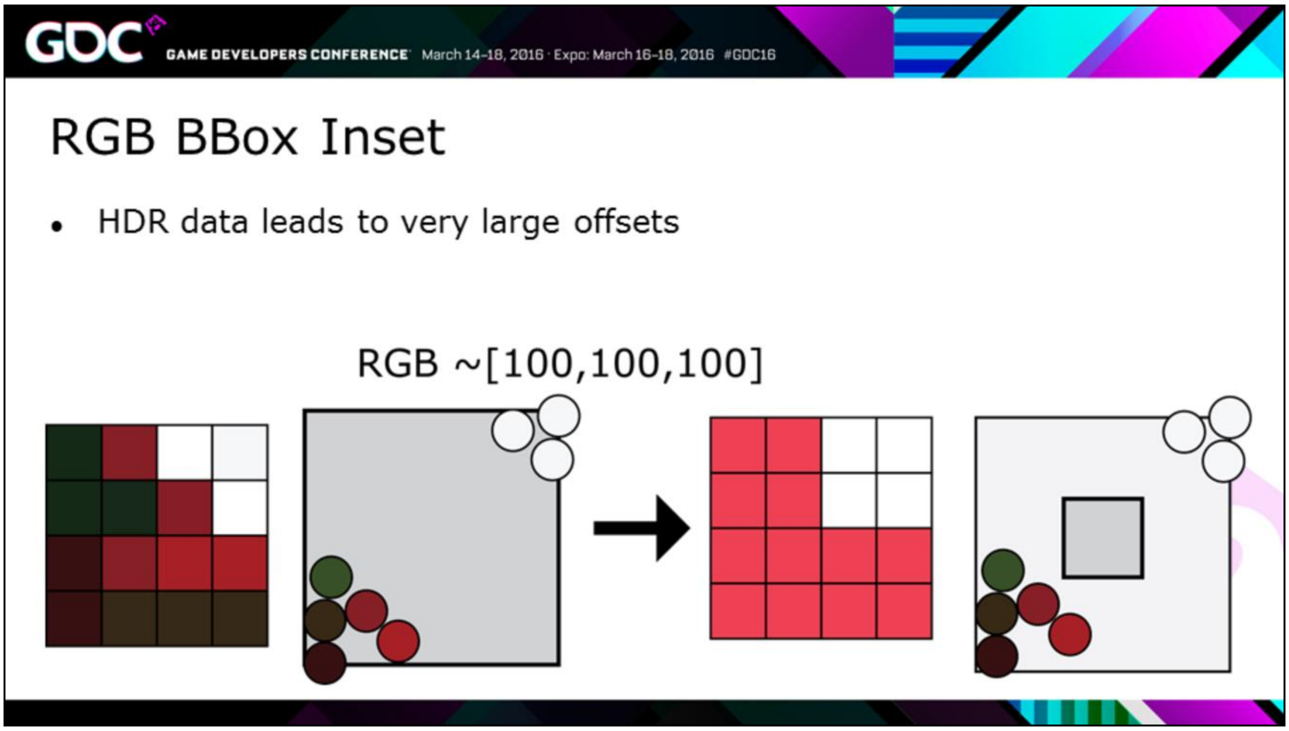


GDC GAME DEVELOPERS CONFERENCE March 14-18, 2016 · Expo: March 16-18, 2016 #GDC16

RGB BBox Inset

- Inset RGB bbox by a small percentage for better precision

The second step of this algorithm is to inset the resulting bounding box by a small percentage. For LDR data this step lowers compression error as most colors will be located inside the new bounding box.



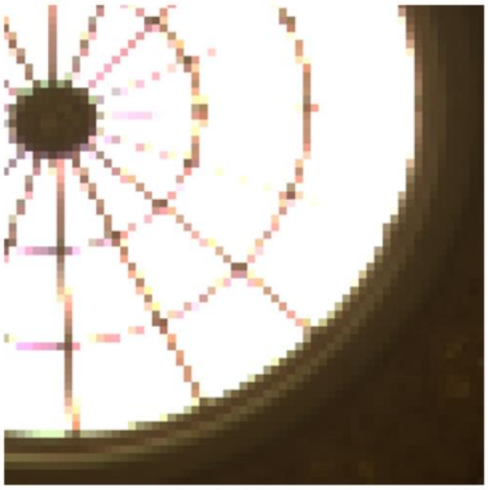
GDC GAME DEVELOPERS CONFERENCE March 14-18, 2016 · Expo: March 16-18, 2016 #GDC16

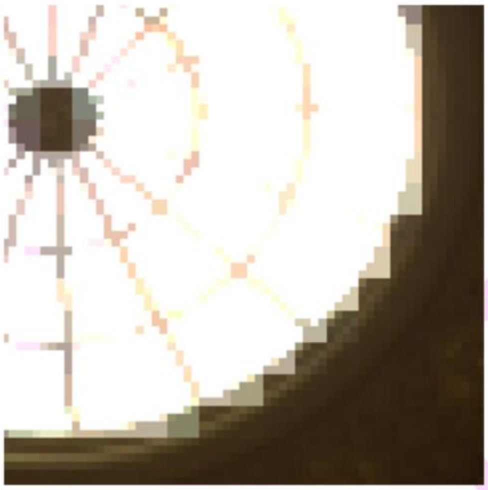
RGB BBox Inset

- HDR data leads to very large offsets

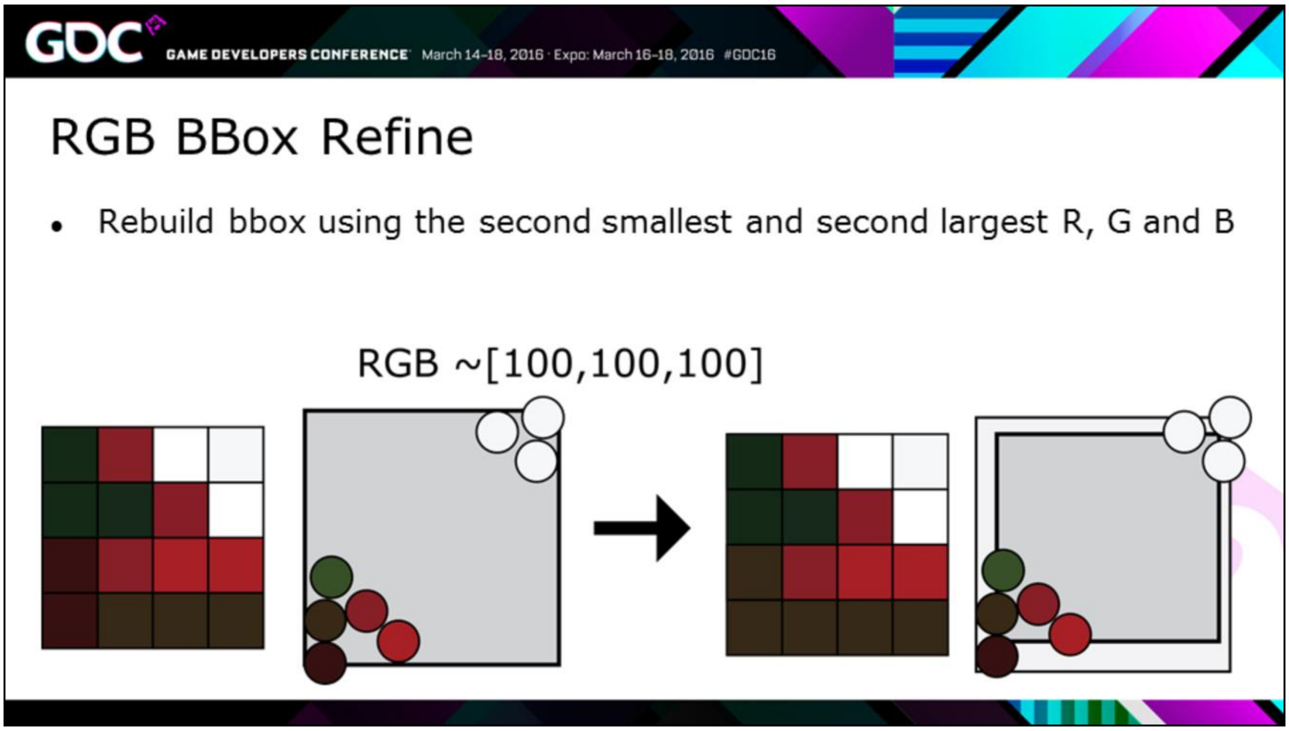
But in case of HDR data, we may have blocks with some dark texels and a some very bright ones. For those blocks we will get some enormous offsets and all the gradients inside this blocks will be lost.

RGB BBox Inset Results


Reference


Compressed

Here we can see how it looks like in practice. There are those nasty blocky artifacts near the edges of the window, where we have blocks containing dark and very bright texels.



GDC GAME DEVELOPERS CONFERENCE March 14-18, 2016 · Expo: March 16-18, 2016 #GDC16

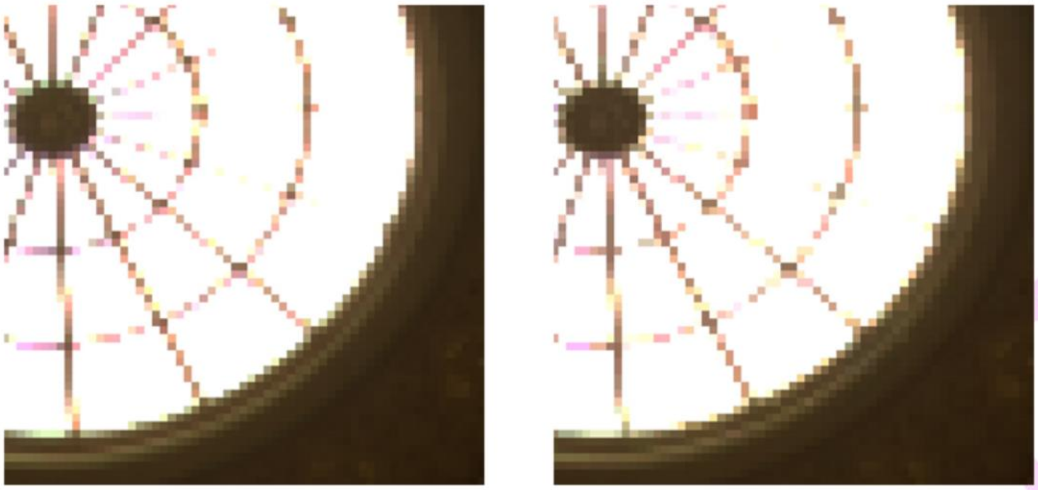
RGB BBox Refine

- Rebuild bbox using the second smallest and second largest R, G and B

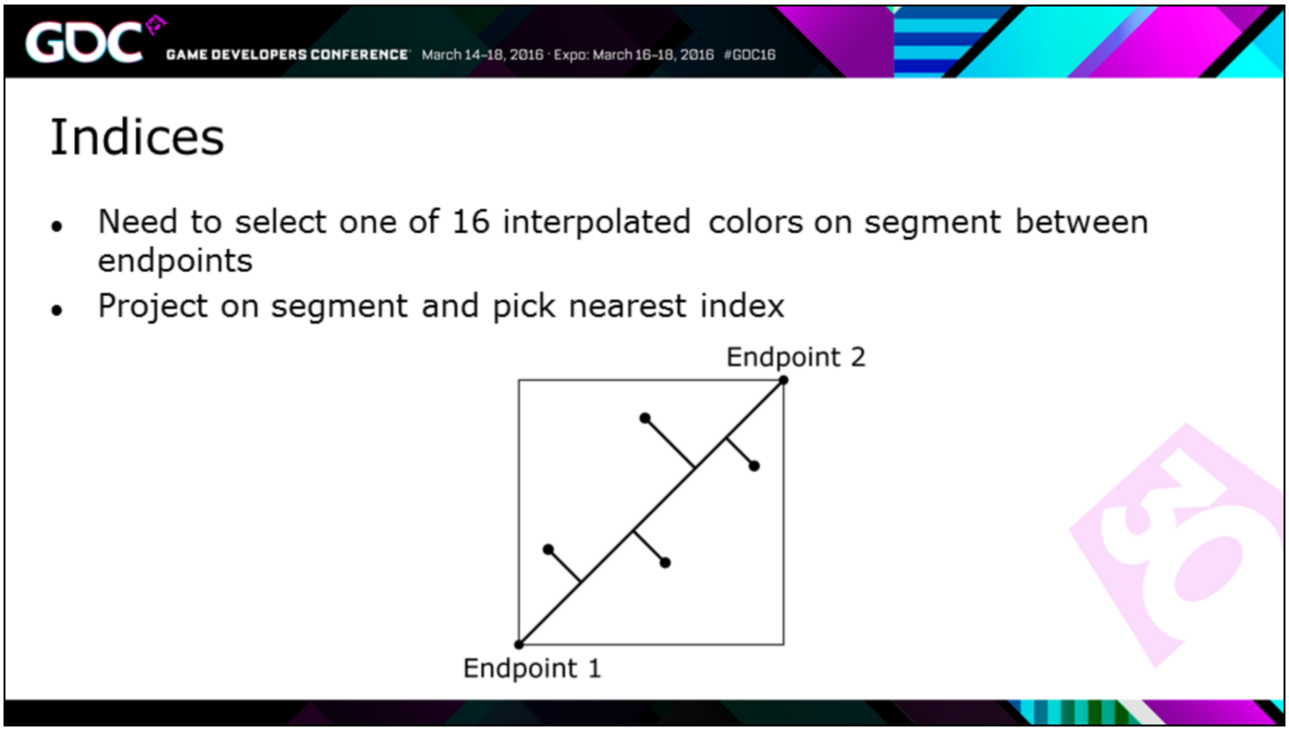
RGB ~[100,100,100]

Instead of insetting this bounding box by a small percentage, a better approach is to rebuild it again. This time using the second smallest and second largest RGB values. This way we can preserve all the gradients inside this block. Additionally, we limit maximum inset to a small percentage of a original bouding box’s size.

RGB BBox Refine Results


Reference Compressed

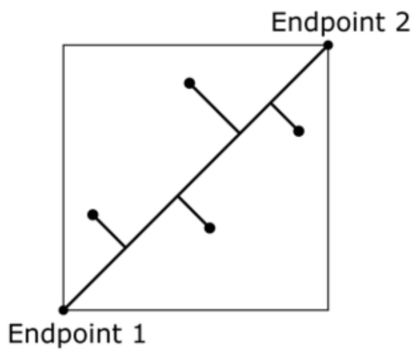
Here we can see how it works in practice. All those nasty blocky artifacts are gone.



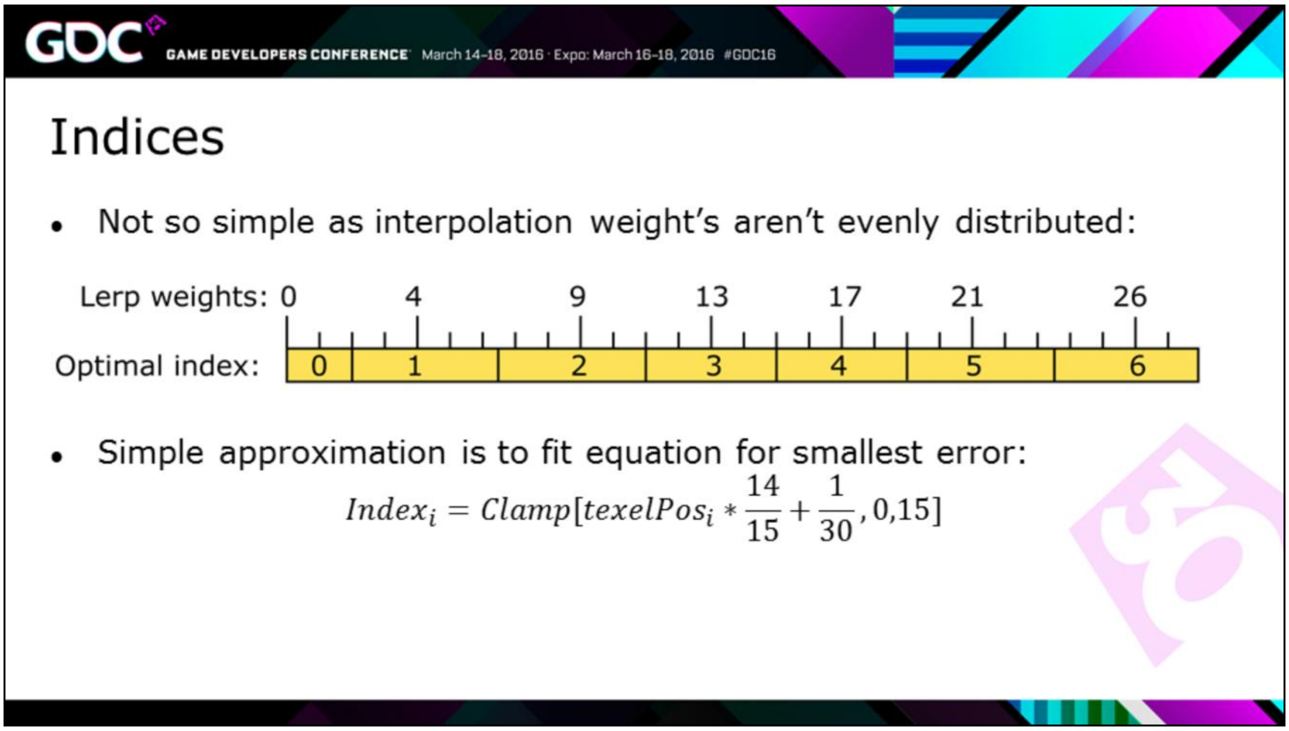
GDC GAME DEVELOPERS CONFERENCE March 14-18, 2016 · Expo: March 16-18, 2016 #GDC16

Indices

- Need to select one of 16 interpolated colors on segment between endpoints
- Project on segment and pick nearest index



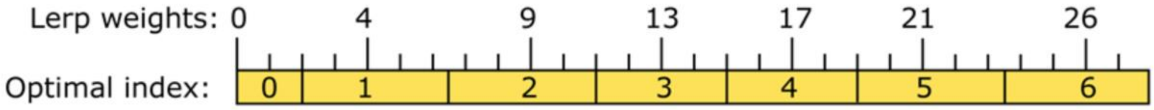
Next step is to compute indices for every texel in a current block. Usual solution was simply to use brute force and check all possible locations on the line segment for every texel. It was fine for formats like BC1, where indices were stored using 2 bits per index, so we had only 4 possible locations on the line segment. With mode 11 we are storing our indices using 4 bits per index, so we have 16 possible locations and using bruteforce would be too slow. Better approach is to take every texel’s color, project it on line segment and pick nearest index.



GAME DEVELOPERS CONFERENCE March 14-18, 2016 · Expo: March 16-18, 2016 #GDC16

Indices

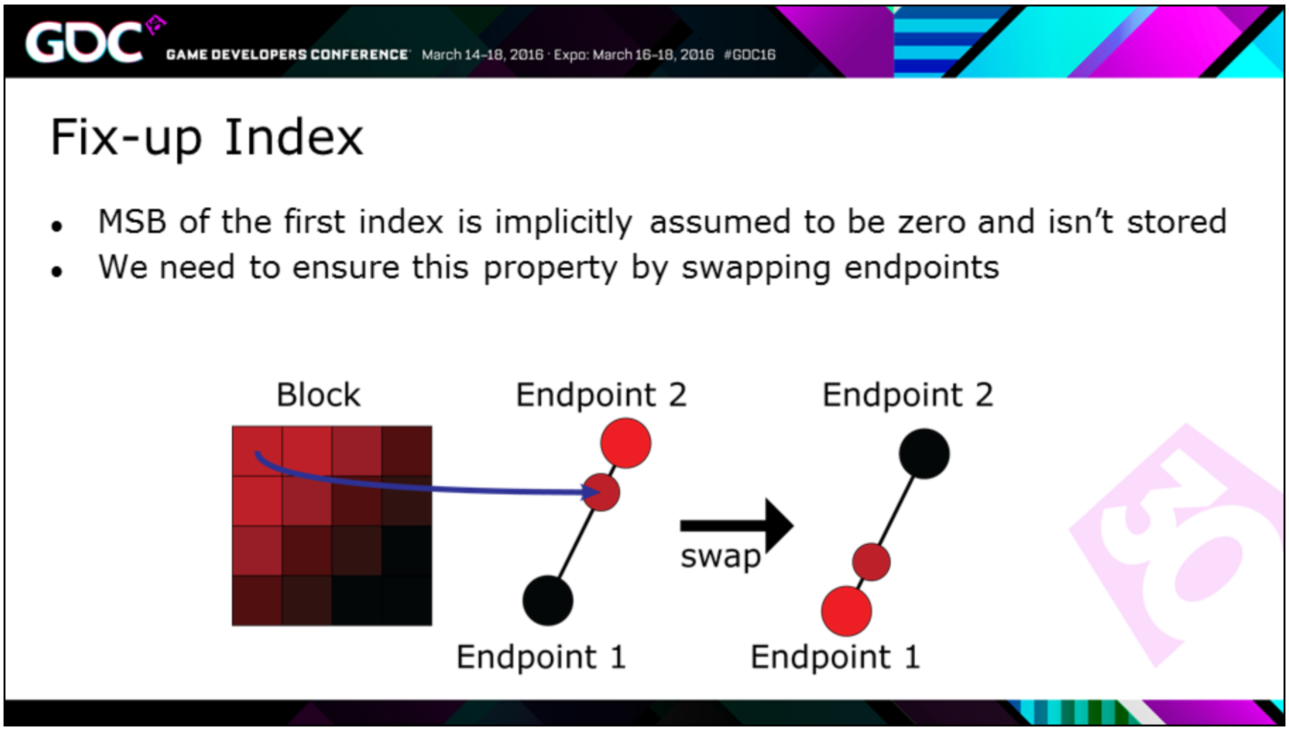
- Not so simple as interpolation weight's aren't evenly distributed:



- Simple approximation is to fit equation for smallest error:

$$Index_i = Clamp[texelPos_i * \frac{14}{15} + \frac{1}{30}, 0{,}15]$$

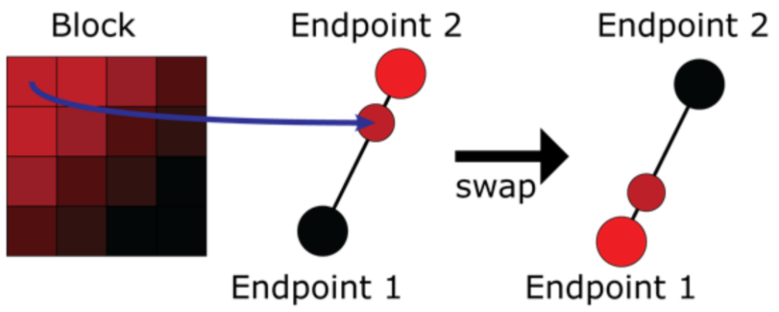
Unfortunately, this is not so simple as interpolation weight's aren't evenly distributed. We can see here that some buckets are smaller and some are larger. What we did here is that we just took a simple linear function and fitted it for smallest error. This results in a very simple approximation and introduced error is negligible.



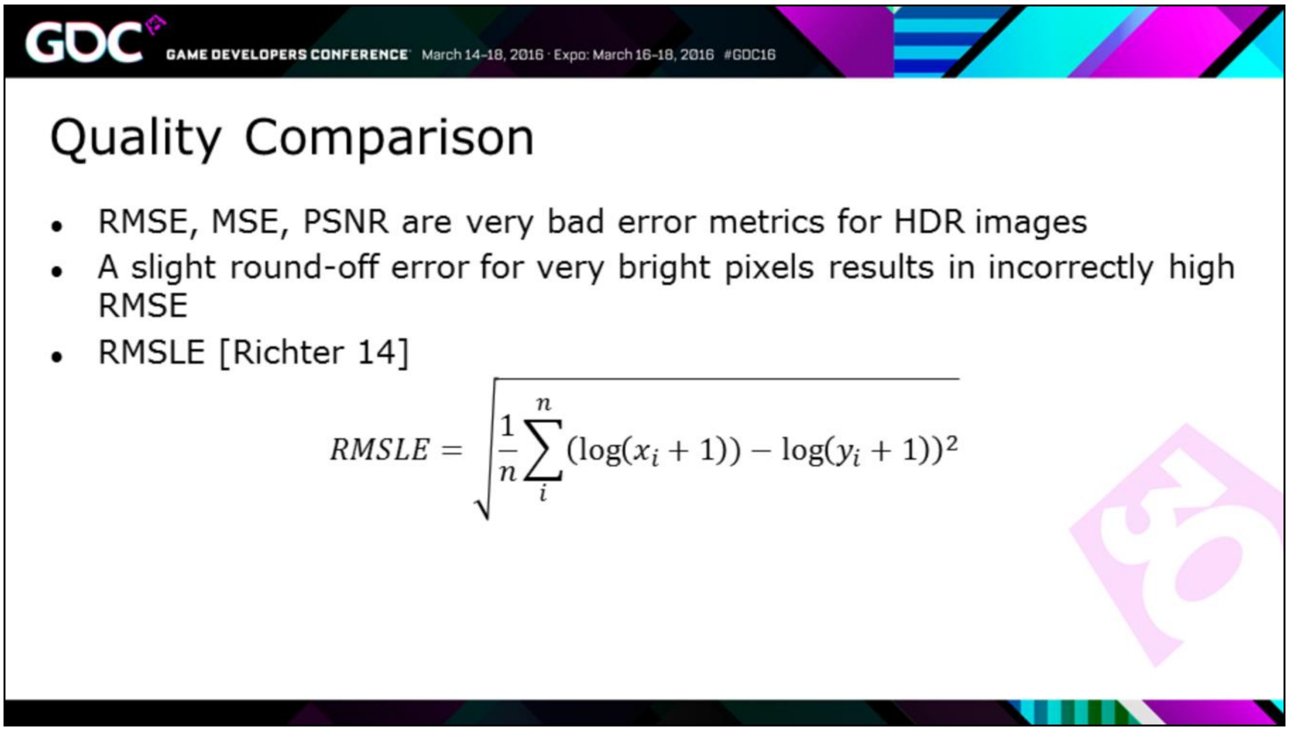
GDC GAME DEVELOPERS CONFERENCE March 14-18, 2016 · Expo: March 16-18, 2016 #GDC16

Fix-up Index

- MSB of the first index is implicitly assumed to be zero and isn't stored
- We need to ensure this property by swapping endpoints



There is one final twist. BC6H doesn't store the most significant bit of the first index in order to save one bit in block. This exploits the property of BC6H compression that endpoints can be always swapped and swapping the endpoints flips the bits of all indices. During compression we need check the first index and if it's too large then we need to swap endpoints and reverse the first index.

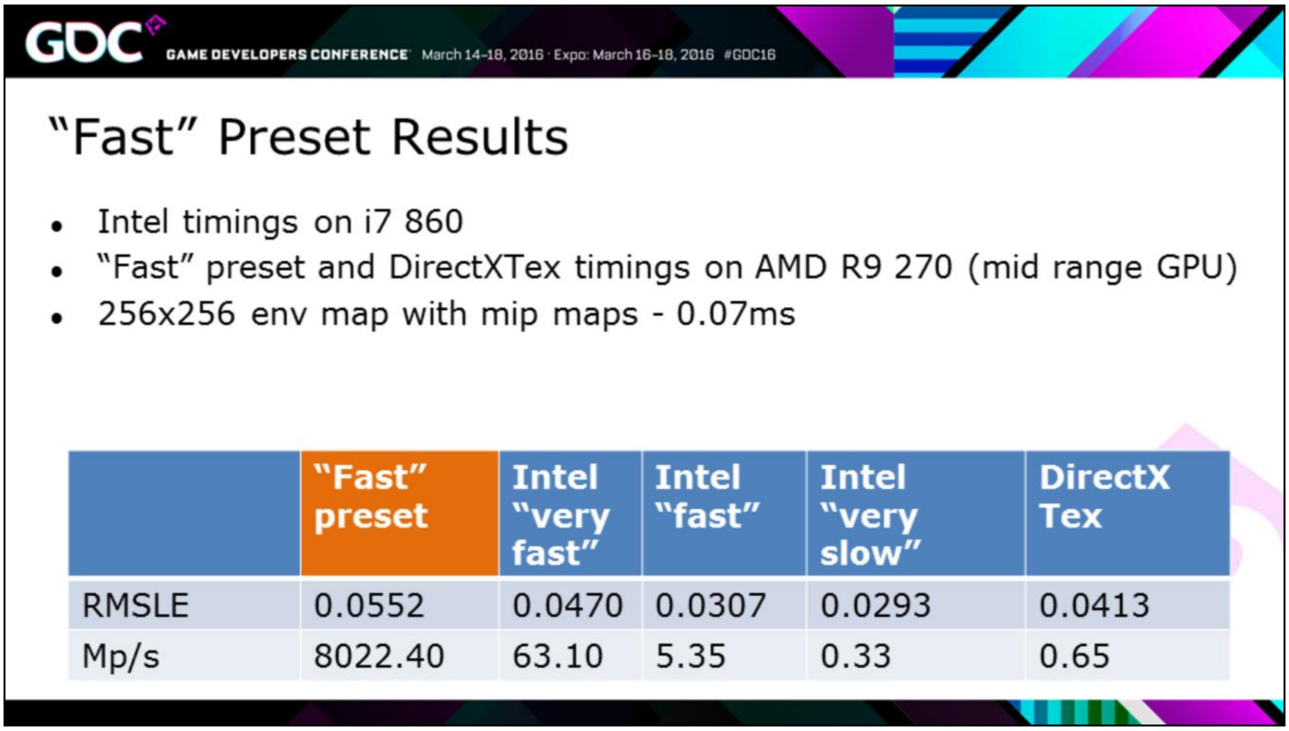

GDC GAME DEVELOPERS CONFERENCE March 14-18, 2016 · Expo: March 16-18, 2016 #GDC16

Quality Comparison

- RMSE, MSE, PSNR are very bad error metrics for HDR images
- A slight round-off error for very bright pixels results in incorrectly high RMSE
- RMSLE [Richter 14]

$$RMSLE = \sqrt{\frac{1}{n}\sum_{i}^{n}(\log(x_i + 1)) - \log(y_i + 1))^2}$$

One very important thing which I learned doing this hobby project is that standard error metrics like RMSE aren't very good for HDR data. We won't be looking at HDR data directly. It will always go through some kind of logarithmic tonemapping curve, so we want more precision in the blacks and less precision in the brights. If we would just use RMSE, then a slight round-off error for a few very bright pixels would result in a incorectly high RMSE value. A better approach is to user RMSLE, which is basically RMSE, but computed in a logarithmic space.


GDC GAME DEVELOPERS CONFERENCE March 14-18, 2016 · Expo: March 16-18, 2016 #GDC16

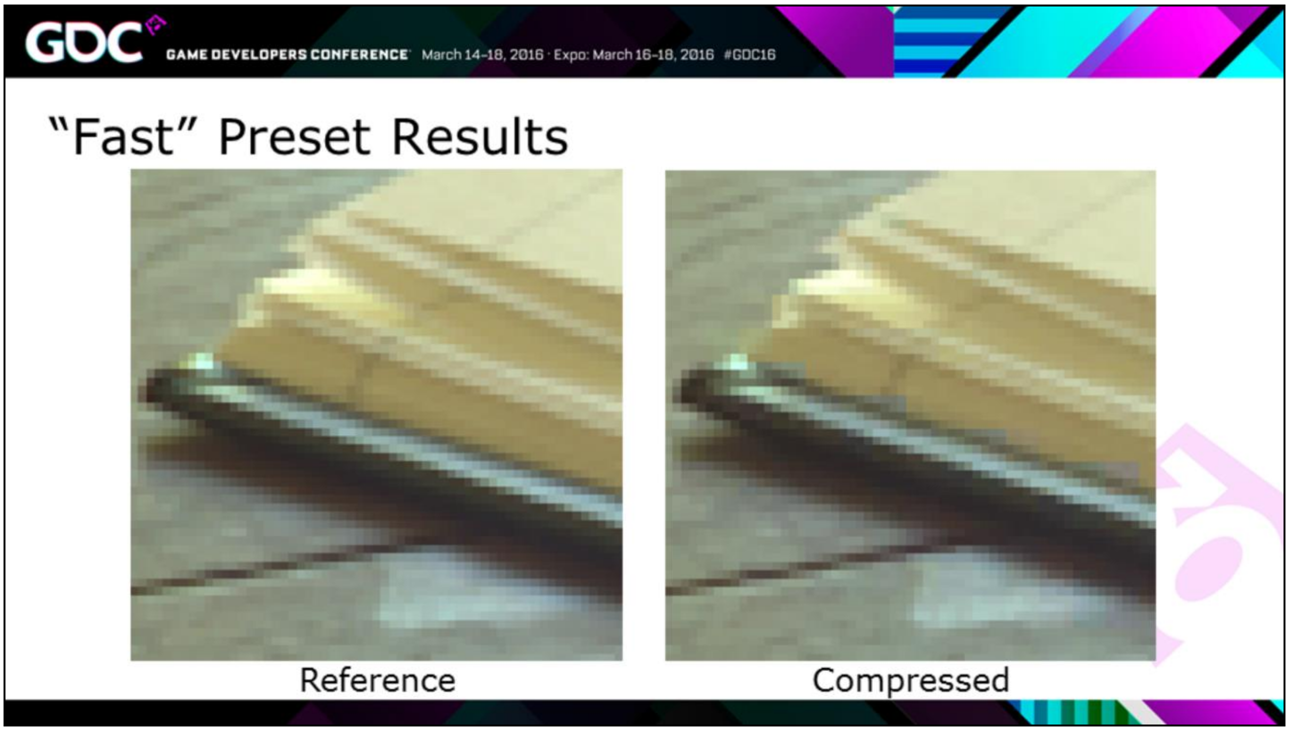
“Fast” Preset Results

- Intel timings on i7 860
- “Fast” preset and DirectXTex timings on AMD R9 270 (mid range GPU)
- 256x256 env map with mip maps - 0.07ms

	“Fast” preset	Intel “very fast”	Intel “fast”	Intel “very slow”	DirectX Tex
RMSLE	0.0552	0.0470	0.0307	0.0293	0.0413
Mp/s	8022.40	63.10	5.35	0.33	0.65

Here are some results for the “fast” preset. We compared “fast” preset with a CPU based Intel’s BC6H compressor and with a GPU based DirectXTex compressor. Intel’s compressor was run on first generation i7 and “fast” preset and DirectXTex were run on a mid range GPU, which is very similar to current gen consoles.

As you can see, the “fast preset” is quite fast. It can compress a standard 256x256 environment map with full mip chain in just 0.07ms, but the quality isn’t so good. It’s clearly worse than the quality we can get from the offline compressors.



GAME DEVELOPERS CONFERENCE March 14-18, 2016 · Expo: March 16-18, 2016 #GDC16

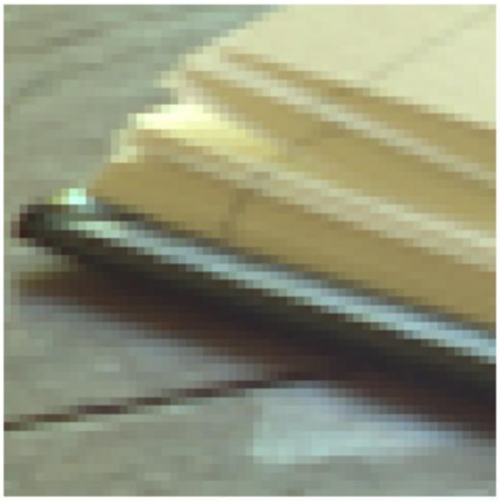
"Fast" Preset Results

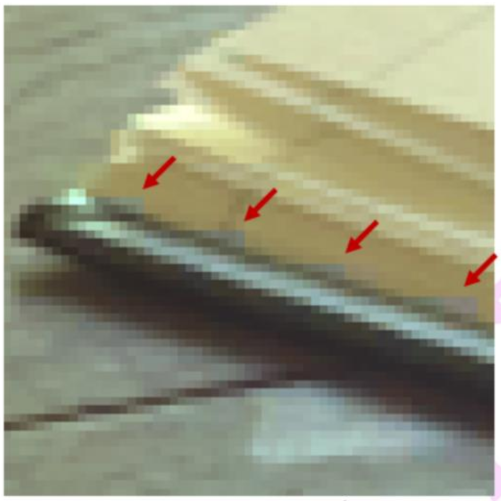
Reference

Compressed

Here we can see how it looks in practice. Images are pretty similar...

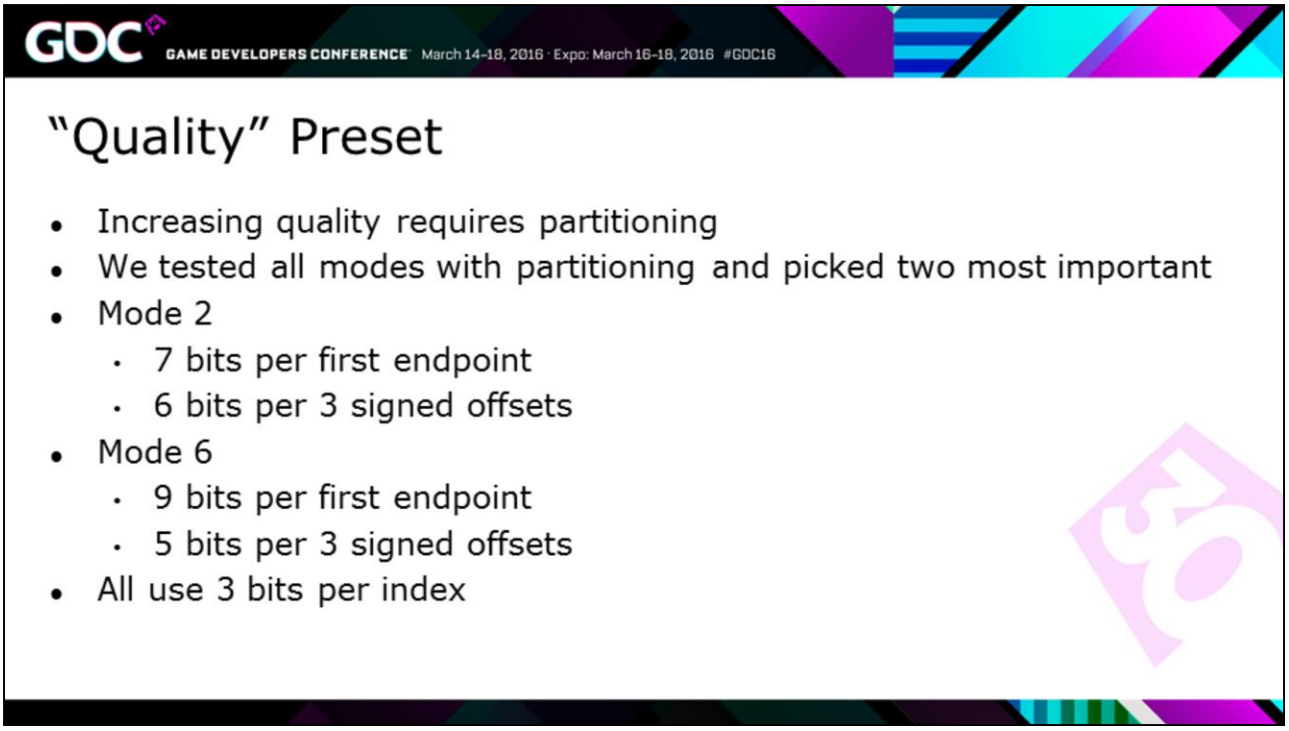
"Fast" Preset Results


Reference


Compressed

...but there are some blocky artifacts in places where we have more than 2 distinct colors in a block. It's hard to approximate well these blocks using just a single line segment.

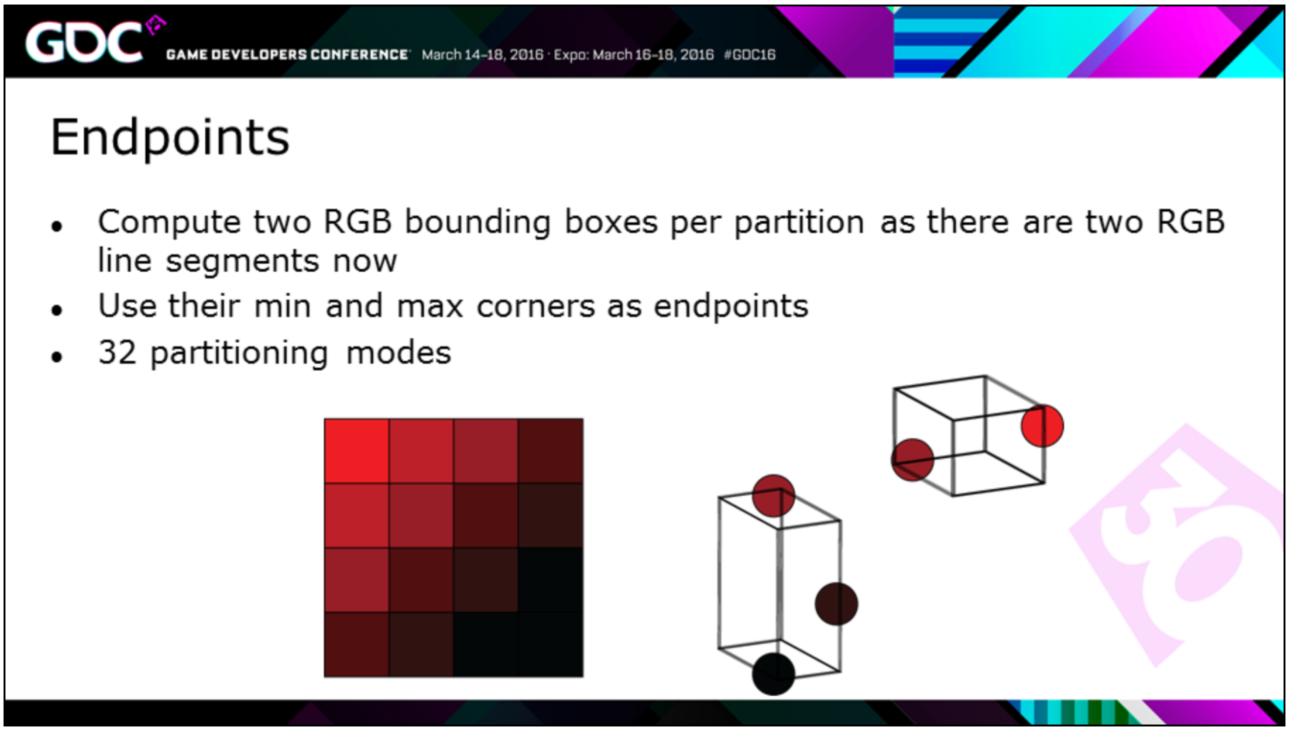
Quality is good enough for things like environment maps, which we won't be looking at directly, but it's not good enough for things directly displayed on the screen like a skybox texture.


GDC GAME DEVELOPERS CONFERENCE March 14-18, 2016 · Expo: March 16-18, 2016 #GDC16

“Quality” Preset

- Increasing quality requires partitioning
- We tested all modes with partitioning and picked two most important
- Mode 2
  - 7 bits per first endpoint
  - 6 bits per 3 signed offsets
- Mode 6
  - 9 bits per first endpoint
  - 5 bits per 3 signed offsets
- All use 3 bits per index

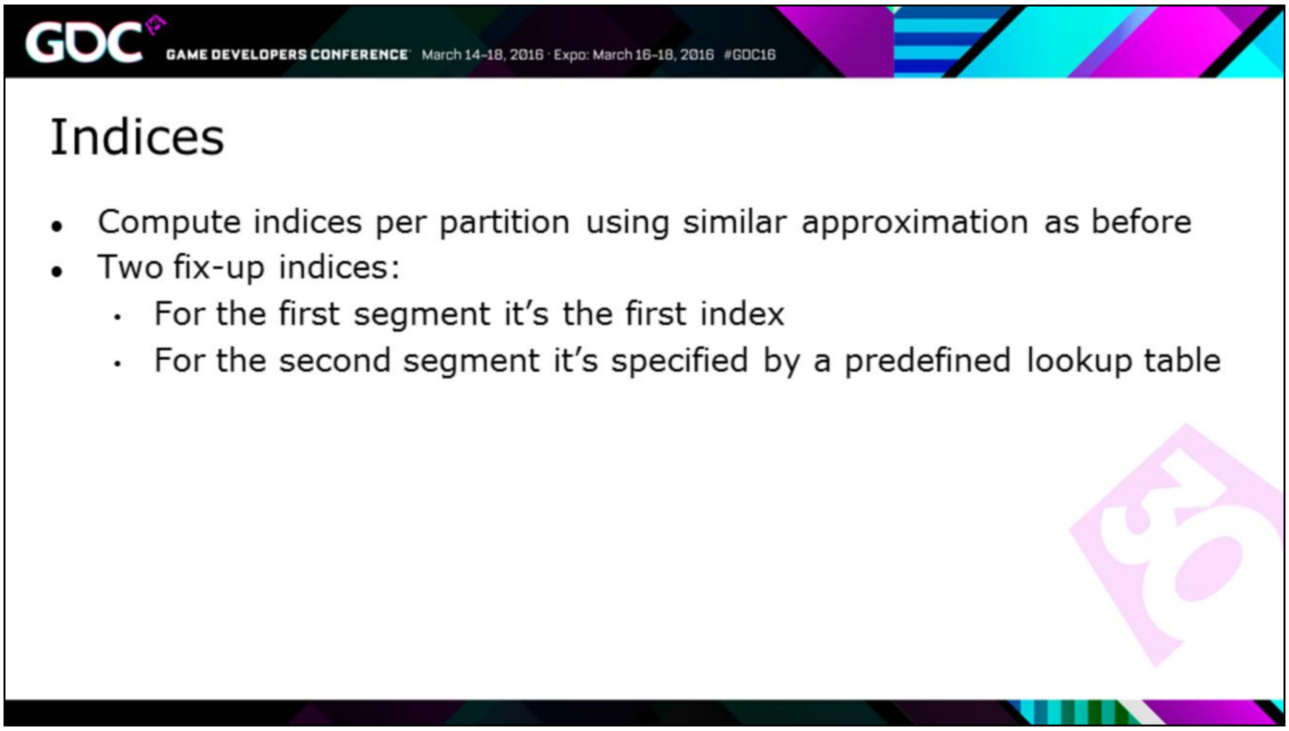
Let’s move to the “quality” preset. Obviously if we want to increase quality, we need to start using those partitioning modes, but we can’t just use them all, as it’s too slow even for offline compressors. We just did multiple tests with different images and modes and we picked two most important modes for the quality: mode 2 and mode 6. Both of them use partitioning and delta compression and store indices using 3 bits per index, but they have different tradeoffs between endpoint precision and offset vector precision.


GDC GAME DEVELOPERS CONFERENCE March 14-18, 2016 · Expo: March 16-18, 2016 #GDC16

Endpoints

- Compute two RGB bounding boxes per partition as there are two RGB line segments now
- Use their min and max corners as endpoints
- 32 partitioning modes

The first step is to compute endpoints. We can use the same algorithm as before, but this time we need to compute two bounding boxes per block as we have two line segments. Aditionally, we need to run this algorithm 32 times, as we have 32 different partitions.

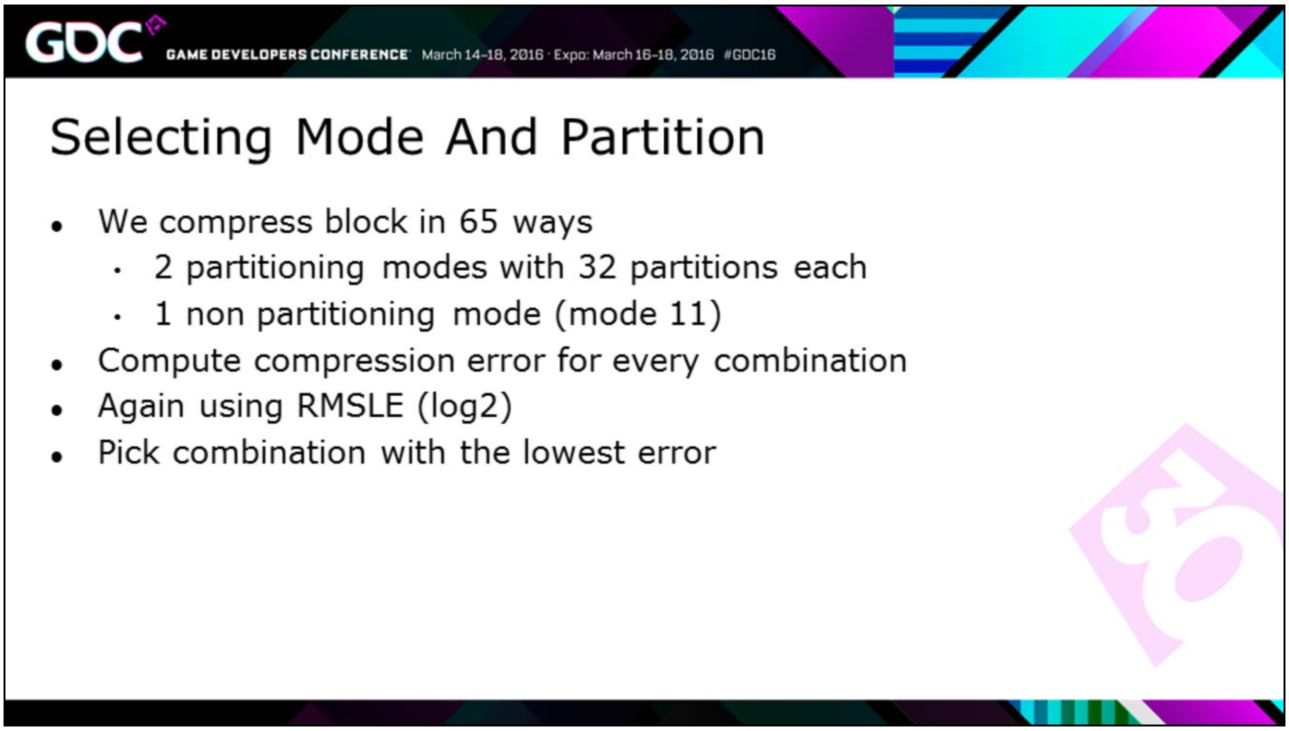

GDC GAME DEVELOPERS CONFERENCE March 14-18, 2016 · Expo: March 16-18, 2016 #GDC16

Indices

- Compute indices per partition using similar approximation as before
- Two fix-up indices:
  - For the first segment it's the first index
  - For the second segment it's specified by a predefined lookup table

The next step is to compute indices. Again, we can use very similar approximation as before, but this time we need to fit our linear function to a new weight interpolation table, as now we are storing indices using 3 bits per index and not 4 bits per index.

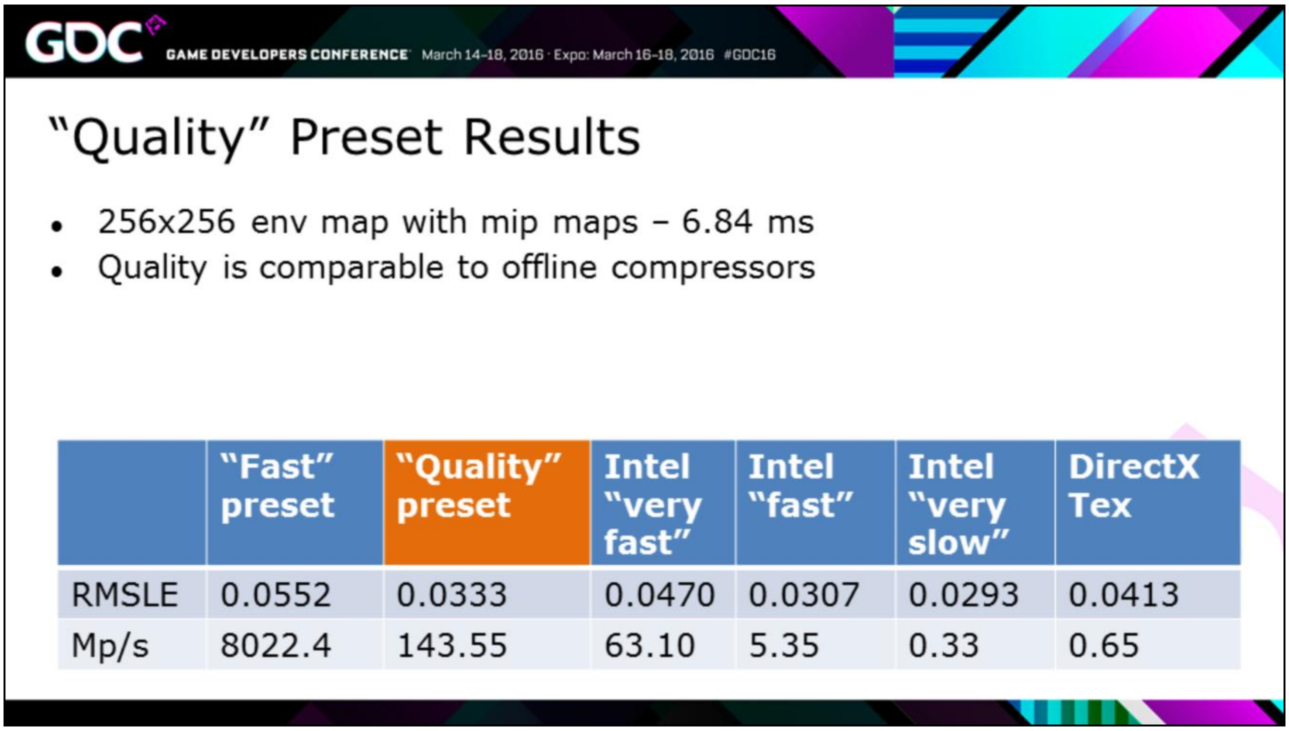
We need also to check the fix-up indices. There are two of those now, as there are two line segments. For the first segment it's just the first index. Location of the second fix-up index is specified by a predefined lookup table and depends on the partition number.


GDC GAME DEVELOPERS CONFERENCE March 14-18, 2016 · Expo: March 16-18, 2016 #GDC16

Selecting Mode And Partition

- We compress block in 65 ways
  - 2 partitioning modes with 32 partitions each
  - 1 non partitioning mode (mode 11)
- Compute compression error for every combination
- Again using RMSLE (log2)
- Pick combination with the lowest error

Finally, we can start compressing our blocks. First we compress using mode 11, which is the same as for the “fast” preset. Then we compress 32 times for mode 2 and 32 times for mode 6. Now we have 65 differently compressed block and need to pick one. We tried different approximations here, but the thing which worked in the end was a simple brute force. We decompress every block in shader, compute it’s RMSLE and pick the one with lowest error.

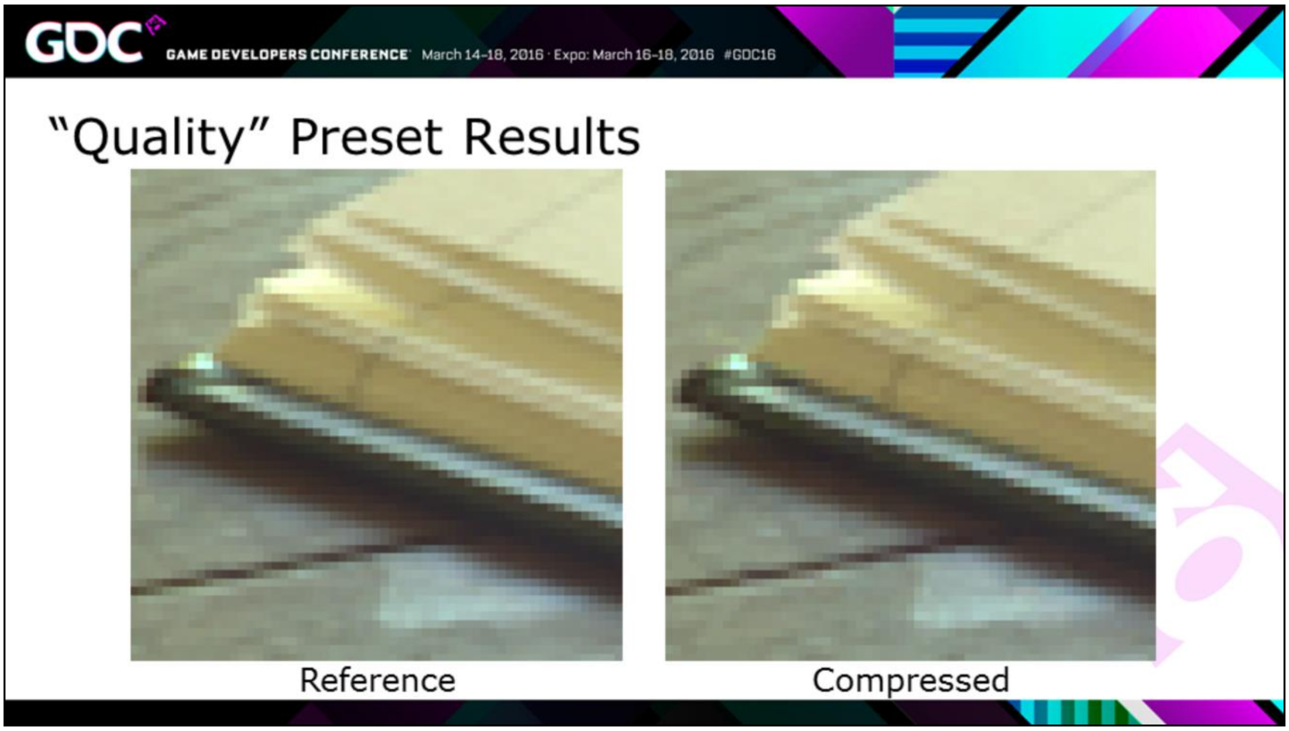

GAME DEVELOPERS CONFERENCE March 14-18, 2016 · Expo: March 16-18, 2016 #GDC16

"Quality" Preset Results

- 256x256 env map with mip maps – 6.84 ms
- Quality is comparable to offline compressors

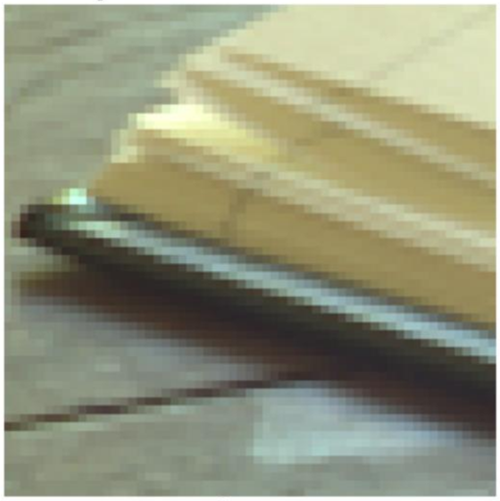
	"Fast" preset	"Quality" preset	Intel "very fast"	Intel "fast"	Intel "very slow"	DirectX Tex
RMSLE	0.0552	0.0333	0.0470	0.0307	0.0293	0.0413
Mp/s	8022.4	143.55	63.10	5.35	0.33	0.65

Here are some results for the "quality" preset. It's much slower than the "fast" preset. Now it takes almost 7ms for compressing a standard environment map. Still it's much faster than offline compressors and the quality is quite nice. It's better than the quality we can get from DirectXTex and it's mainly because DirectXTex optimizes for RMSE, which is bad idea for compressing HDR data. The quality of the "quality" preset is really comparable with the quality we can get from Intel's BC6H compressor.

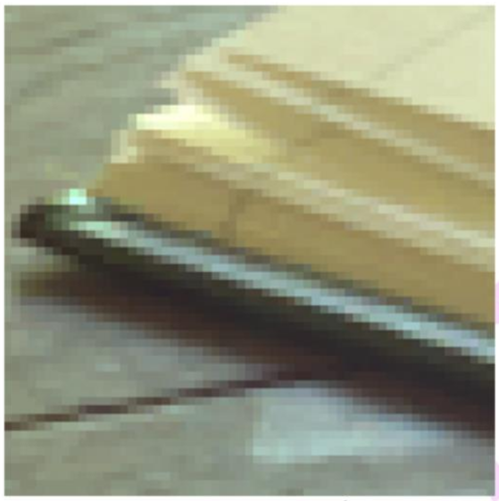


GDC GAME DEVELOPERS CONFERENCE March 14-18, 2016 · Expo: March 16-18, 2016 #GDC16

“Quality” Preset Results

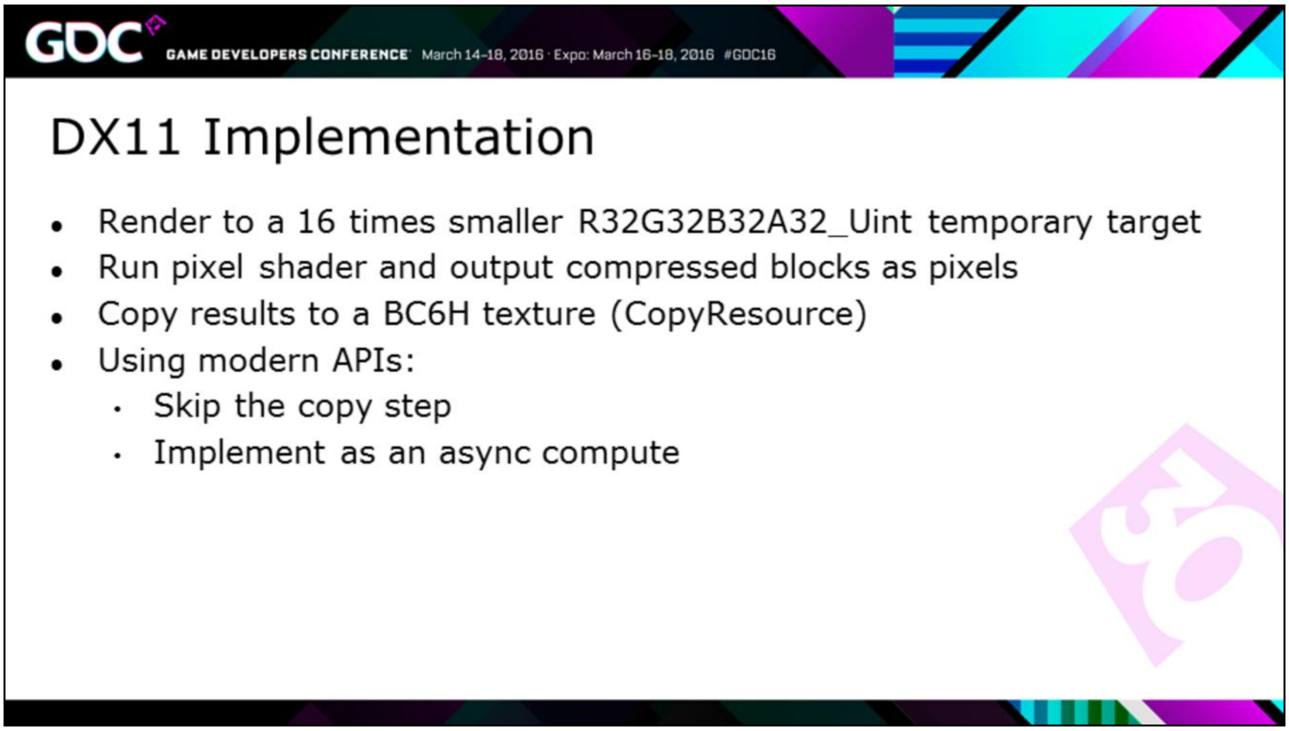


Reference



Compressed

Here we can see how it looks in practice. Images are pretty similar and it’s hard to notice any compression artifacts.

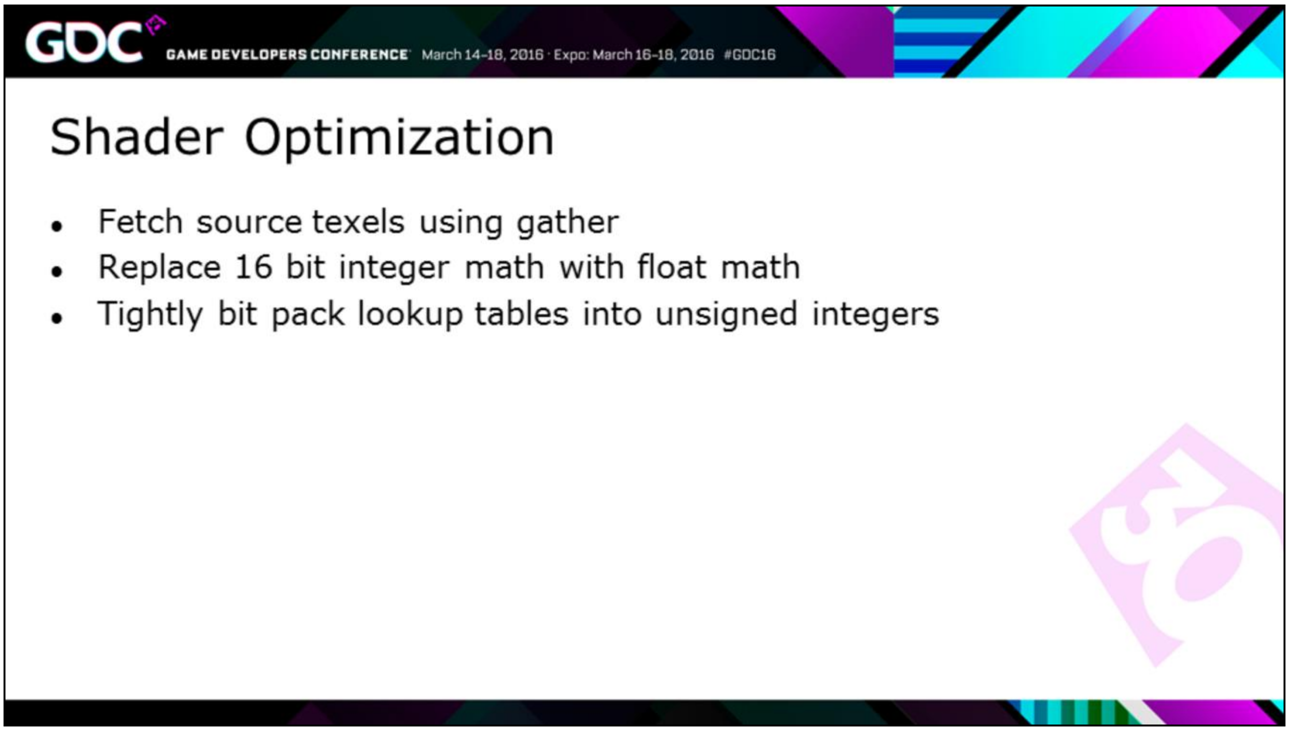

GDC GAME DEVELOPERS CONFERENCE March 14-18, 2016 · Expo: March 16-18, 2016 #GDC16

DX11 Implementation

- Render to a 16 times smaller R32G32B32A32_Uint temporary target
- Run pixel shader and output compressed blocks as pixels
- Copy results to a BC6H texture (CopyResource)
- Using modern APIs:
  - Skip the copy step
  - Implement as an async compute

Now I'm going to talk a bit about the implementation using DX11. It's pretty straightforward. First, we create a temporary RGBA 32bit unsigned integer render target, which is 16 times smaller than the source texture. Then we run a pixel shader, which reads 16 source texels, compressed them and outputs compressed block to that temporary render target. Finally, we need to copy data from this temporary render target to destination BC6H texture using CopyResource function.

With modern APIs, like we have on consoles, we can skip this last copy step and just render directly into a destination BC6H texture. We can also implement this algorithm using async compute and run it in parallel with GBuffer rendering or shadow map rasterization.


GDC GAME DEVELOPERS CONFERENCE March 14-18, 2016 · Expo: March 16-18, 2016 #GDC16

Shader Optimization

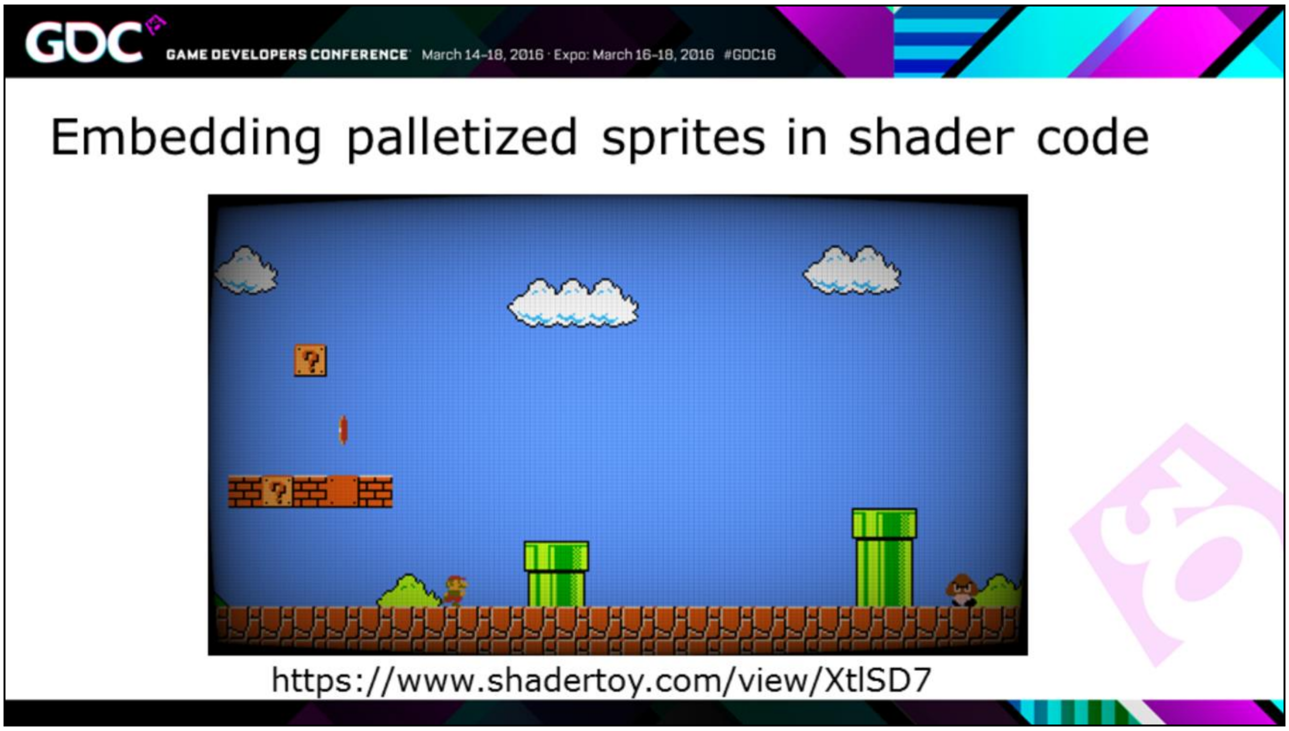
- Fetch source texels using gather
- Replace 16 bit integer math with float math
- Tightly bit pack lookup tables into unsigned integers

Here are some tips about shader optimization.

The first thing is that we aren't interested in source alpha, so instead of doing 16 sample instructions we can just do 12 gather instructions.

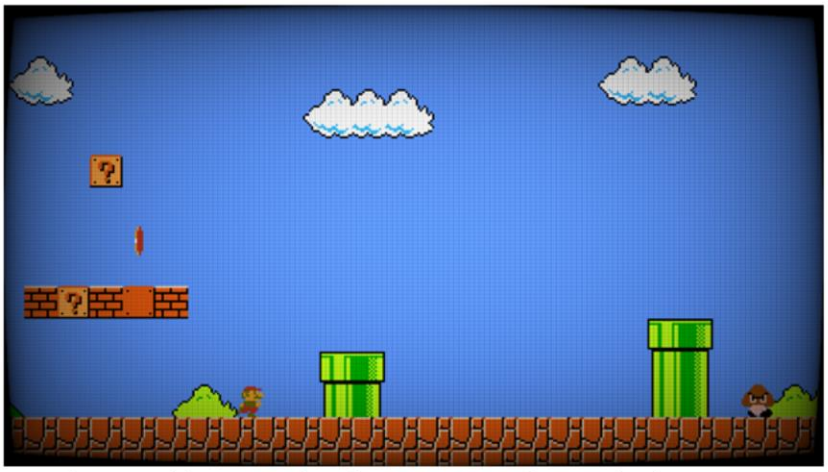
Second thing is that we are using here a lot of 16 bit integer math and integer math is pretty slow on modern GPUs. Especially, operations like integer divides and multiplies. Fortunately, floating point has 23 bit mantissa and 1 bit for sign, so 16 bit integer can be represented without any error and we can replace all the integer math with floating point math.

Finally there are some lookup tables. There is one which assigned every texel in a block to one of two line segments, depending on the partition number. The second one specifies the location of the fix-up index for the second line segment and again it depends on the partition number. We can tightly bit pack those lookup tables into unsigned integers and use some masks and shifts for lookup. This way we can replace a bunch of conditional instructions with simple math.



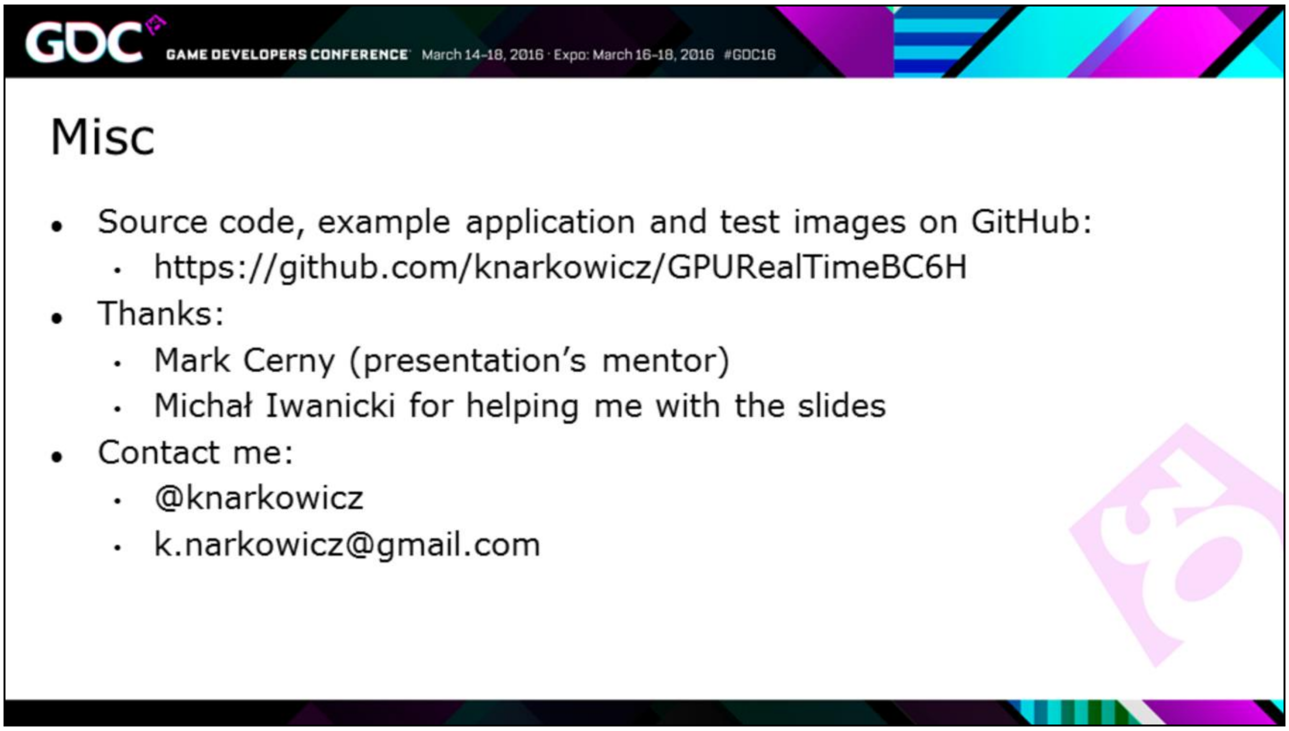
GDC GAME DEVELOPERS CONFERENCE March 14-18, 2016 · Expo: March 16-18, 2016 #GDC16

Embedding palletized sprites in shader code



https://www.shadertoy.com/view/XtlSD7

By the way it also has a less boring use case, as it's a great way for embedding palletized sprites and textures directly into shader code. It's great for things like ShaderToy, where we can't use external textures.


GDC GAME DEVELOPERS CONFERENCE March 14-18, 2016 · Expo: March 16-18, 2016 #GDC16

Misc

- Source code, example application and test images on GitHub:
  - https://github.com/knarkowicz/GPURealTimeBC6H
- Thanks:
  - Mark Cerny (presentation's mentor)
  - Michał Iwanicki for helping me with the slides
- Contact me:
  - @knarkowicz
  - k.narkowicz@gmail.com

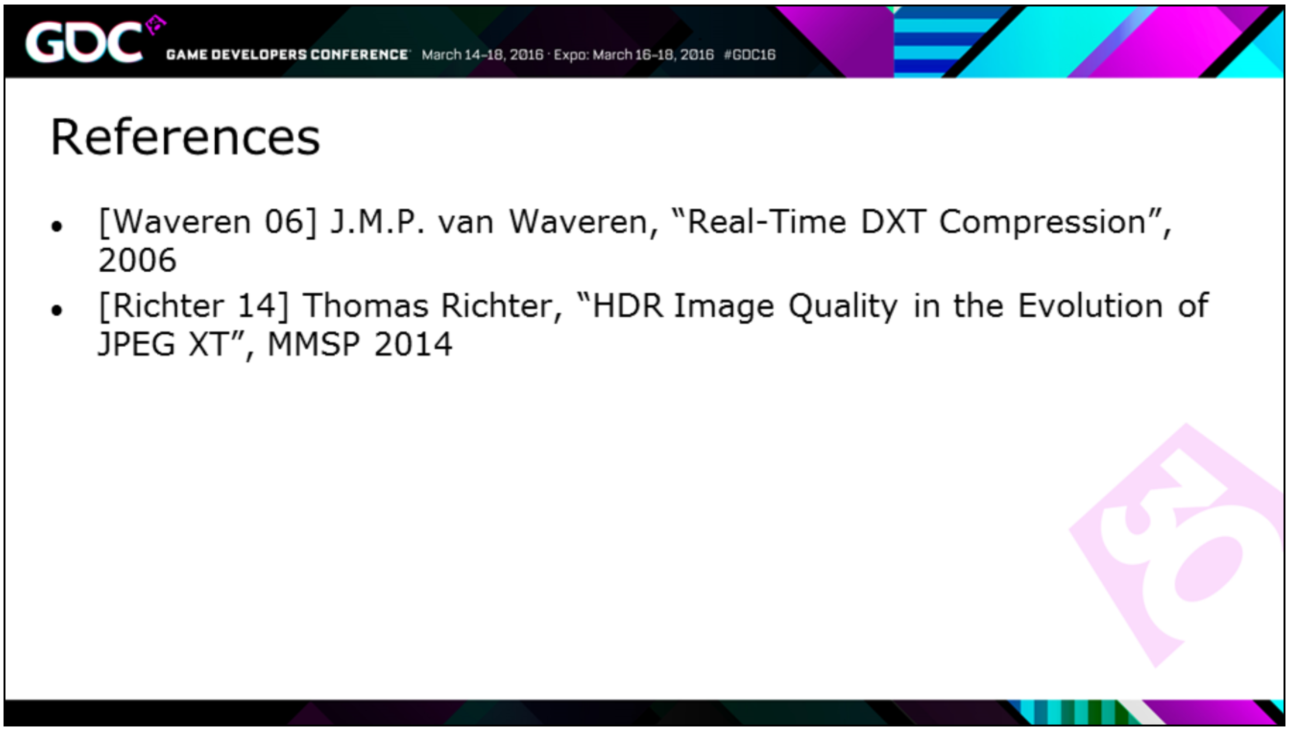
You can find full source code with all the gritty details, example application and test images on the GitHub.

I want also to thank Mark Cerny, who was the mentor for this presentation and Michal Iwanicki for helping me with the slides.

You can contact me using twitter or email.

Thank you for your attention.

Questions?



References

- [Waveren 06] J.M.P. van Waveren, "Real-Time DXT Compression", 2006
- [Richter 14] Thomas Richter, "HDR Image Quality in the Evolution of JPEG XT", MMSP 2014