Lecture 6:

Texturing Part II: Texture Compression

Visual Computing Systems
CMU 15-869, Fall 2013

Review: mechanisms to reduce aliasing in the graphics pipeline

- **When sampling visibility?**
  - **Supersampling: sample signal densely (multiple times per pixel)**

- **When sampling shading? (appearance)**
  - **Prefiltering: remove high frequencies from texture signal prior to shading**

Review: operations in a texture fetch

For each texture fetch in a shader program:

1. Compute du/dx, du/dy, dv/dx, dv/dy differentials from quad fragment
2. Compute mip-map level: d (for tri-linear filtering)
3. Convert normalized texture coordinate uv to texel coordinates: tu,tv
4. Compute required texels **
5. If texture data in filter footprint (eight texels for trilinear filtering) is not in cache
   - Load texel data from DRAM
   - Decompress texel data
6. Perform tri-linear interpolation according to (tu,tv,d) to get filtered texture sample

A texture fetch involves both data access and also significant amounts of computation: all modern GPUs have dedicated fixed-function hardware support for texture sampling and texture decompression.

** May involve wrap, clamp, etc. of texel coordinates according to sampling mode configuration

What type of data reuse does a texture cache capture?

- Spatial locality, not temporal locality
  - The same texels are required to filter texture fetches from adjacent fragments (due to overlap of filter support regions)
  - Little-to-no temporal locality within a fragment shader (little reason for a shader to access the same part of the texture map twice)

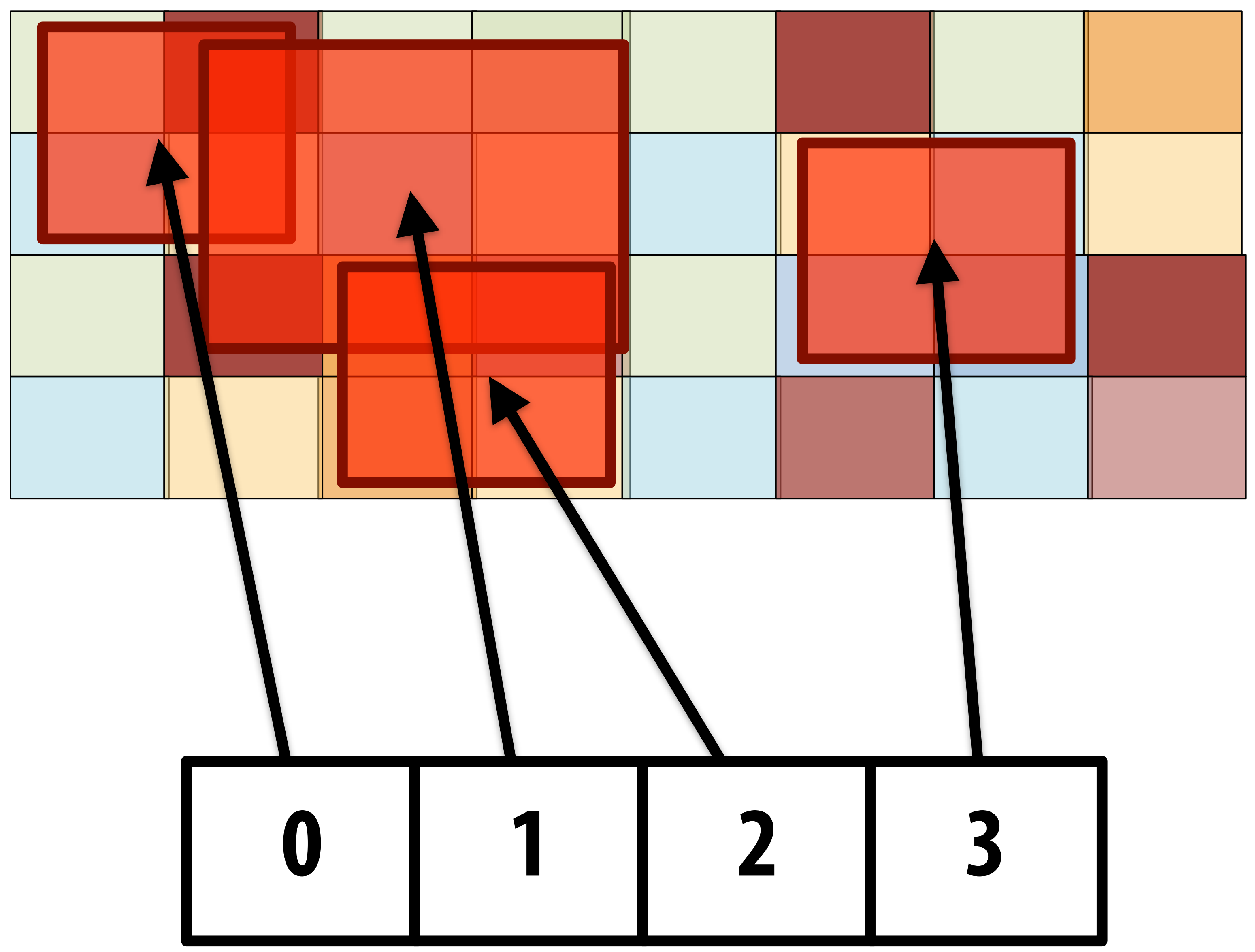


Figure shows filter support regions from texture fetches from four adjacent fragments

Texture data access characteristics

- **Key metric: unique texel to fragment ratio**
  - Number of unique texels accessed when rendering a scene divided by number of fragments processed [see Igeny reading for stats: often less than < 1]
  - What is the worst case ratio? (assuming trilinear filtering)
  - How can incorrect computation of texture miplevel (d) affect this?
- **In practice, caching behavior is good, but not CPU workload good**
  - [Montrym & Moreton 95] design for 90% hits
  - Why? (only so much spatial locality)
- **Implications**
  - GPU must have solution for memory access latency
  - GPU must provide high memory bandwidth for texture data access
  - GPU must reduce its bandwidth requirements using texture compression

Texture compression

Texture compression

- Goal: reduce bandwidth requirements of texture access
- Texture is read-only data
  - Compression can be performed offline, can take significantly longer than decompression
  - Lossy compression schemes are permissible
- Design requirements
  - Support random texel access (constant time access to any texel)
  - High performance decompression
  - Simple algorithms (low-cost hardware implementation)
  - High compression ratio
  - High visual quality

Simple scheme: color palette (indexed color)

- Lossless (if images contains small number of unique colors)

Color palette (eight colors)

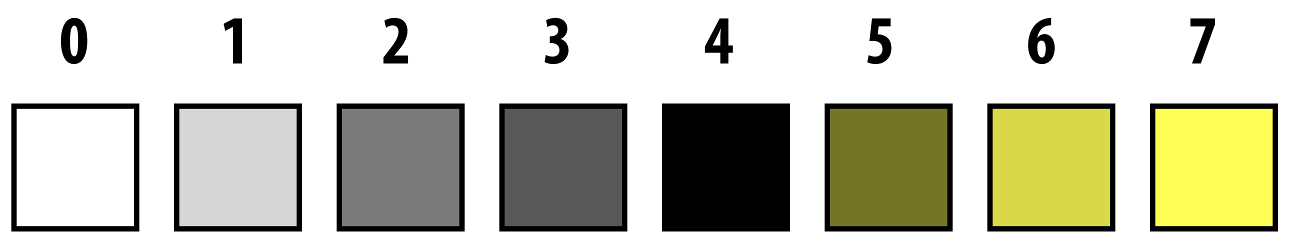
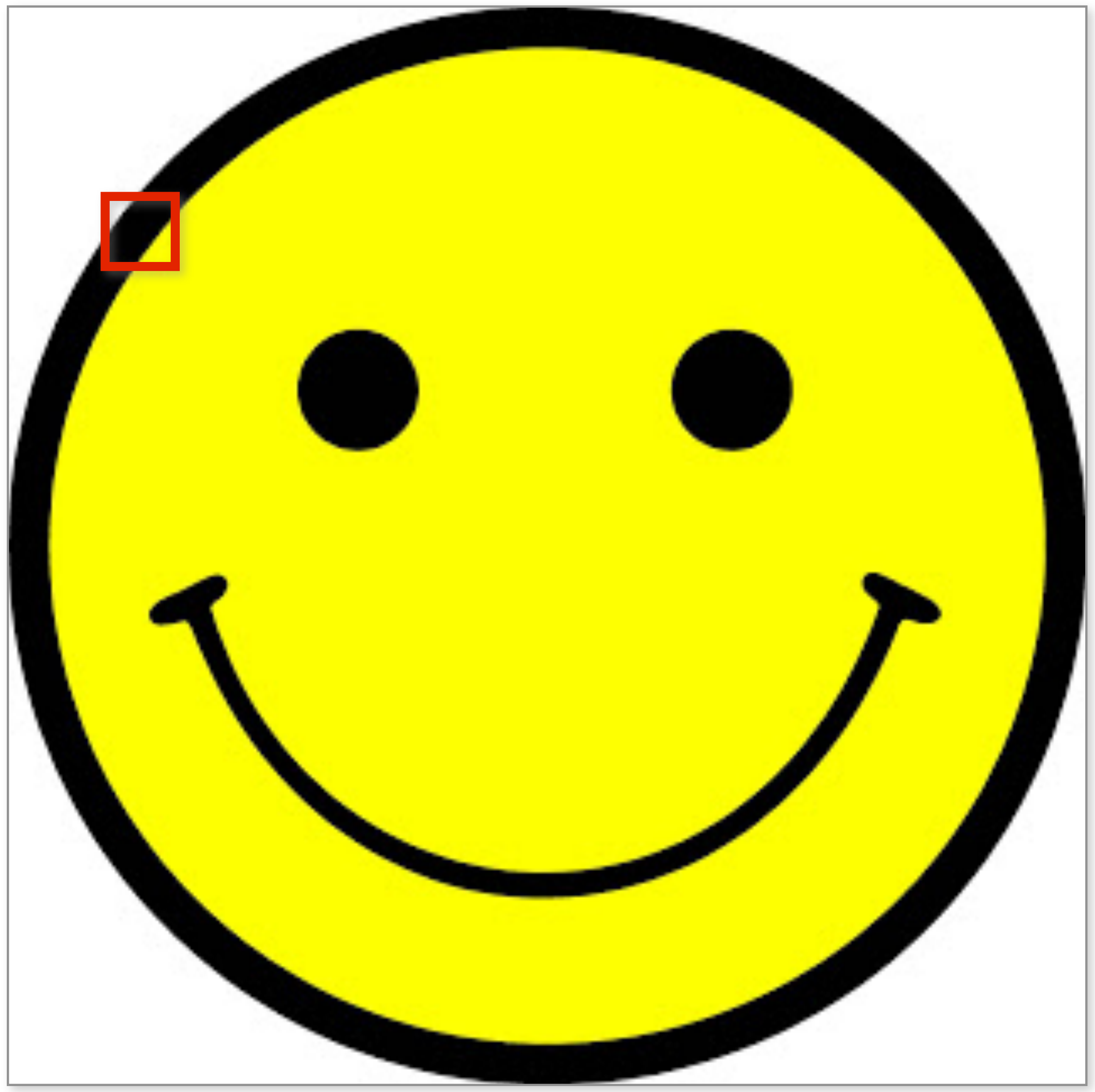
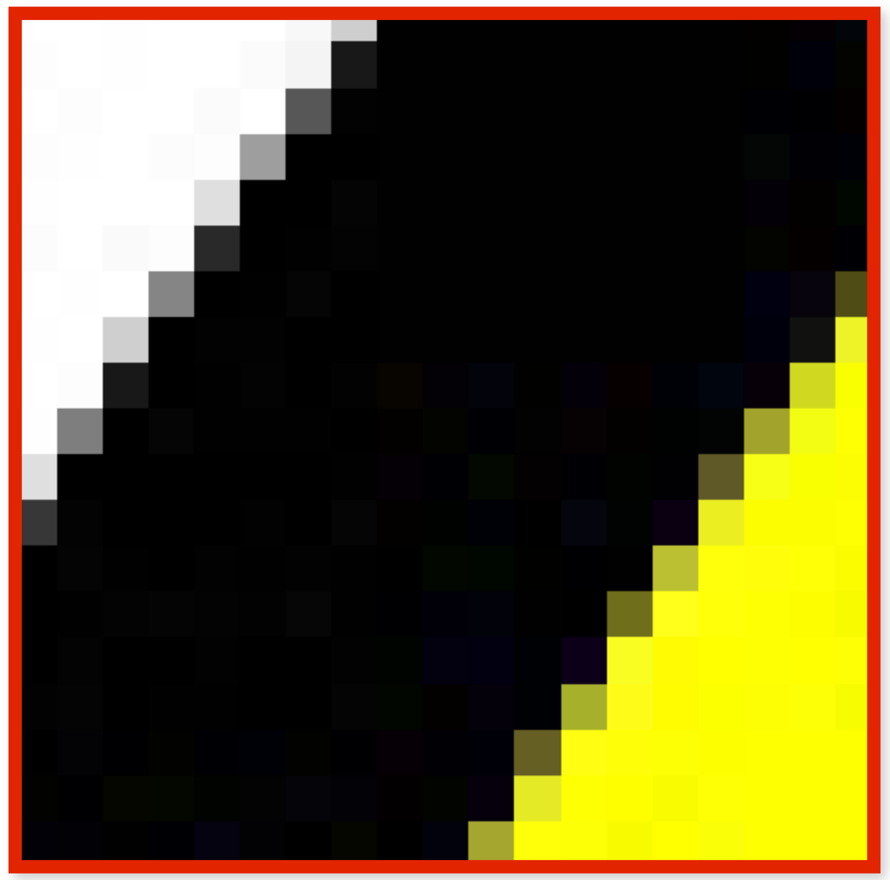


Image encoding in this example:

3 bits per pixel + eight RGB values in palette (8x24 bits)

0	1	3	6
0	2	6	7
1	4	6	7
4	5	6	7





Per-block palette

- **Block-based compression scheme on 4x4 pixel blocks**
  - Idea: there might be many colors across entire image, but can approximate all values in any 4x4 pixel region using only a few unique colors

- **Per-block palette (e.g., four colors in palette)**
  - 12 bytes for palette (assume 24 bits per RGB color: 8-8-8)
  - 2 bits per pixel (4 bytes for per-pixel indices)
  - 16 bytes (3x compression on original data: 16x3=48 bytes)

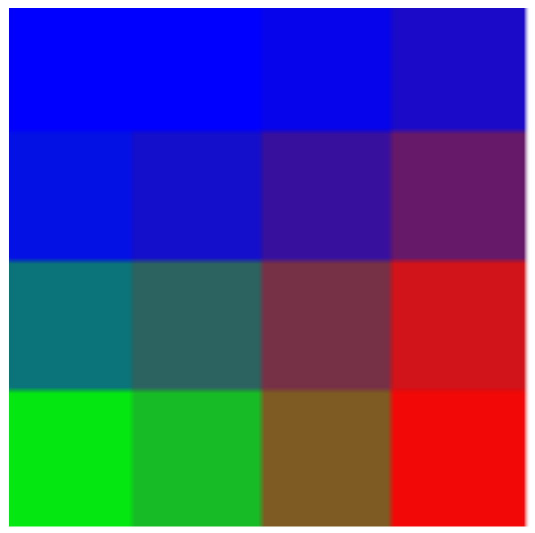
- **Can we do better?**

S3TC

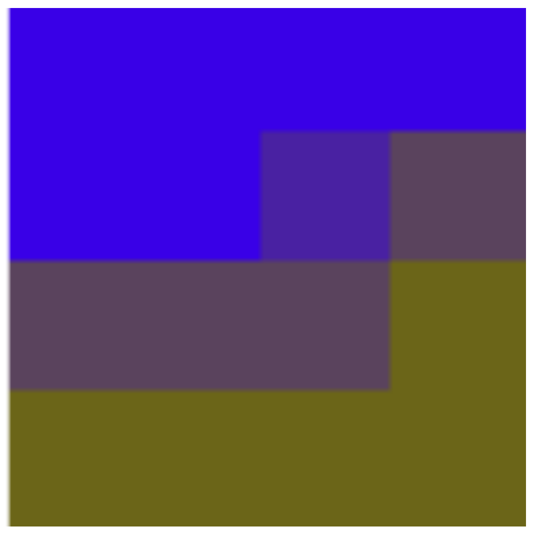
(Called BC1 or DXTC by Direct3D)

- **Palette of four colors encoded in two bytes:**
  - Two low-precision base colors: C_0 and C_1 (2 bytes each: RGB 5-6-5 format)
  - Other two colors computed from base values
    - $\frac{1}{3}C_0 + \frac{2}{3}C_1$
    - $\frac{2}{3}C_0 + \frac{1}{3}C_1$
- **Total footprint of 4x4 pixel block: 8 bytes**
  - 4 bpp, 6:1 compression ratio (fixed ratio: independent of data values)
- **S3TC assumption:**
  - All pixels in a 4x4 block lie on a line in RGB color space
- **Additional mode:**
  - If C0 < C1, then third color is $\frac{1}{2}C_0 + \frac{1}{2}C_1$ and fourth color is transparent black

S3TC artifacts



Original data

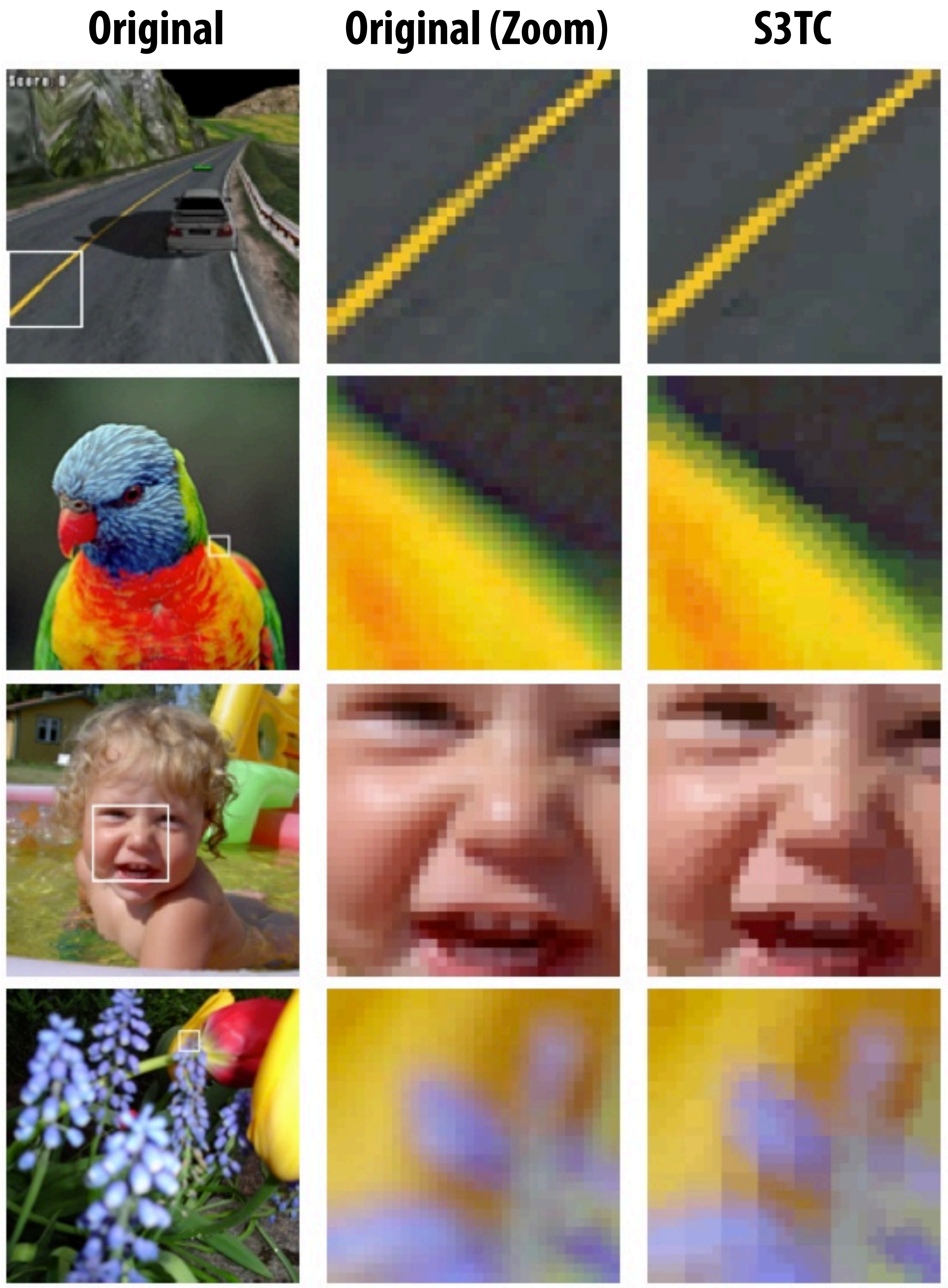


Compressed result

Cannot interpolate red and blue to get green (here compressor chose blue and yellow as base colors to minimize overall error)

But scheme can work well in practice on natural images. (see images at right)

Image credit:
http://renderingpipeline.com/2012/07/texture-compression/



[Strom et al. 2007]

Y’CbCr color space

Y’ = perceived luminance

Cb = blue-yellow deviation from gray

Cr = red-cyan deviation from gray

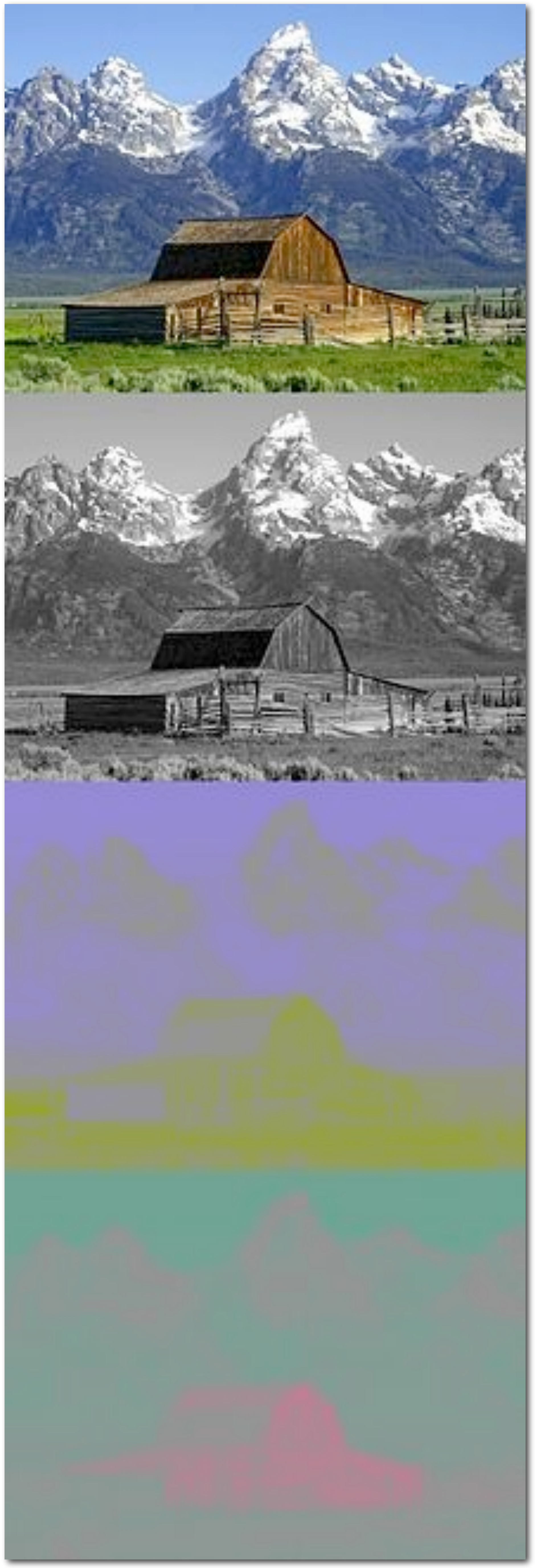


Image credit: Wikipedia

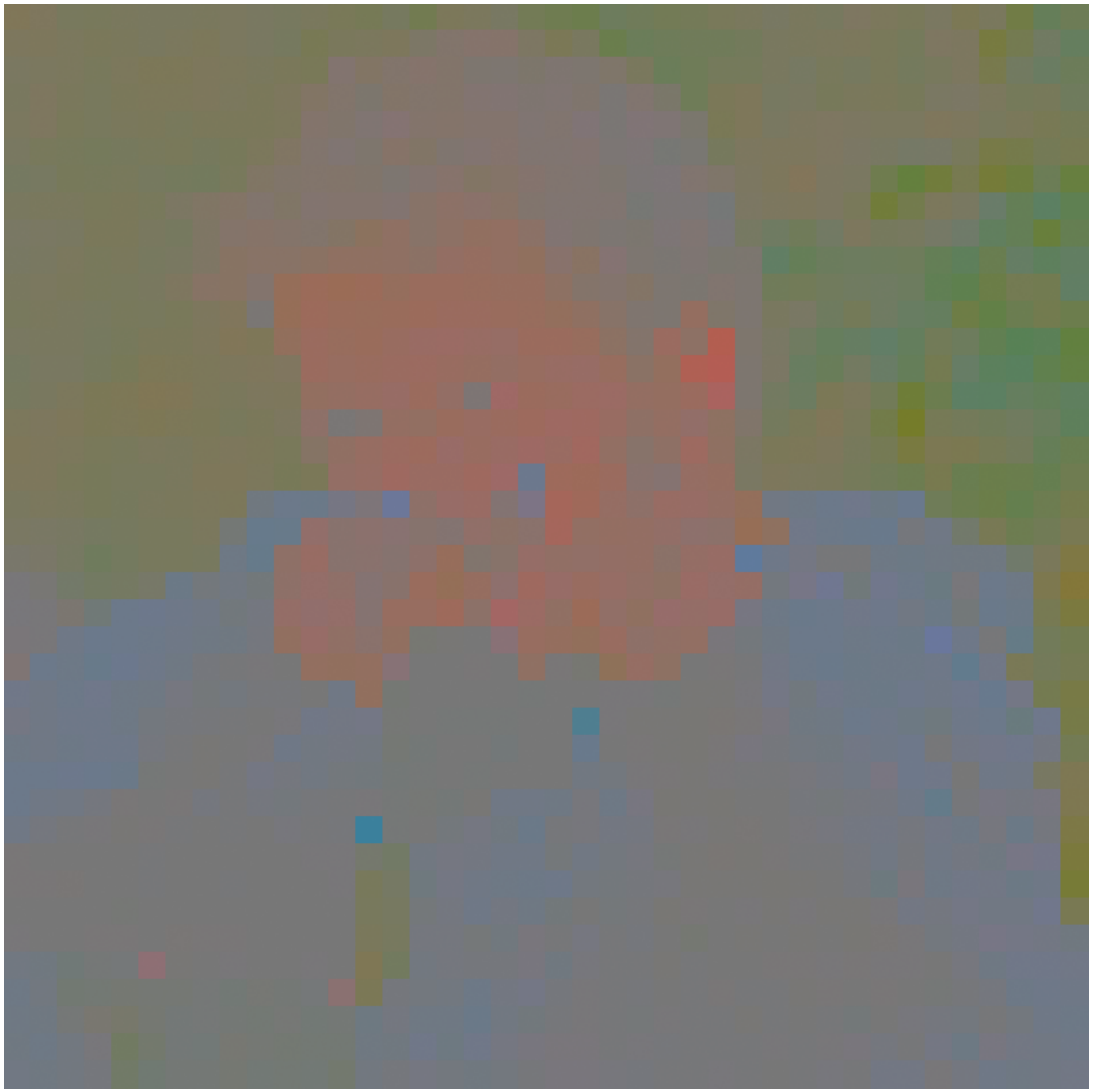
Conversion from RGB to Y’CbCr:

$$\begin{aligned}
Y' &= 16 + \frac{65.738 \cdot R'_D}{256} + \frac{129.057 \cdot G'_D}{256} + \frac{25.064 \cdot B'_D}{256} \\
C_B &= 128 + \frac{-37.945 \cdot R'_D}{256} - \frac{74.494 \cdot G'_D}{256} + \frac{112.439 \cdot B'_D}{256} \\
C_R &= 128 + \frac{112.439 \cdot R'_D}{256} - \frac{94.154 \cdot G'_D}{256} - \frac{18.285 \cdot B'_D}{256}
\end{aligned}$$

Demo

Original picture of Kayvon

Demo

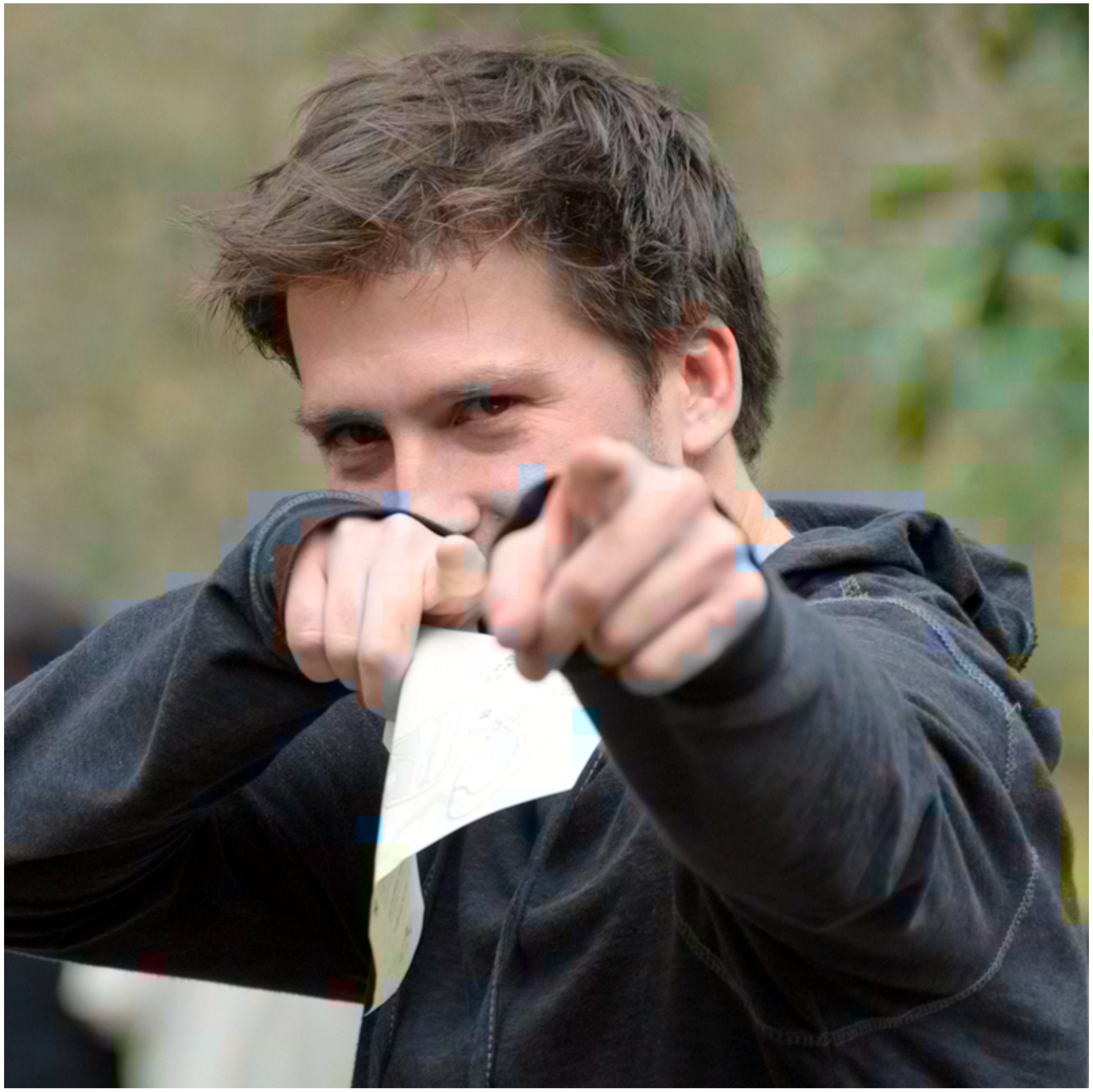


Color channels downsampled by a factor of 20 in each dimension

Demo

Full resolution luminance

Demo


Reconstructed result

Chroma subsampling

Y'CbCr is an efficient storage (and transmission) representation because Y' can be stored at higher resolution than CbCr without much loss in perceived visual quality

4:2:2 representation:

Store Y' at full resolution

Store Cb, Cr at full vertical resolution, but only half horizontal resolution

Y'_{00} Cb_{00} Cr_{00}	Y'_{10}	Y'_{20} Cb_{20} Cr_{20}	Y'_{30}
Y'_{01} Cb_{01} Cr_{01}	Y'_{11}	Y'_{21} Cb_{21} Cr_{21}	Y'_{31}

PACKMAN

[Strom et al. 2004]

- **Block-based compression on 2x4 pixel blocks**
  - Idea: vary luminance per pixel, but specify color per block
- **Each block encoded as:**
  - A single base color per block (12 bits: RGB 4-4-4)
  - 4-bit index identifying one of 16 predefined luminance modulation tables
  - Per-pixel 2-bit index into luminance modulation table (8x2=16 bits)
  - Total block size = 12 + 4 + 16 = 32 bits (6:1 compression ratio)
- **Decompression:**
  - `texel[i] =  base_color + table[table_id][table_index[i]];`

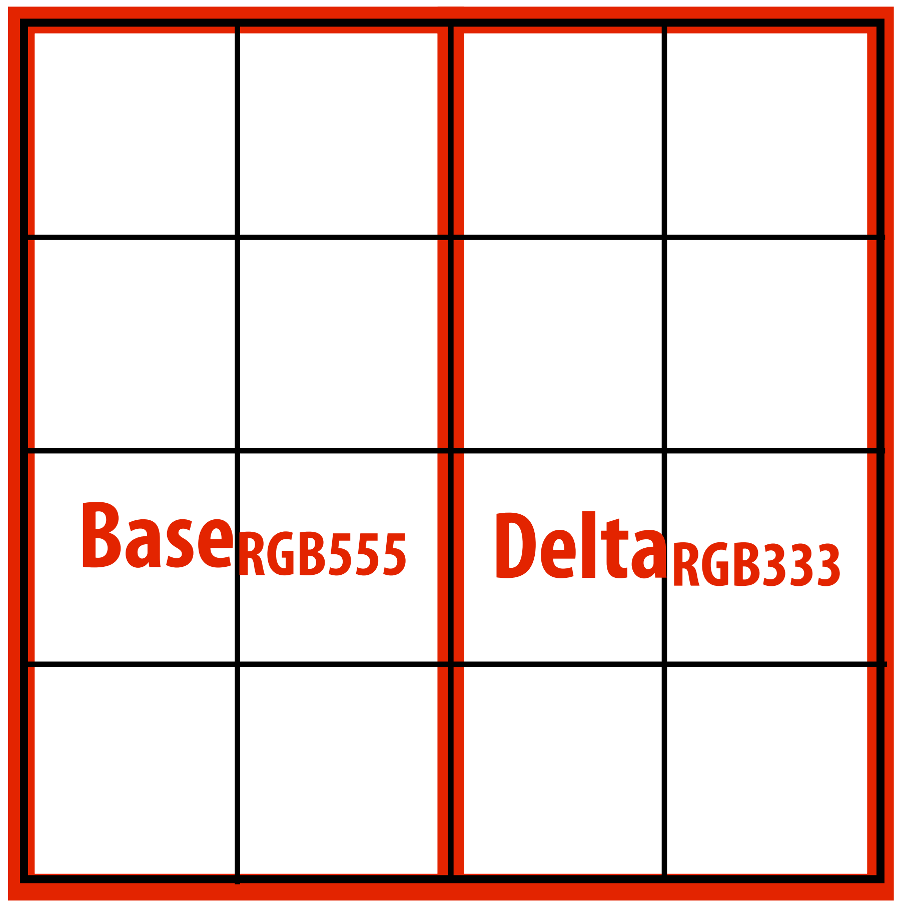
table codeword	0	1	2	3	4	5	6	7
	-8	-12	-31	-34	-50	-47	-80	-127
	-2	-4	-6	-12	-8	-19	-28	-42
	2	4	6	12	8	19	28	42
	8	12	31	34	50	47	80	127

Example codebook for modulation tables (8 of 16 tables shown)

iPackman (ETC)

[Strom et al. 2005]

- Improves on problems of heavily quantized and sparsely represented chrominance in PACKMAN
  - Higher resolution base chrominance + differential represents color more accurately
- Operates on 4x4 pixel blocks
  - Optionally represent 4x4 block as two eight-pixel subblocks with differentials (else use PACKMAN for two subblocks)
    - 1 bit designates whether differential scheme is in use
  - Base color for first block (RGB 5-5-5: 15 bits)
  - Color differential for second block (RGB 3-3-3: 9 bits)
  - 1 bit designating if subblocks are 4x2 or 2x4
  - 3-bit index identifying modulation table per subblock (2x3 bits)
  - Per-pixel table offsets (2x16 bits)
  - Total compressed block size: 1 + 15 + 9 + 1 + 6 + 32 = 64 bits (6:1 ratio)



PACKMAN vs. iPACKMAN quality comparison

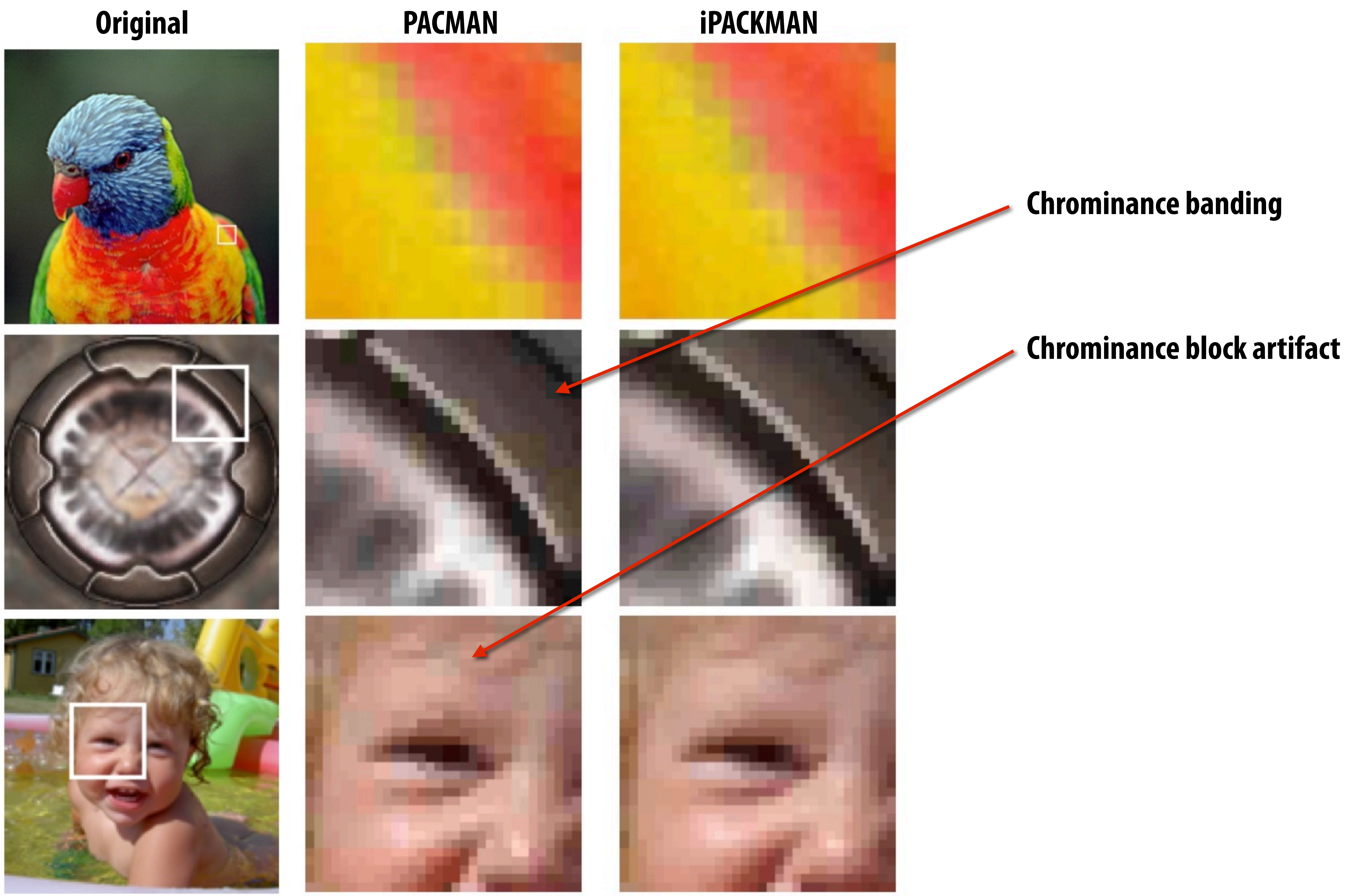
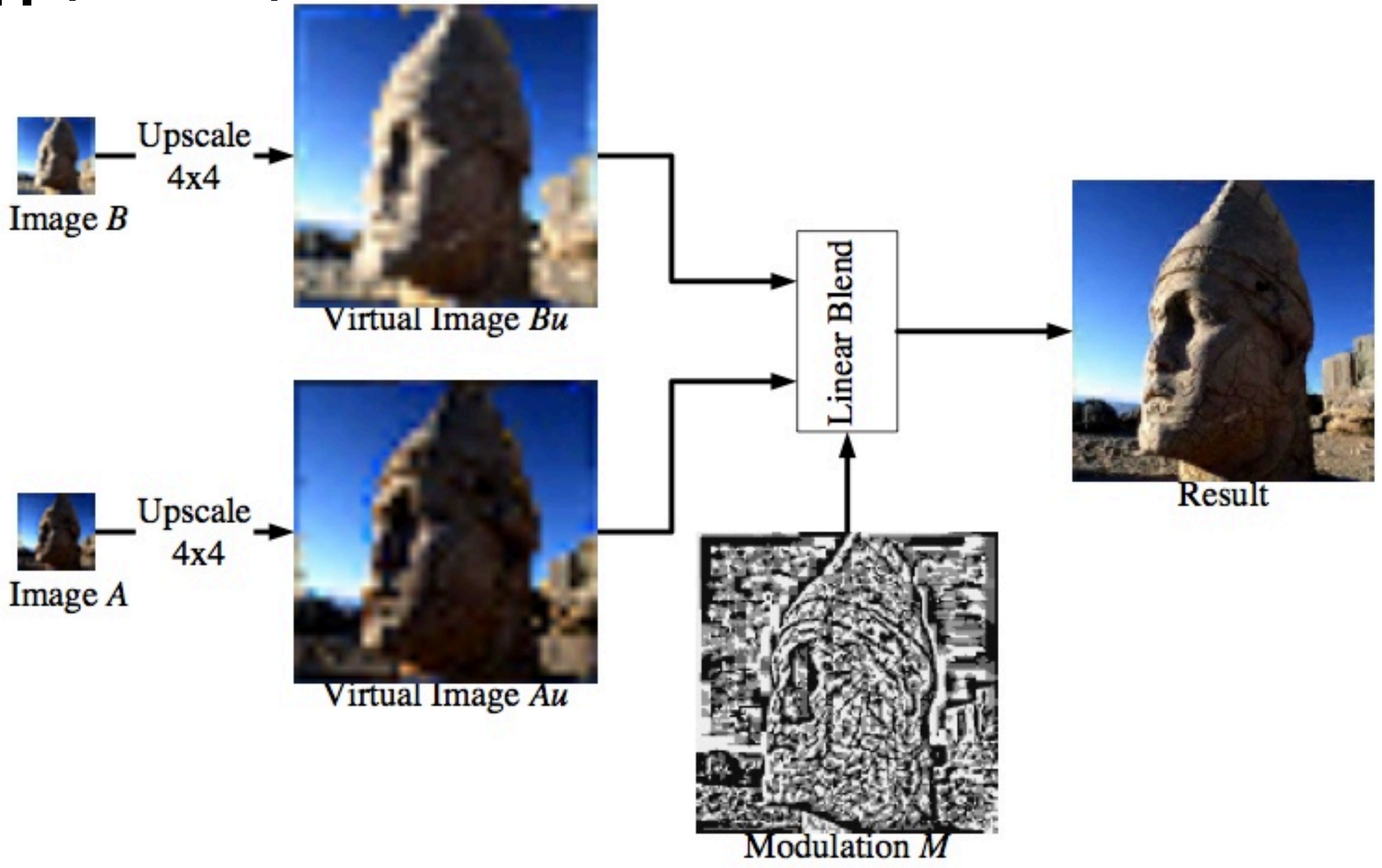


Image credit: Strom et al. 2005

PVRTC

[Fenney et al. 2003]

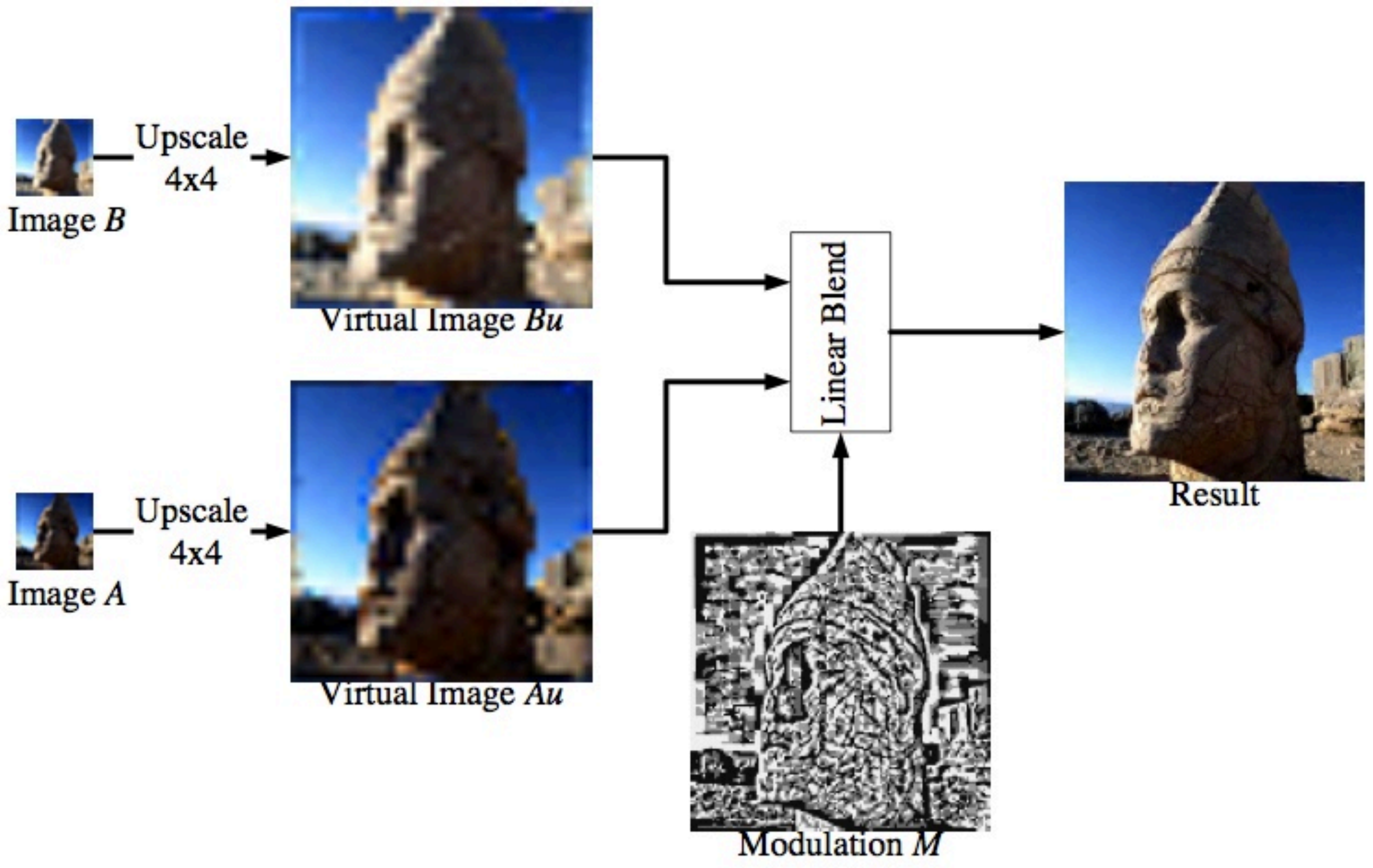
- Not a block based format (used in Imagination PowerVR GPUs)
  - Store low-frequency base images A and B
    - Base images downsampled by factor of 4 in each dimension ($^1/_{16}$ fewer texels)
    - Store base image pixels in RGB 5:5:5 format (+ 1 bit alpha)
  - Store 2-bit modulation factor per pixel
  - Total footprint: 4 bpp (6:1 ratio)



PVRTC

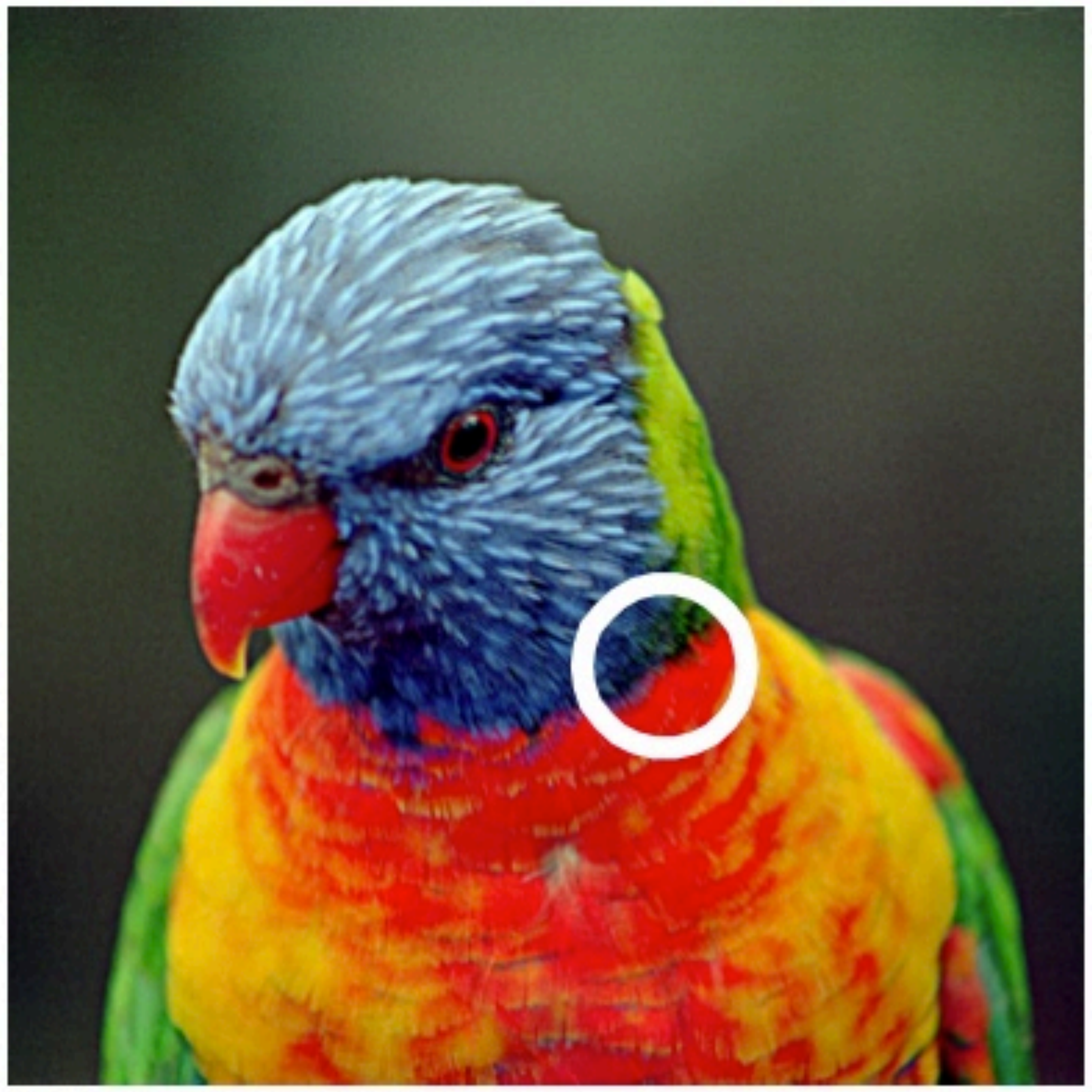
[Fenney et al. 2003]

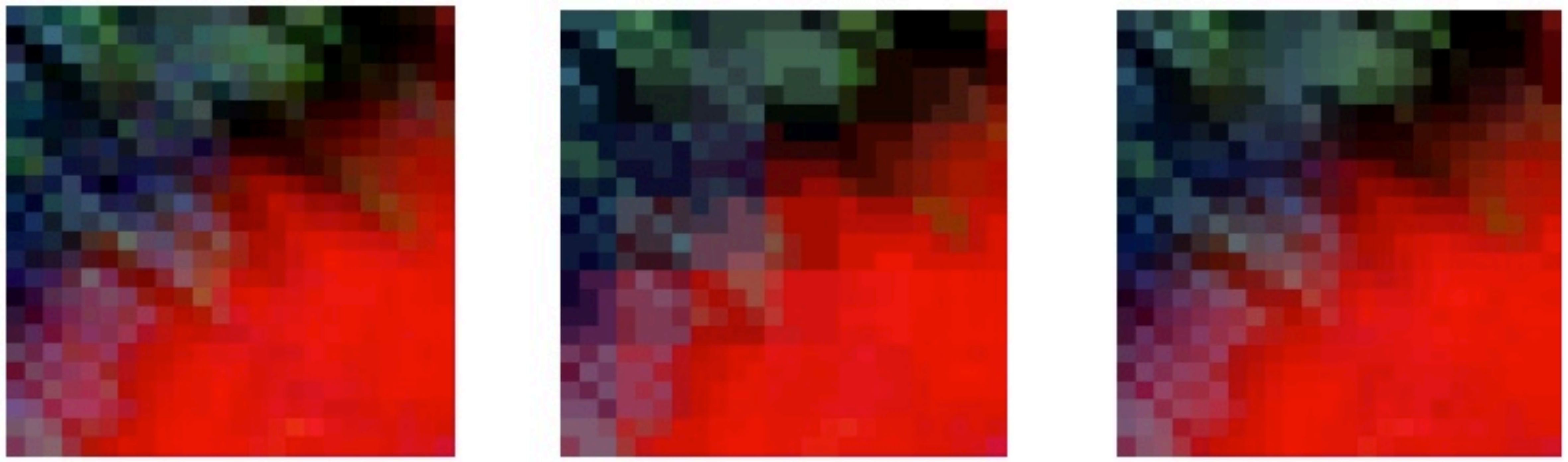
- **Decompression:**
  - Bilerp A and B (upsample) to get value at desired texel
  - Interpolate upsampled values according to modulation factor



PVRTC avoids blocking artifacts

Recall: bilinear upsampling of low-resolution base images





PVRTC

Image credit: Fenney et al. 2003

Summary: texture compression

- **Many schemes target 6:1 fixed compression ratio (4 bpp)**
  - Predictable performance
  - 8 bytes per block desirable for memory transfers
- **Employ lossy compression**
  - Exploit characteristics of the human visual system to minimize <u>perceived</u> error
- **Block-based vs. not-block based**
  - Block-based: S3TC/DXTC/BC1, iPACKMAN/ETC/ETC2, ASTC (not discussed today)
  - Not-block-based: PVRTC
- **Only decompression discussed today:**
  - Compression can be done off-line (except for textures produced at runtime by rendering to buffers)

Texture Access Latency

Texture access is a long-latency operation

For each texture fetch in a shader program:

1. Compute du/dx, du/dy, dv/dx, dv/dy differentials from quad fragment
2. Compute mip-map level: d (for tri-linear filtering)
3. Convert normalized texture coordinate uv to texel coordinates: tu,tv
4. Compute required texels
5. If texture data in filter footprint is not in cache (recall: GPUs miss cache often)
   - Load texel data from DRAM
   - Decompress texel data
6. Perform tri-linear interpolation according to (tu,tv,d) to get filtered texture sample

Latency involves math for texel determination, decompression, and filtering (not just latency of fetching data from memory)

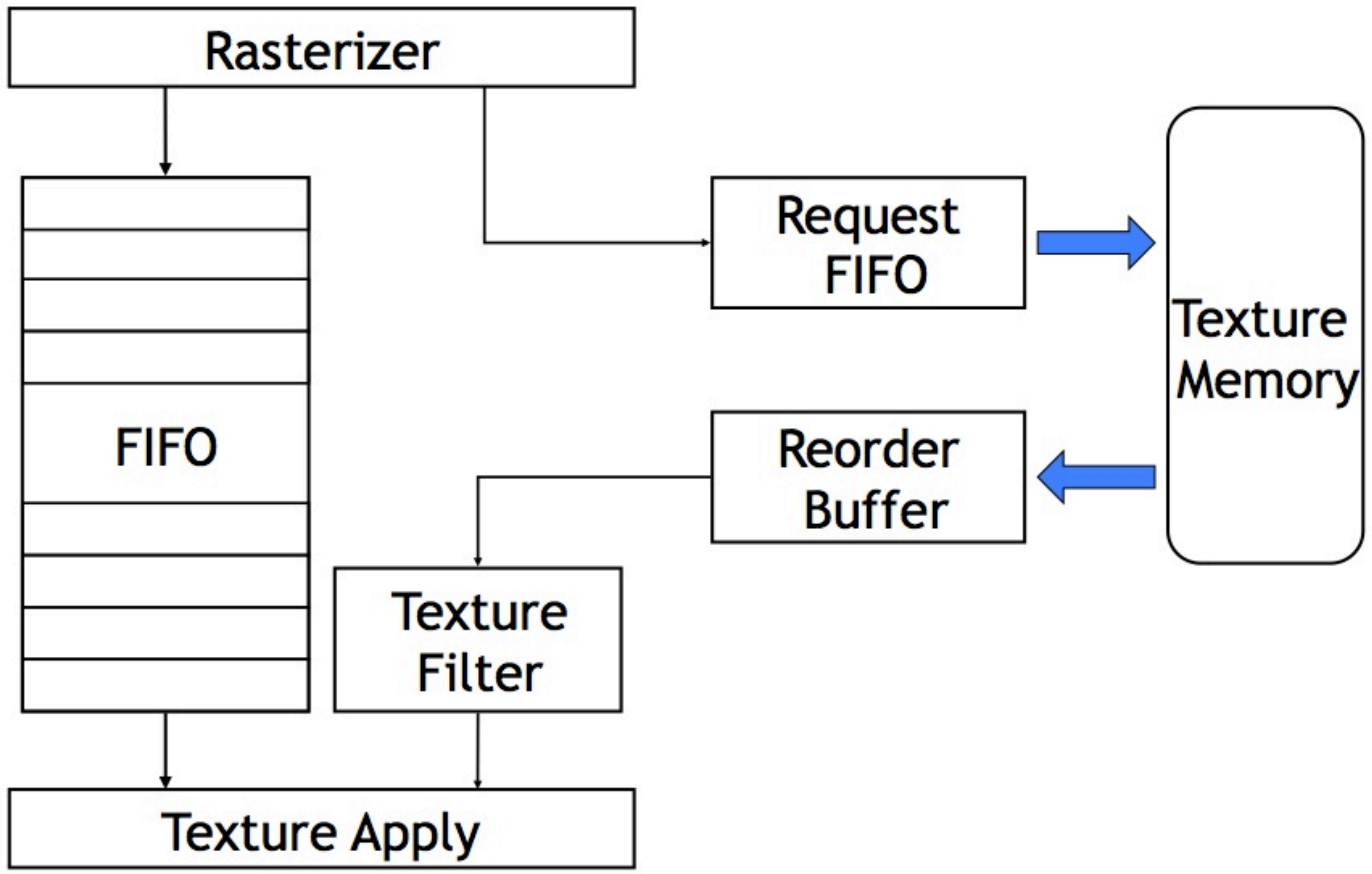
Addressing texture access latency challenge

- Processor requests filtered texture data → processor waits hundreds of cycles (significant loss of performance)

- Solution prior to programmable cores on GPUs: texture prefetching
  - Today's reading: Igehy et al. *Prefetching in a Texture Cache Architecture*

- Solution in modern GPUs: heavily multi-threaded processor cores
  - Will omit today, but will discuss in detail in a future lecture

Prefetching example: large fragment FIFOs



Note: This diagram does not contain a texture cache. See reading for implementation of prefetching with caching.

Slide credit: Akeley and Hanrahan

Texture system summary

- **A texture lookup is a lot more than a 2D array access**
  - Significant computational and bandwidth expense, implemented in specialized fixed-function hardware

- **Bandwidth reduction mechanism: GPU texture caches**
  - Primarily serve to amplify limited DRAM bandwidth
  - Not to reduce latency to off-chip memory
  - Small in size, multi-ported (e.g., need to access eight texels simultaneously)

  - Tiled rasterization order + tiled texture layout serve to increase cache hits

- **Bandwidth reduction mechanism: texture compression**
  - Lossy compression schemes
  - Fixed-compression ratio encodings (e.g, 6:1 ratio, 4 bpp is common for RGB data)
  - Allow random access into compresses representation

- **Latency avoidance/hiding mechanisms:**
  - Prefetching (in the old days)
  - Multi-threading (in modern GPUs)