Image Processing Tricks in OpenGL

Simon Green

NVIDIA Corporation

Overview

- Image Processing in Games
- Histograms
- Recursive filters
- JPEG Discrete Cosine Transform

Image Processing in Games

- Image processing is increasingly important in video games
- Games are becoming more like movies
  - a large part of the final look is determined in “post”
  - color correction, blurs, depth of field, motion blur
- Important for accelerating offline tools too
  - pre-processing (lightmaps)
  - texture compression

Image Histograms

- Image histograms give frequency of occurrence of each intensity level in image
  - useful for image analysis, HDR tone mapping algorithms
- OpenGL imaging subset has histogram functions
  - but this is not widely supported
- Solution - calculate histograms using multiple passes and occlusion query

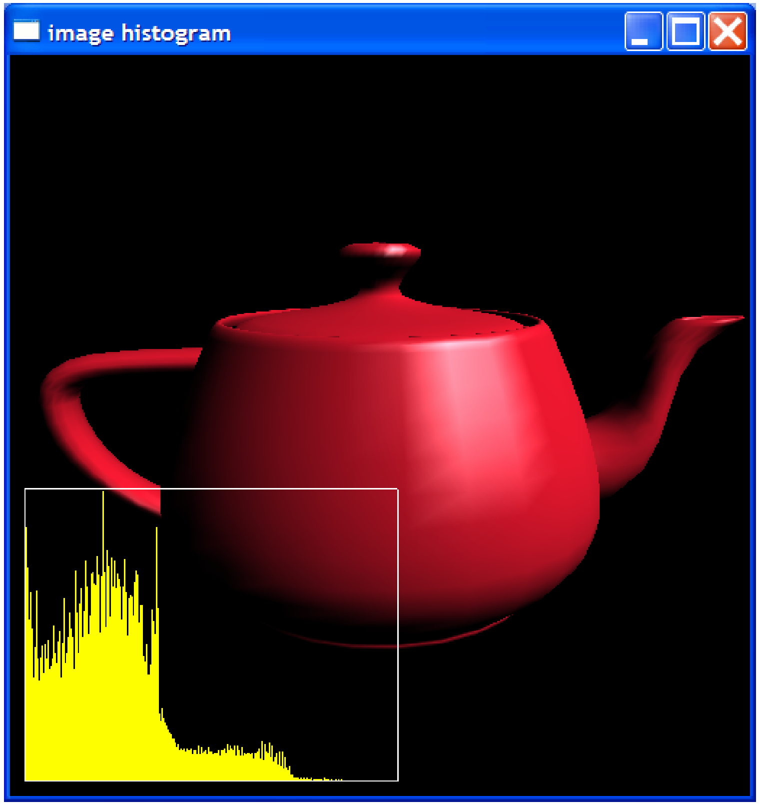
Histograms using Occlusion Query

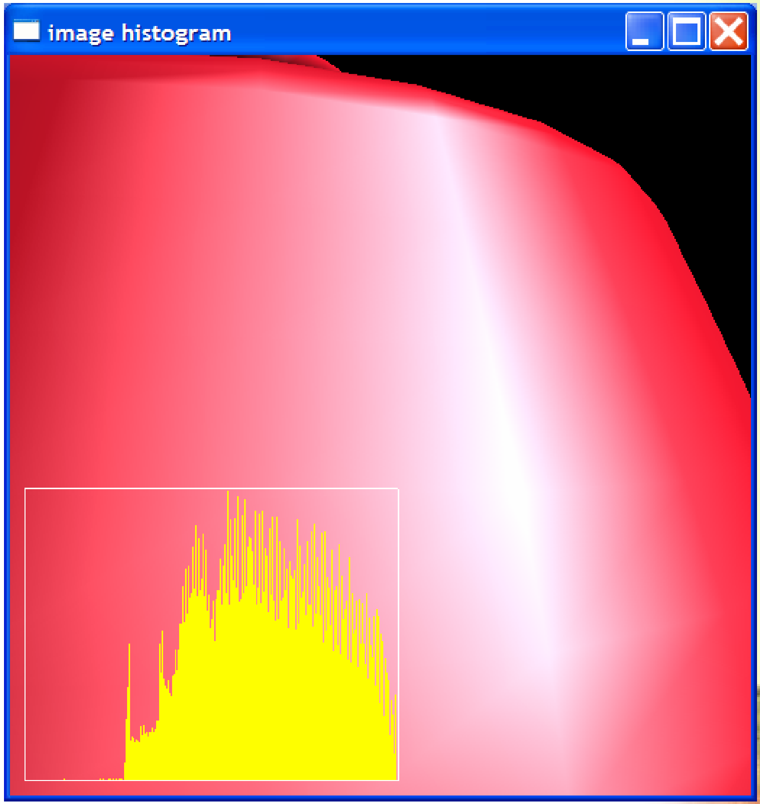
- Render scene to texture
- For each bucket in histogram
  - Begin occlusion query
  - Draw quad with scene texture
    - Use fragment program that discards fragments outside appropriate luminance range
  - End occlusion query
  - Get number of fragments that passed, store in histogram array
- Process histogram
- Requires n passes for n buckets

Histogram Fragment Program

```
float4 main(in float4 wpos : WPOS,
            uniform samplerRECT tex,
            uniform float min,
            uniform float max,
            uniform float3 channels
            ) : COLOR
{
    // fetch color from texture
    float4 c = texRECT(tex, wpos.xy);

    // calculate luminance or select channel
    float lum = dot(channels, c.rgb);

    // discard pixel if not inside range
    if (lum < min || lum >= max)
        discard;

    return c;
}
```

Histogram Demo





Performance

- Depends on image size, number of passes
- 40fps for 32 bucket histogram on 512 x 512 image, GeForce 5900
- For large histograms, may be faster to readback and compute on CPU

Recursive (IIR) Image Filters

- Most existing blur implementations use standard convolution – filter output is only function of surrounding pixels
- If we scan through the image, can we make use of the previous filter outputs?
- Output of a recursive filter is function of previous inputs *and* previous outputs
  - feedback!
- Simple recursive filter

$$y[n] = a*y[n-1] + (1-a)*x[n]$$

Recursive Image Filters

- Require fewer samples for given frequency response
- Can produce arbitrarily wide blurs for constant cost
  - this is why Gaussian blurs in Photoshop take same amount of time regardless of width
- But difficult to analyze and control
  - like a control system, trying to follow its input
  - mathematics is very complicated!

FIR vs. IIR

- Impulse response of filter is how it responds to unit impulse (discrete delta function):
  - also known as point spread function
- Finite Impulse Response (FIR)
  - response to impulse stops outside filter footprint
  - stable
- Infinite Impulse Response (IIR)
  - response to impulse can go on forever
  - can be unstable
  - widely used in digital signal processing

Review: Building Summed Area Tables using Graphics Hardware

- Presented at GDC 2003
- Each texel in SAT is the sum of all texels below and to the left of it
- Implemented by rendering lines using render-to-texture
  - Sum columns first, and then rows
  - Each row or column is rendered as a line primitive
  - Fragment program adds value of current texel with texel to the left or below

Building Summed Area Table

1	1	1	1
1	1	1	1
1	1	1	1
1	1	1	1

Original image

1	2	3	4
1	2	3	4
1	2	3	4
1	2	3	4

Sum columns

4	8	12	16
3	6	9	12
2	4	6	8
1	2	3	4

Sum rows

- For n x m image, requires rendering 2 x n x m pixels, each of which performs two texture lookups

Problems With This Technique

- Texturing from same buffer you are rendering to can produce undefined results
  - e.g. Texture cache changed from NV3x to NV4x – broke SAT demo
  - Don’t rely on undefined behaviour!
- Line primitives do not make very efficient use of rasterizer or shader hardware
  - Most modern graphics hardware processes groups of pixels in parallel

Solutions

- **Use two buffers, ping-pong between them**
  - **Copy changes back from destination buffer to source each pass**
  - **Buffer switching is fast with framebuffer object extension**
- **Can also unroll loop so that we render 2 x n quads instead of lines**
  - **Unroll fragment program so that it does computations for two fragments**
  - **Use per-vertex color to determine if we're rendering odd or even row/column**

Implementing IIR Image Filters

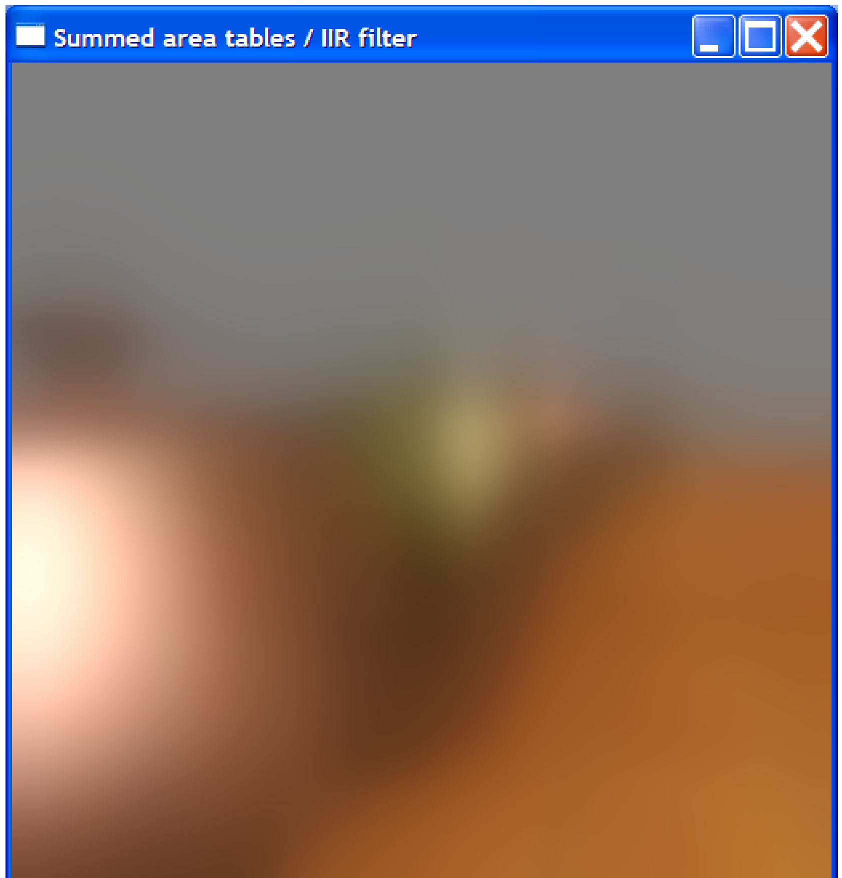
- Can implement recursive (IIR) image filters using same technique as summed area table
- Scan through image, rendering line or quad primitives
- Fragment program reads from previous output buffer and previous input buffer, writes to third buffer
- Process rows, then columns

Simple IIR Filter

```
float4 main(vf30 In,
            uniform samplerRECT y,    // out
            uniform samplerRECT x,    // in
            uniform float4 delta,
            uniform float4 a,         // filter coefficients
            ) : COLOR
{
    float2 n = In.WPOS.xy);     // current
    float2 nm1 = n + delta.xy;  // previous

    return lerp(texRECT(y, nm1), texRECT(x, n), a[0]);
}
```

Simple IIR Filter (Before)

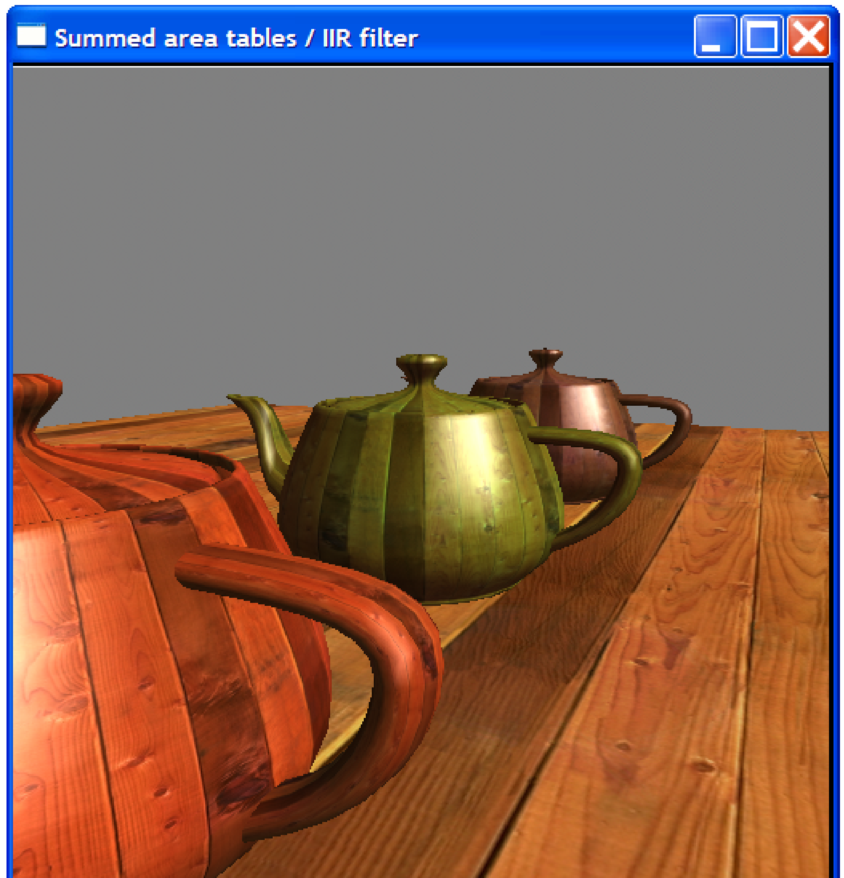
Simple IIR Filter (After)



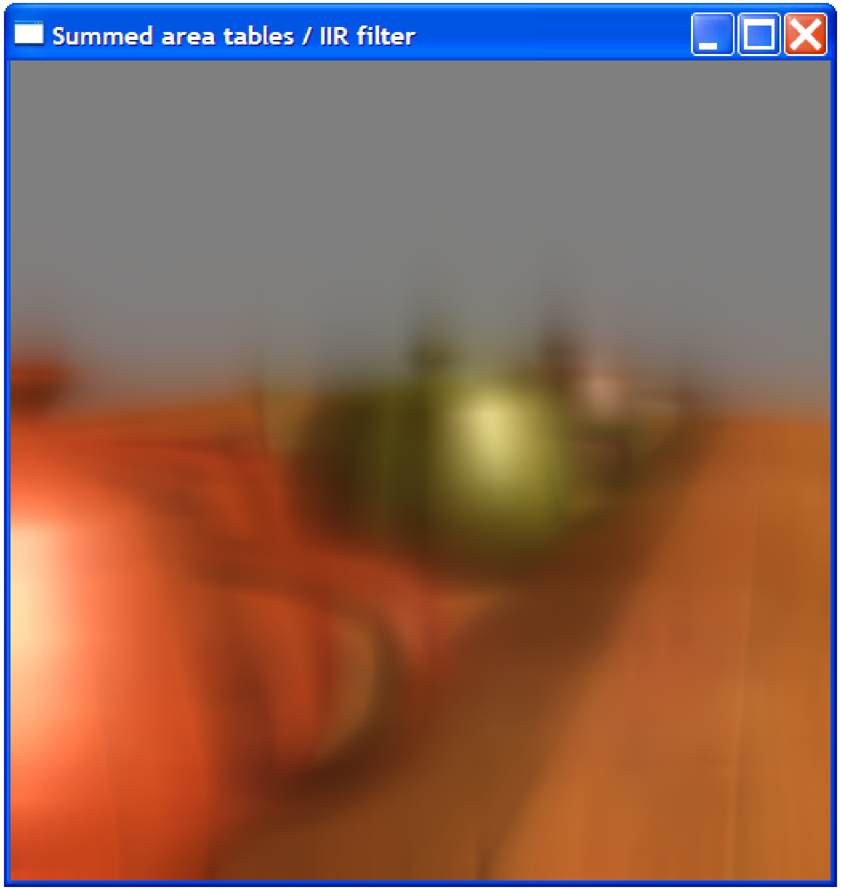
Symmetric Recursive Filtering

- Recursive filters are directional
- Causes phase shift of data
- Not a problem for time series (e.g. audio), but very obvious with images
- Can combine multiple recursive filters to construct zero-phase shift filter
- Run filter in positive direction (left to right) first, and then in negative direction (right to left)
  - Phase shifts cancel out

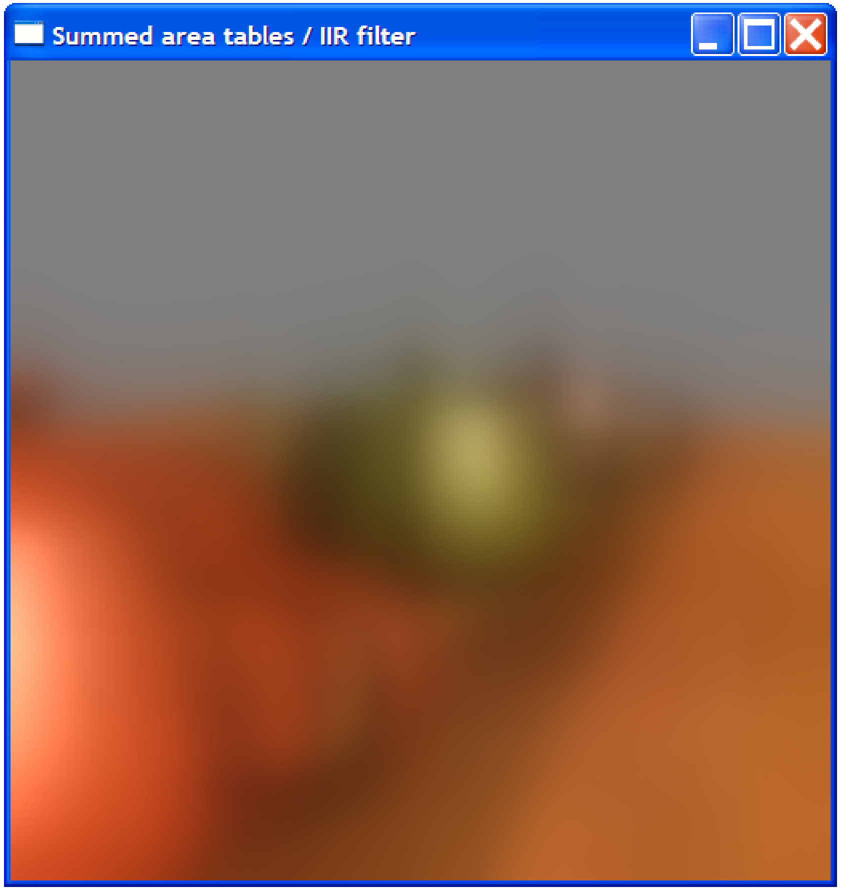
Original Image



Result after Filter in Positive X & Y



Result after Filter in Negative X & Y



Resonant Image Filters

- Second order IIR filters can produce more interesting effects:

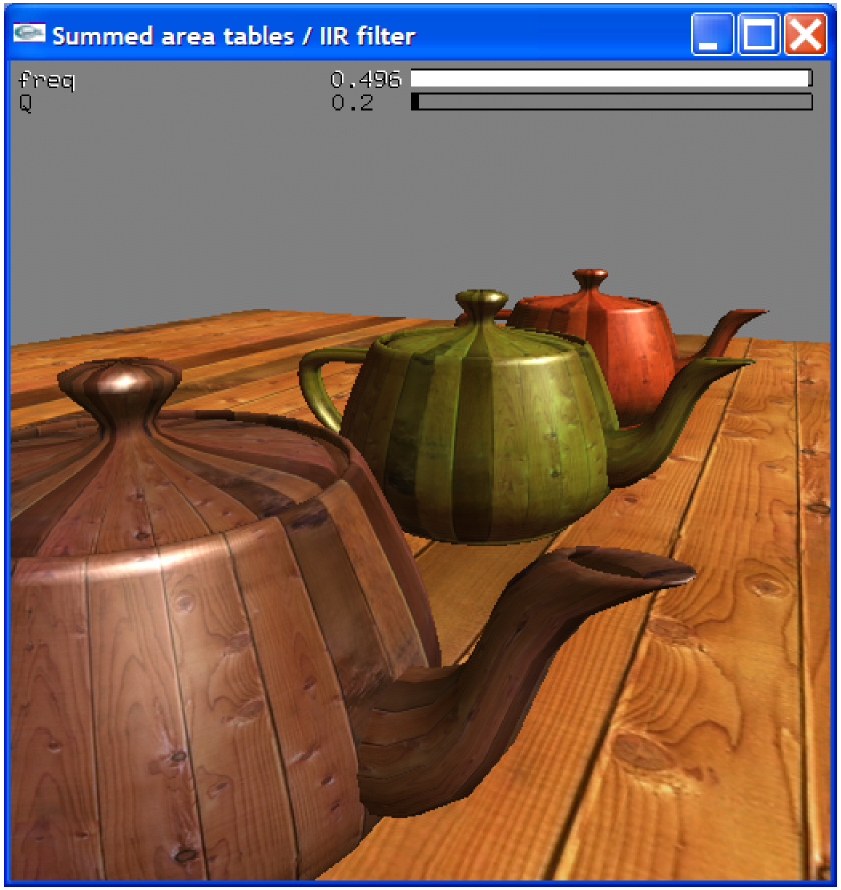
$$y[n] = b0*x[n] + b1*x[n-1] + b2*x[n-2] – a1*y[n-1] – a2*y[n-2]$$

- Close model of analog electronic filters in real world (resistor / capacitor)
  - Act like damped oscillators
- Can produce interesting non-photorealistic looks in image domain

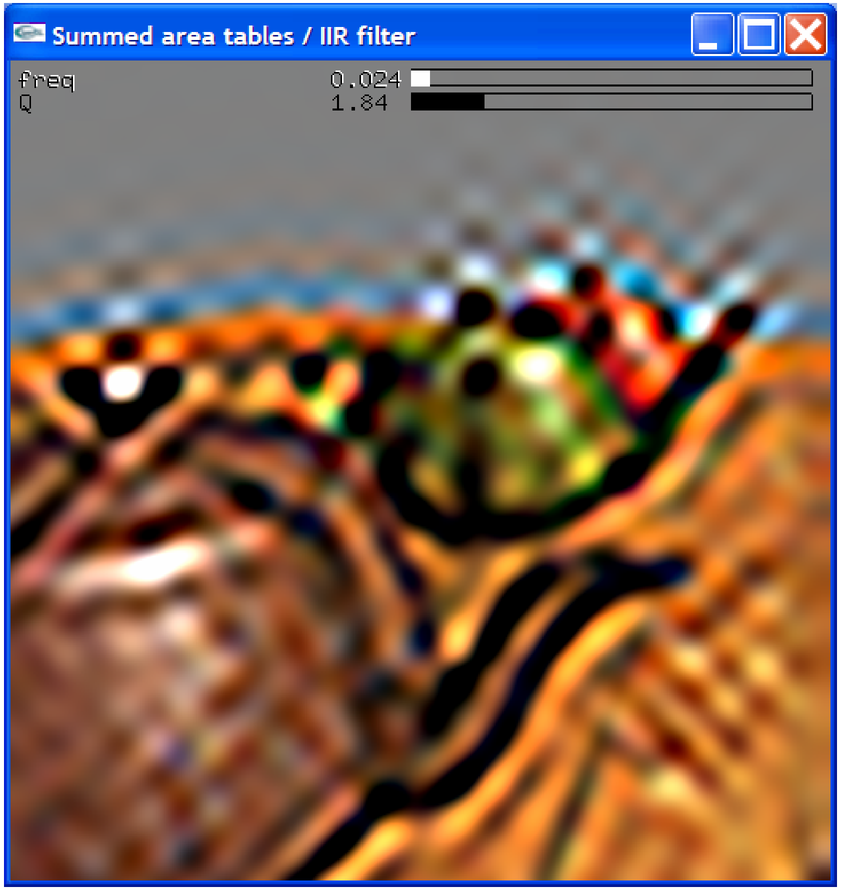
Second Order IIR Filter

```
float4 main(vf30 In,
            uniform samplerRECT y,    // out
            uniform samplerRECT x,    // in
            uniform float4 delta,
            uniform float4 a,         // filter coefficients
            uniform float4 b
            ) : COLOR
{
    float2 n = In.WPOS.xy);     // current
    float2 nm1 = n + delta.xy; // previous
    float2 nm2 = n + delta.zw;

    // second order IIR
    return b[0]*texRECT(x, n) + b[1]*texRECT(x, nm1) + b[2]*texRECT(x, nm2) -
           a[1]*texRECT(y, nm1) - a[2]* texRECT(y, nm2);
}
```

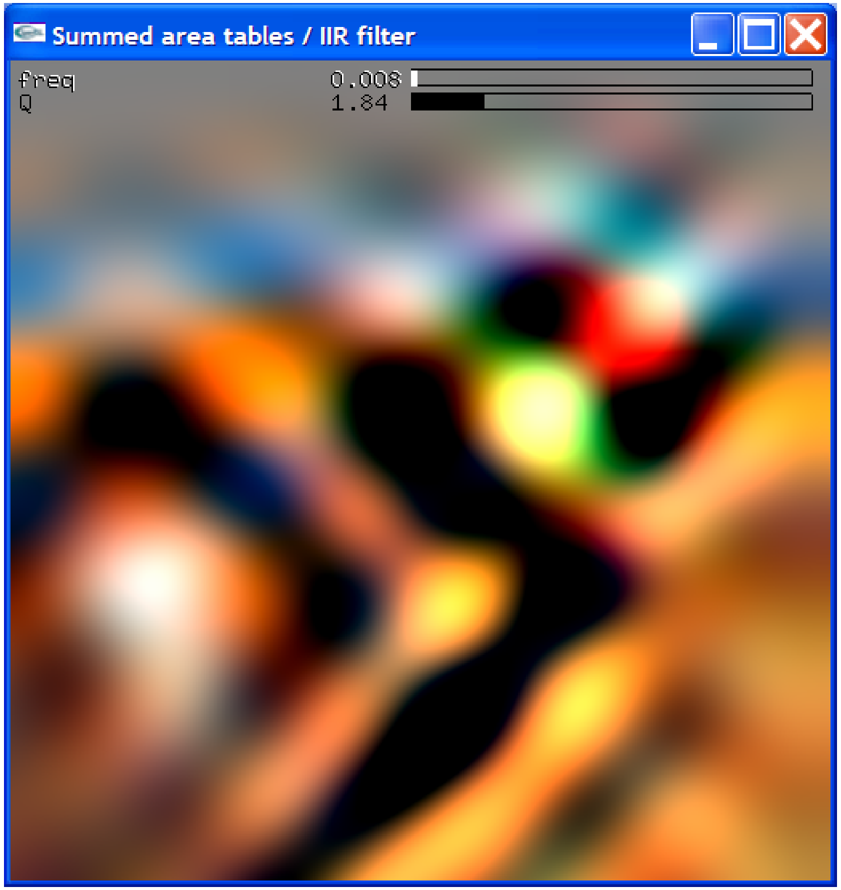
Resonant Image Filters



Resonant Image Filters



Resonant Image Filters



Discrete Cosine Transform

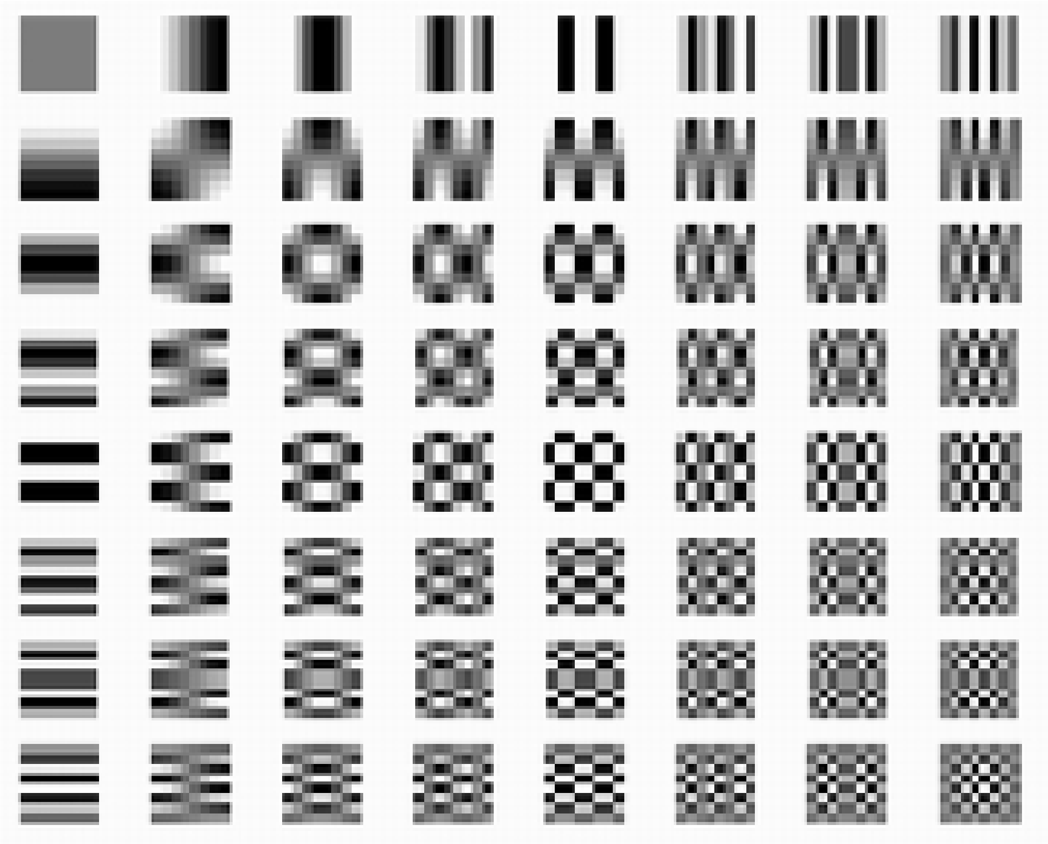
- DCT is similar to discrete Fourier transform
  - Transforms image from spatial domain to frequency domain (and back)
  - Used in JPEG and MPEG compression

$$F(u,v) = \frac{1}{4} C(u)C(v) \left[\sum_{x=0}^{7} \sum_{y=0}^{7} f(x,y) * \cos\frac{(2x+1)u\pi}{16} \cos\frac{(2y+1)v\pi}{16} \right]$$

where: $C(u),\ C(v) = 1/\sqrt{2}$ for $u,v = 0$;

$C(u),\ C(v) = 1$ otherwise.

DCT Basis Images



Performing The DCT in Shader

- Shader implementation based on work of the Independent JPEG Group
  - monochrome (currently)
  - floating point
- Could be used as part of a GPU-accelerated compressor/decompressor
  - File decoding, Huffman compression would still need to be done on CPU
- Game applications
  - None!

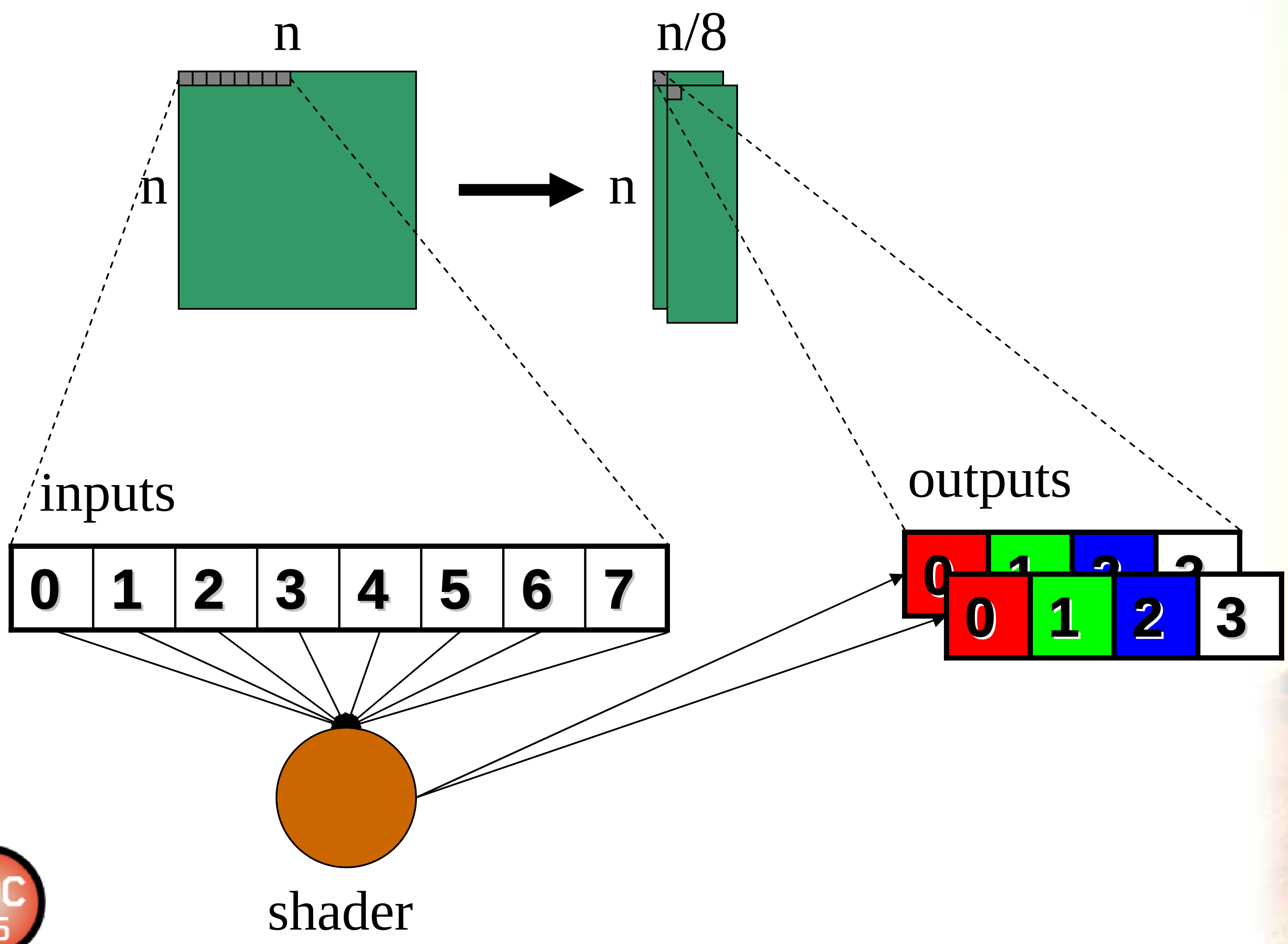
DCT Operation

- DCT used in JPEG operates on 8x8 pixel blocks
  - Trade-off between
- 2D DCT is separable into 1D DCT on rows, followed by 1D DCT on columns
- Arai, Agui, and Nakajima's algorithm
  - 5 multiplies and 29 adds for 8 pixels
  - Other multiplies are simple scales of output values

Partitioning the DCT

- **Problem:**
  - 1D DCT is a function of 8 inputs, produces 8 outputs
- **Shader likes n inputs, 1 output per pixel**
  - don't want to duplicate effort across pixels
- **Solution:**
  - Render quad 1/8th width or height
  - Shader reads 8 neighboring texels
  - Writes 8 outputs to RGBA components of two render targets using MRT
  - Data is unpacked on subsequent passes

Partitioning the DCT (Rows)



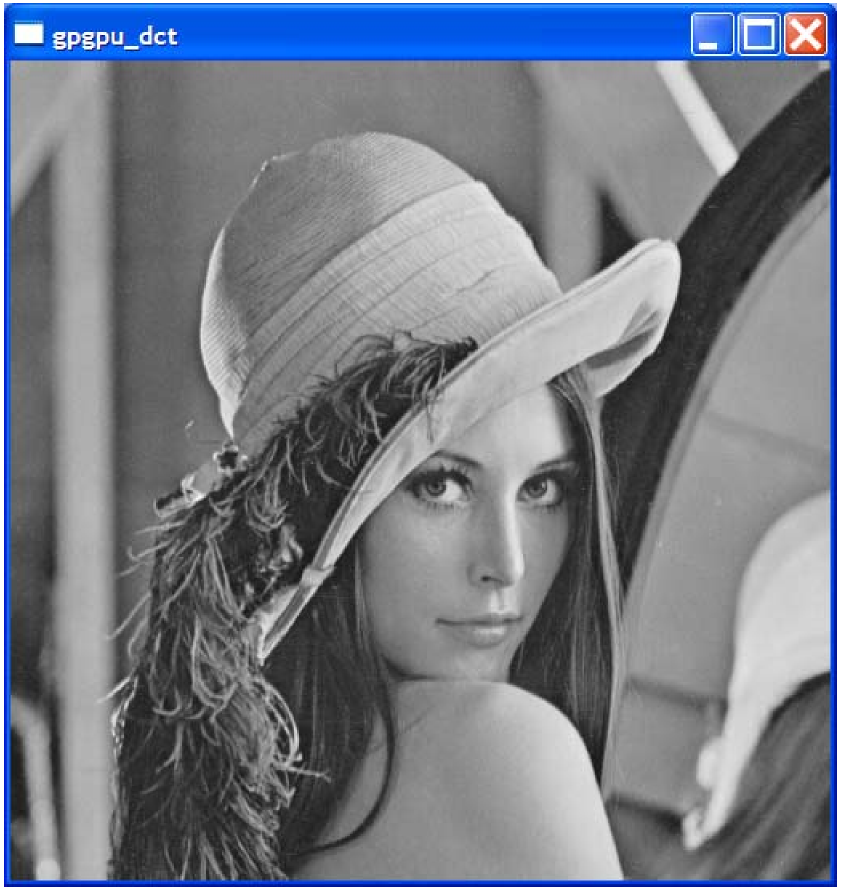
FDCT Shader Code

```
// based on IJG jfdctflt.c
void DCT(float d[8], out float4 output0,
         out float4 output1)
{
     float tmp0, tmp1, tmp2, tmp3, tmp4, tmp5,
           tmp6, tmp7;
     float tmp10, tmp11, tmp12, tmp13;
     float z1, z2, z3, z4, z5, z11, z13;

    tmp0 = d[0] + d[7];
    tmp7 = d[0] - d[7];
    tmp1 = d[1] + d[6];
    tmp6 = d[1] - d[6];
    tmp2 = d[2] + d[5];
    tmp5 = d[2] - d[5];
    tmp3 = d[3] + d[4];
    tmp4 = d[3] - d[4];

    /* Even part */
    tmp10 = tmp0 + tmp3;   /* phase 2 */
    tmp13 = tmp0 - tmp3;
    tmp11 = tmp1 + tmp2;
    tmp12 = tmp1 - tmp2;

    output0[0] = tmp10 + tmp11; /* phase 3 */
    output0[1] = tmp10 - tmp11;

    z1 = (tmp12 + tmp13) * 0.707106781; /* c4 */
    output0[2] = tmp13 + z1;            /* phase 5 */
    output0[3] = tmp13 - z1;

    /* Odd part */
    tmp10 = tmp4 + tmp5;   /* phase 2 */
    tmp11 = tmp5 + tmp6;
    tmp12 = tmp6 + tmp7;

    /* The rotator is modified from fig 4-8 to avoid extra
negations. */
    z5 = (tmp10 - tmp12) * 0.382683433; /* c6 */
    z2 = 0.541196100 * tmp10 + z5; /* c2-c6 */
    z4 = 1.306562965 * tmp12 + z5; /* c2+c6 */
    z3 = tmp11 * 0.707106781; /* c4 */

    z11 = tmp7 + z3;       /* phase 5 */
    z13 = tmp7 - z3;

    output1[0] = z13 + z2; /* phase 6 */
    output1[1] = z13 - z2;
    output1[2] = z11 + z4;
    output1[3] = z11 - z4;
}
```

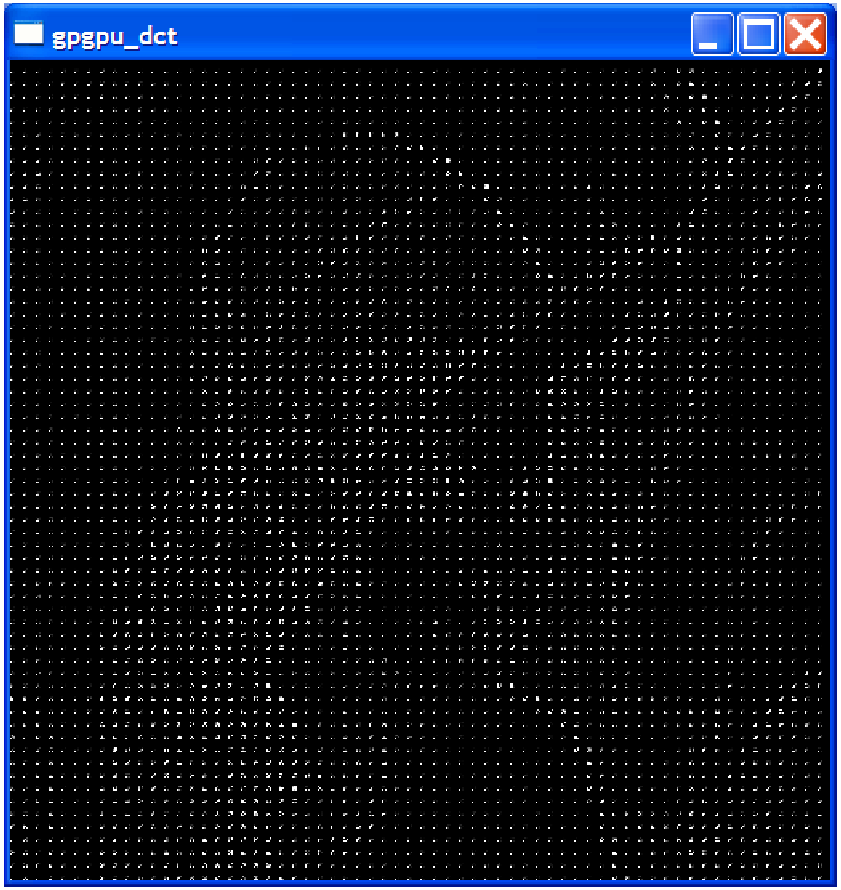
Unpacking Code

```
float4 DCT_unpack_rows_PS(float2 texcoord : TEXCOORD0,
                          uniform samplerRECT image,
                          uniform samplerRECT image2
                          ) : COLOR
{
   float2 uv = texcoord * float2(1.0/8.0, 1.0);
    float4 c = texRECT(image, uv);
    float4 c2 = texRECT(image2, uv);

    // rearrange data into correct order
    //    x y z w
    // c  0 4 2 6
    // c2 5 3 1 7
    int i = frac(texcoord.x/8.0) * 8.0;
    float4 sel0 = (i == float4(0, 4, 2, 6));
    float4 sel1 = (i == float4(5, 3, 1, 7));
    return dot(c, sel0) + dot(c2, sel1);
}
```

Original Image



After FDCT (DCT coefficients)



After IDCT

Performance

- Around 160fps for FDCT followed by IDCT on 512 x 512 monochrome image on GeForce 6800 Ultra
- Still a lot of room for optimization
  - make better use of vector math
  - could process two channels simultaneously (4 MRTs)
- JPEGs are usually stored as luminance and 2 chrominance channels
  - Chroma is at lower resolution
  - Could also do resampling and color space conversion on GPU

Questions?

References

- Infinite Impulse Response Filters on Wikipedia
- "The JPEG Still Picture Compression Standard", Wallace G, Communications of the ACM Volume 34, Issue 4
- Discrete Cosine Transform on Wikipedia