Real-Time BC6H Compression on GPU

Krzysztof Narkowicz
Lead Engine Programmer
Flying Wild Hog

Introduction

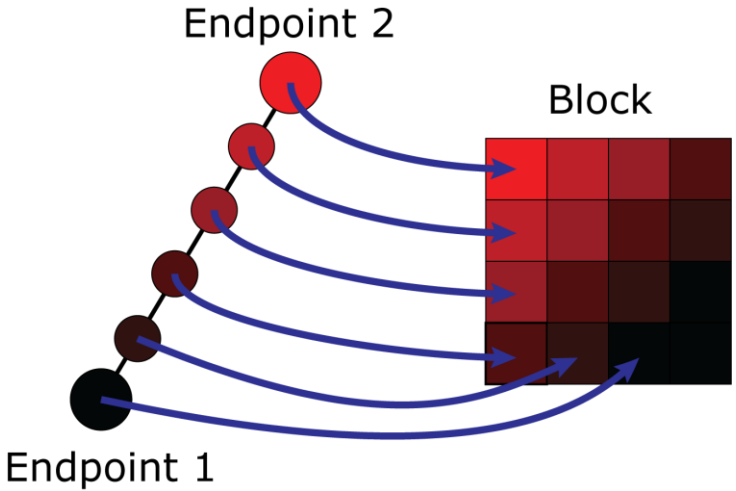
- BC6H is lossy block based compression designed for FP16 HDR textures
- Hardware supported since DX11 and current-gen consoles
- Fixed size 4x4 texel blocks
- No alpha
- 6:1 compression ratio (8bits per texel)
- Great replacement of hacky encodings like RGBE, RGBM, RGBK...

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

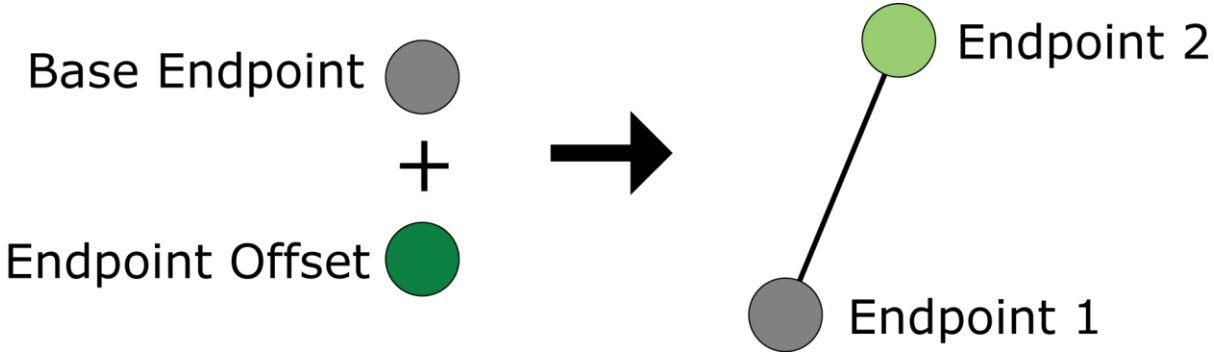
BC6H Endpoints And Indices

- 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



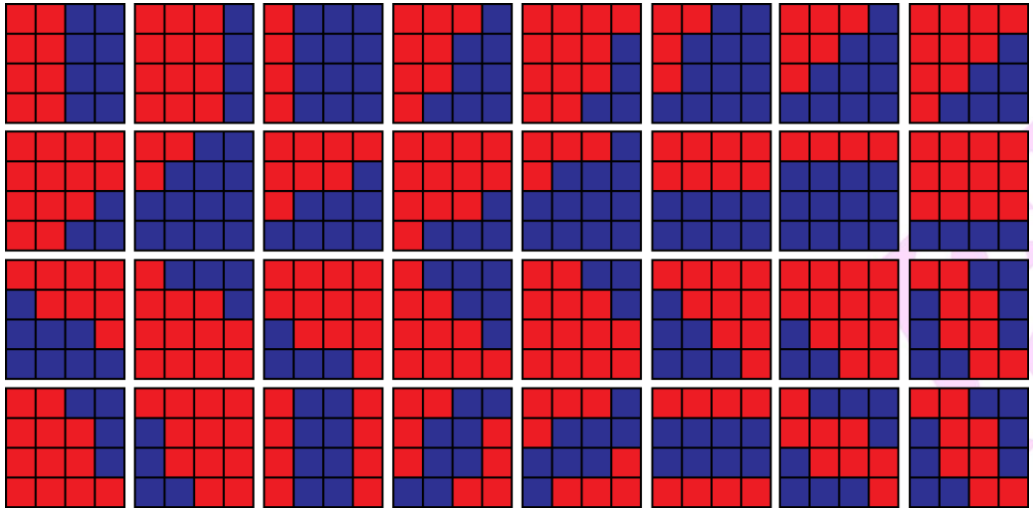
BC6H Delta Compression

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



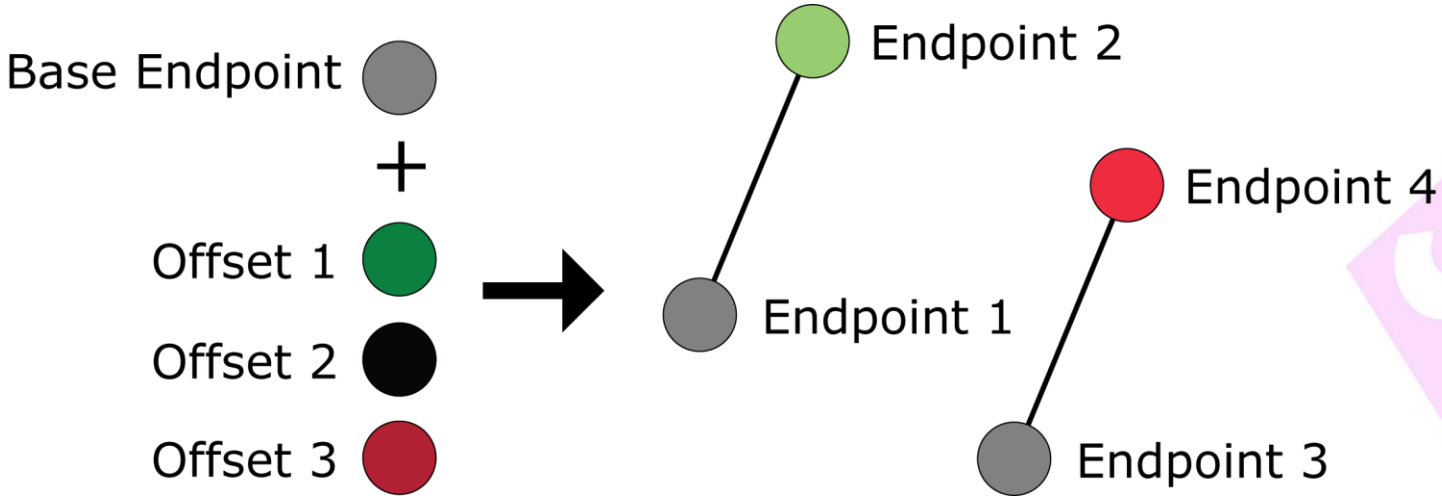
BC6H Partitioning

- 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



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



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

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

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

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

Real-Time Compression

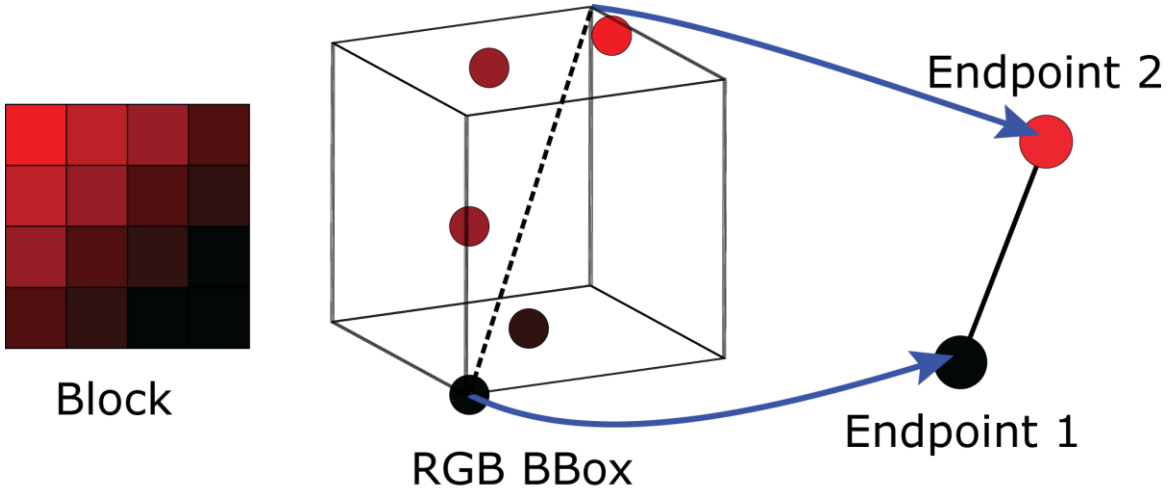
- Performance or quality?
- "Fast"
  - Performance is crucial
  - Quality doesn't have to be top notch
- "Quality"
  - Quality is important
  - Compression can take a few ms

“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)

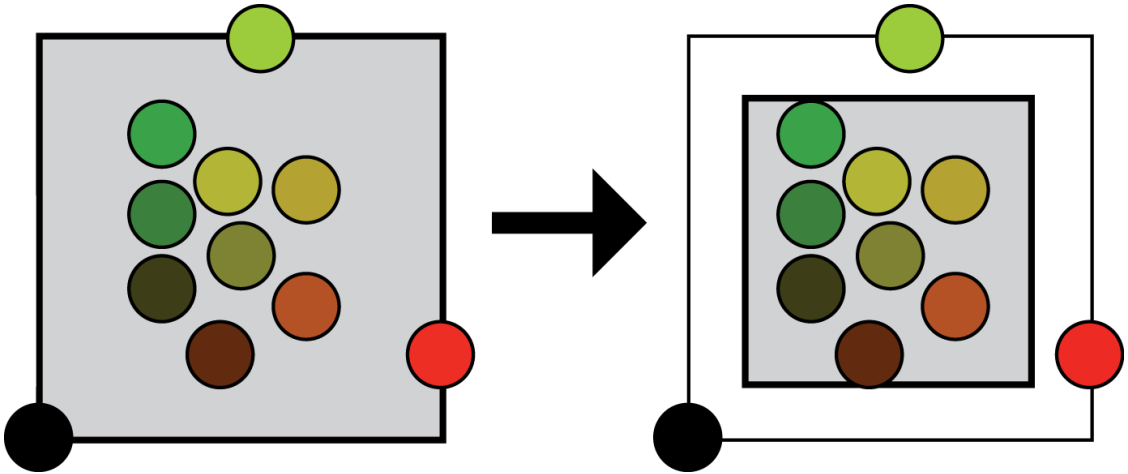
Endpoints

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



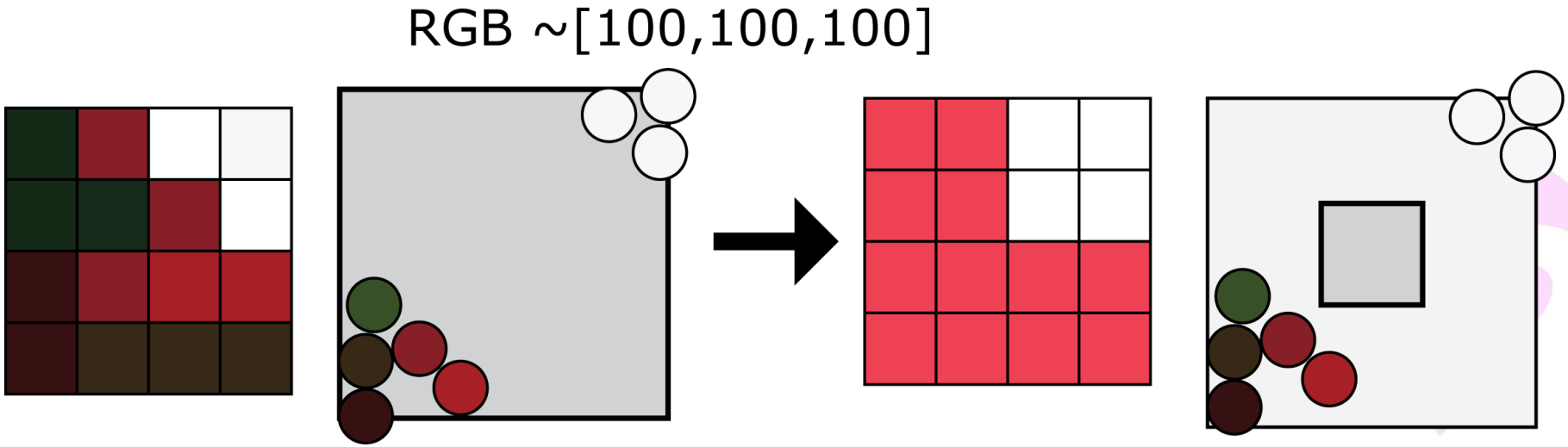
RGB BBox Inset

- Inset RGB bbox by a small percentage for better precision

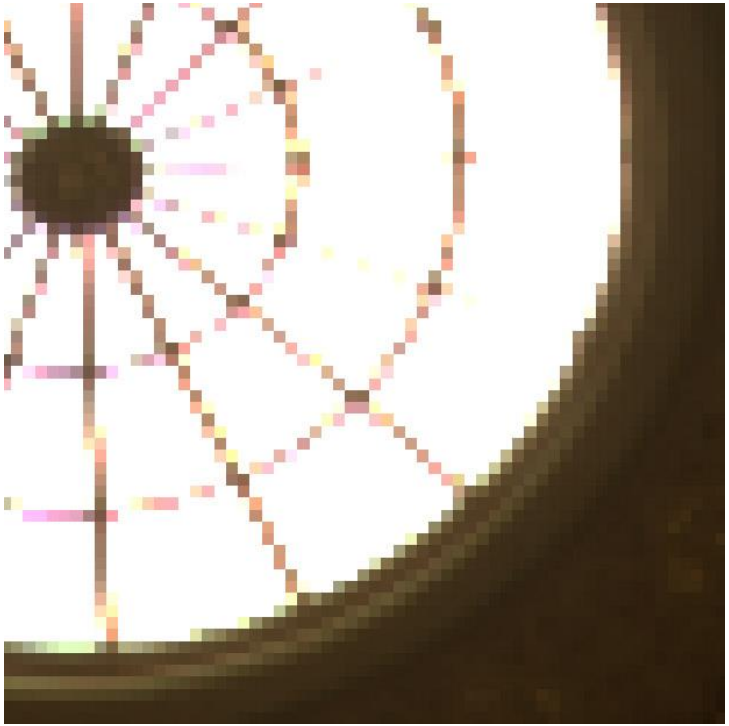


RGB BBox Inset

- HDR data leads to very large offsets



RGB BBox Inset Results

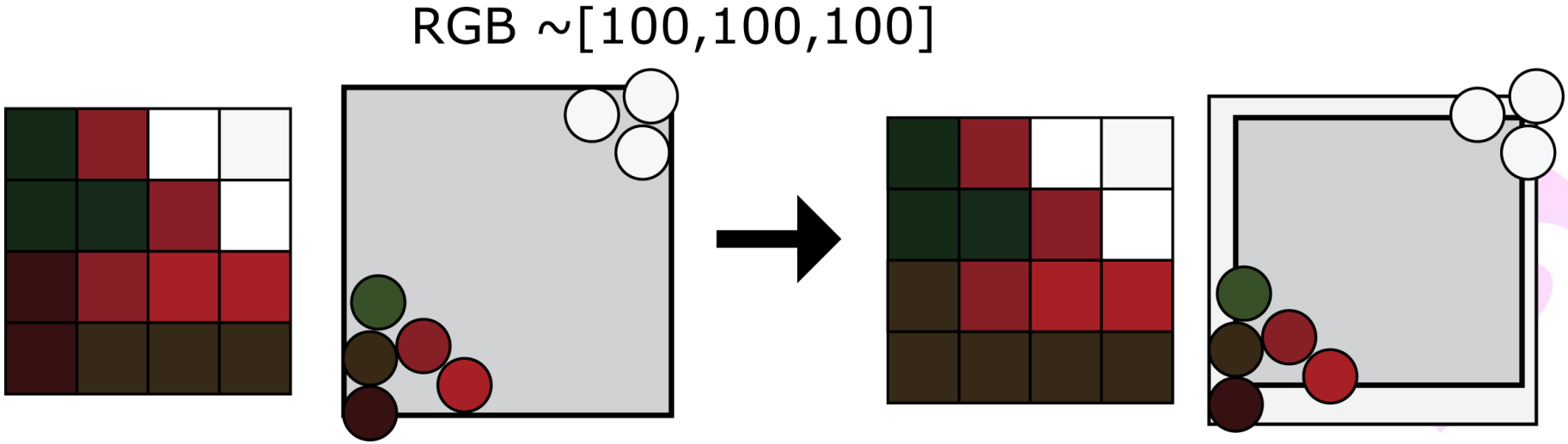


Reference

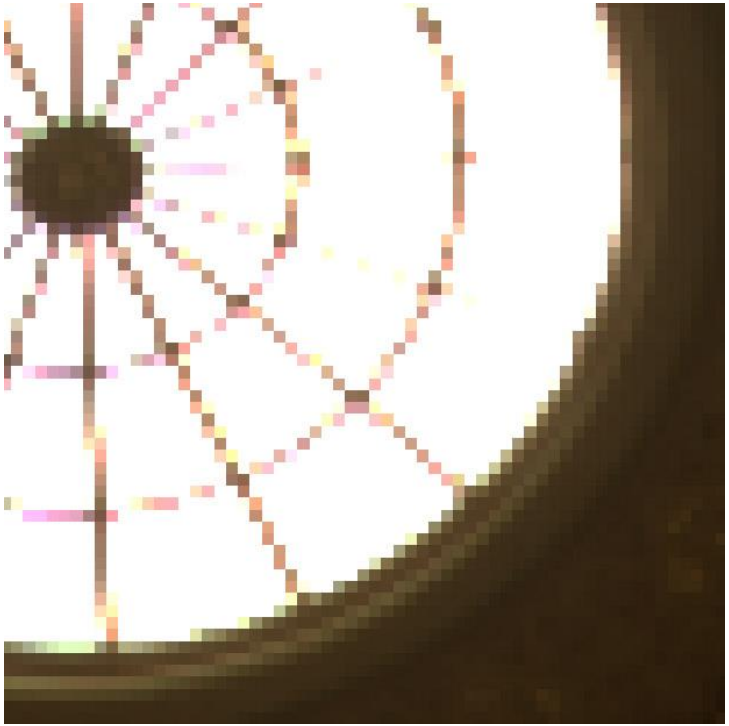
Compressed

RGB BBox Refine

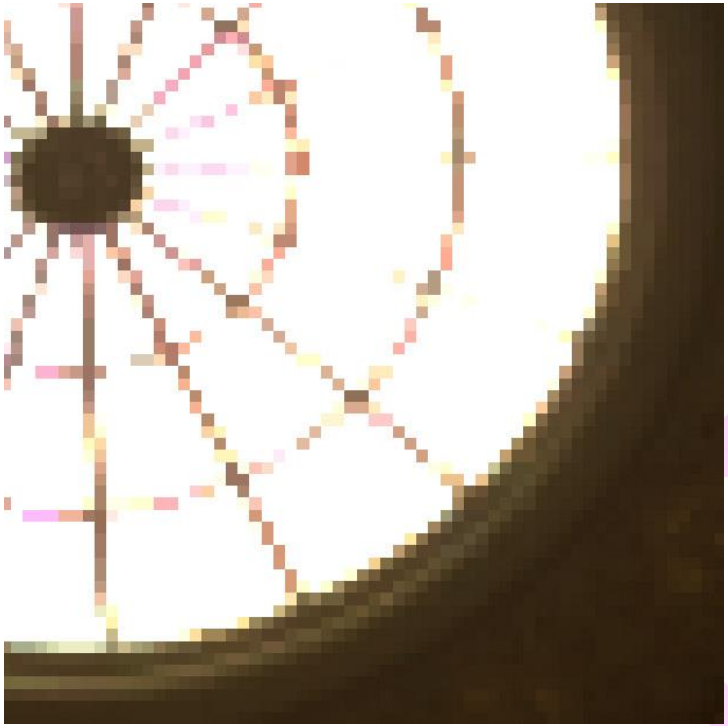
- Rebuild bbox using the second smallest and second largest R, G and B



RGB BBox Refine Results



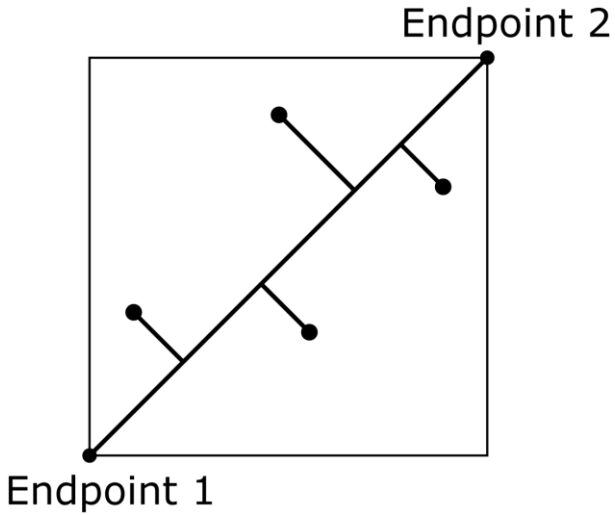
Reference



Compressed

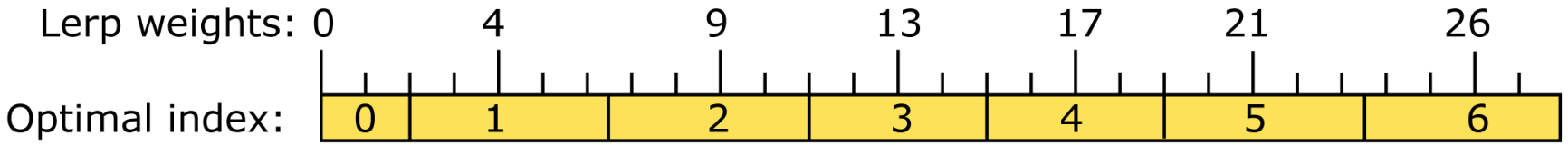
Indices

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



Indices

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

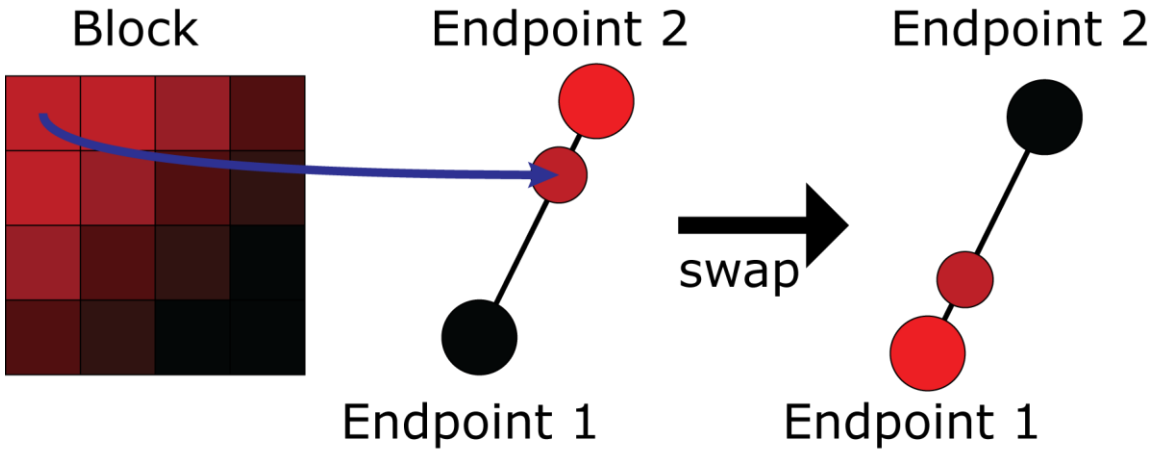


- Simple approximation is to fit equation for smallest error:

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

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



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]

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

"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

“Fast” Preset Results

Reference

Compressed

“Fast” Preset Results

Reference

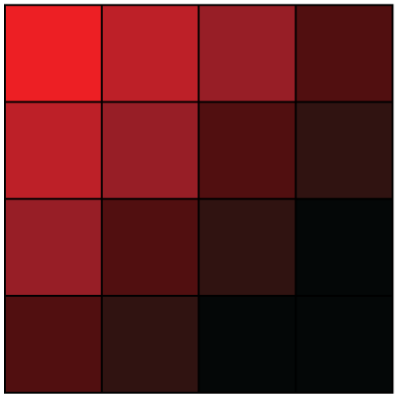
Compressed

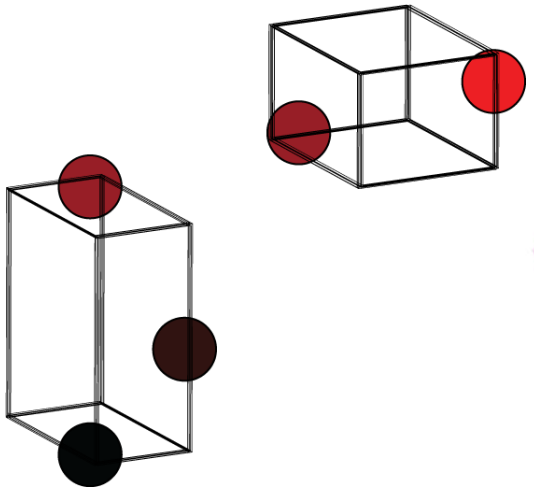
“Quality” Preset

- 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

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





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

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

“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

“Quality” Preset Results

Reference

Compressed

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

Shader Optimization

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

Embedding palletized sprites in shader code

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

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


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