Utilizing texture units for Image Processing

Erik N. Steen PhD

Principal Engineer

GE Vingmed Ultrasound

Why use GPUs for Image processing

- GPU: A tremendous additional «computer within the computer»
  - *Hardware support for basic operations used in image processing algorithms*
  - *Many image processing algorithms are parallel in nature*
  - *Amount of code can sometimes be greatly reduced ..*

Texture access

- Global memory access through separate texture cache
  - ***L2 cache used as well on Fermi GPUs***
- Efficient for localized (1D, 2D 3D) memory access patterns
  - ***Automated handling of borders***
- Includes interpolators
  - ***Interpolators offer additional computational resources***

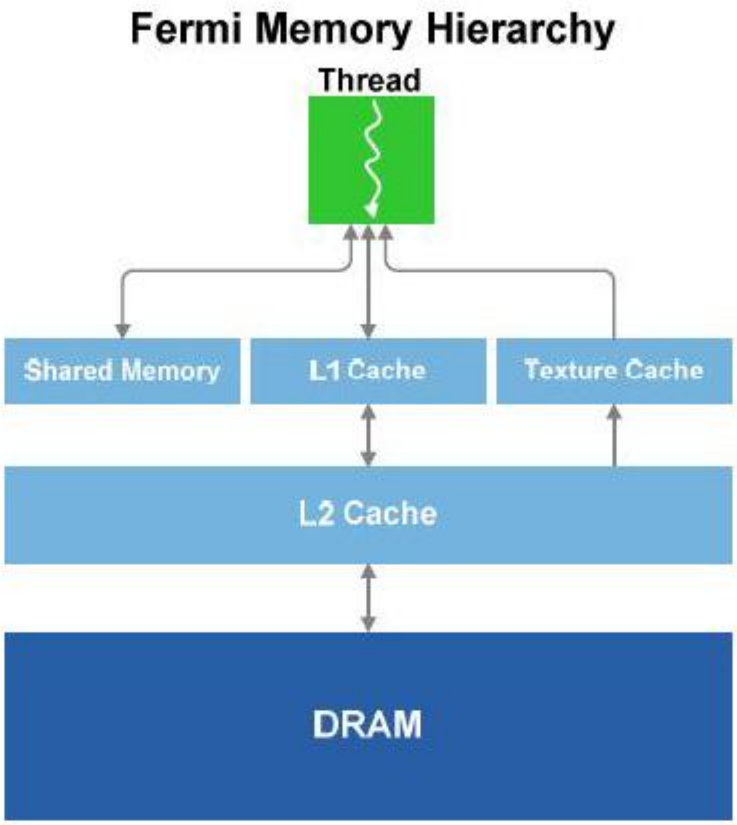

Fermi Memory Hierarchy
Thread
Shared Memory
L1 Cache
Texture Cache
L2 Cache
DRAM

Image processing with texturing

- Image data can be copied to pitched linear memory

OR

- Image data can be copied to a cudaArray

- BUT:
- GPU kernels can only work in parallel with data transfers if:

  - *Copies are made asynchronous (using CUDA streams)*
  - *Pitch linear memory is used.*

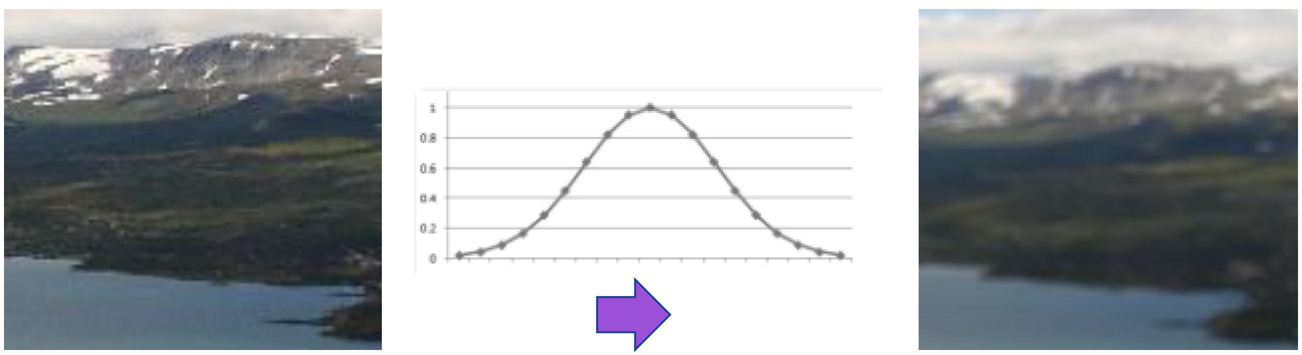
Case 1: Convolution filters

- Convolution filters are very frequently used in image processing for tasks such as:
  - Smoothing
  - Edge enhancement
  - Resampling
- The CUDA SDK contains example code based on
  - Constant memory for coefficients
  - Data access using either texturing or shared memory
- The 2D Gaussian filter can be implemented efficiently as a 2-step separable 1D filter (row-wise & column-wise)

Convolution filters

s(x-4)	s(x-3)	s(x-2)	s(x-1)	s(x)	s(x+1)	s(x+2)	s(x+3)	s(x+4)

$$o(x) = \sum_{i=-4}^{4} w(i) \cdot s(x+i)$$



Example: Gaussian Blur

Filter coefficients can be stored in constant memory

1D convolution (from SDK)

```
__global__ void convolutionRowsKernel(
   float *d_Dst,
   int imageW,
   int imageH )
{
   const   int ix = IMAD(blockDim.x, blockIdx.x, threadIdx.x);
  const   int iy = IMAD(blockDim.y, blockIdx.y, threadIdx.y);

  const float  x = (float)ix + 0.5f;
  const float  y = (float)iy + 0.5f;

  if(ix >= imageW || iy >= imageH)  return;
    float sum = 0;
     // Note template implementation in SDK is more efficient

     for(int k = -KERNEL_RADIUS; k <= KERNEL_RADIUS; k++)
        sum += tex2D(texSrc, x + (float)k, y) * c_Kernel[KERNEL_RADIUS -
     k];

     d_Dst[IMAD(iy,  imageW, ix)] = sum;
}
```

Fast 1D convolution filters

- In several cases, two and two filter taps can be combined:
  - Consider the following filter coefficients (taps):

$$y_0 = 0.0625 \cdot x_0 + 0.250 \cdot x_1 + 0.375 \cdot x_2 + 0.250 \cdot x_3 + 0.0625 \cdot x_4$$

  - Trivially implemented with 5 texture fetches & 5 MADDs

  - Can be re-written using two (texture based) linear interpolations

$$y_0 = 0.3125 \cdot (0.25 \cdot x_0 + 0.75 \cdot x_1) + 0.375 \cdot x_2 + 0.3125 \cdot (0.75 \cdot x_3 + 0.25 \cdot x_4)$$

Now implemented with 3 texture fetches & 3 MADDs

<u>-> Significant speedup, especially for larger filter kernels</u>

Optimized 1D convolution code

SDK Example: «convolutionTexture»

```
__global__ void convolutionRowsKernel(
   float *d_Dst,
   int imageW,
   int imageH )
{
   .
   .
   .
   .
      for(int k = -KERNEL_RADIUS k <= KERNEL_RADIUS; k++)
         // Relative sample position stored in constant memory
          sum +=


         tex2D(texSrc, x + c_Offset[KERNEL_RADIUS – k], y) *
       c_Kernel[KERNEL_RADIUS - k];


     d_Dst[IMAD(iy, imageW, ix)] = sum;
}
```

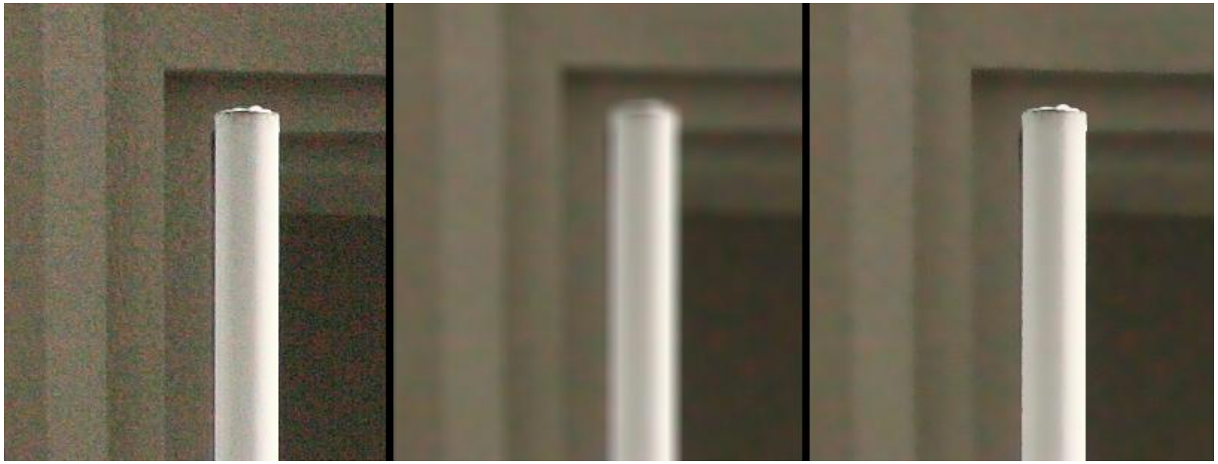
Bilateral (edge preserving) filtering

Gaussian filtering:

- Filtered pixel influenced more by pixels close in space

Bilateral filtering includes one additional criterion:

- Filtered pixel influenced more by pixels close in value



Original Image Gaussian blur Bilateral filtering

Bilateral (edge preserving) filtering

$$BF[I]_{\mathbf{p}} = \frac{1}{W_{\mathbf{p}}} \sum_{\mathbf{q} \in S} G_{\sigma_s}(\|\mathbf{p}-\mathbf{q}\|)\, G_{\sigma_r}(|I_{\mathbf{p}} - I_{\mathbf{q}}|)\, I_{\mathbf{q}}$$

Weights based on distance as well as difference in value

space σ_s : Spatial extent of the kernel

range σ_r : Edge amplitude

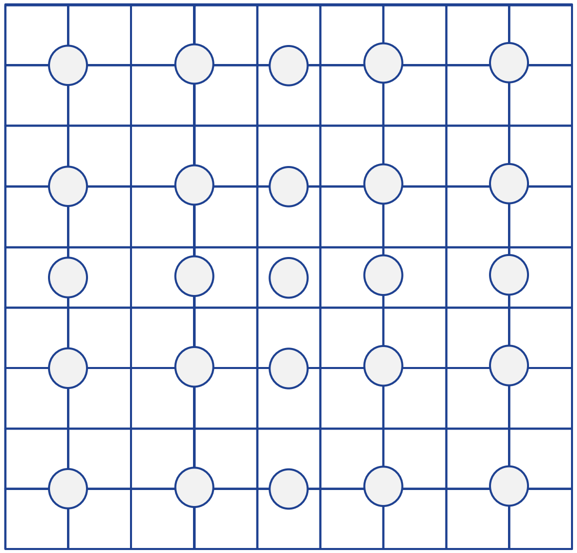
SDK example can be found in "bilateralfilter"

Fast Bilateral filtering

- Signal dependent weights can be stored into a 1D texture
  - Or computed analytically
- Constant memory used for spatial weights and offsets

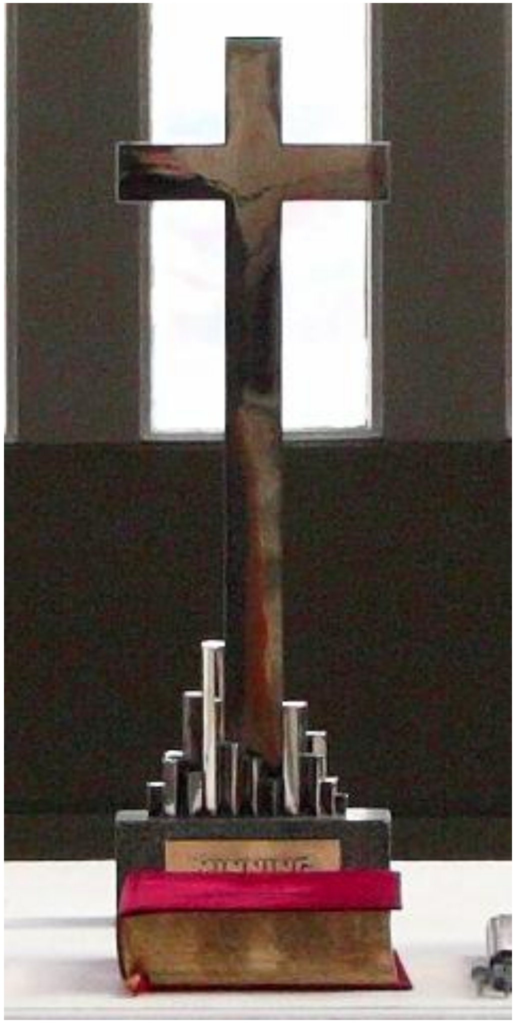
```
float centerValue = tex2D(texSrc,x,y);
float sum = 0.0F;
for(int j = -KERNEL_RADIUS; j < KERNEL_RADIUS;  j++) {
    for(int i = -KERNEL_RADIUS; i < KERNEL_RADIUS;  i++) {
            float2 offset            = c_Offset[KERNEL_RADIUS – j] [KERNEL_RADIUS – i];
            float   rangeWeight   = c_Kernel[KERNEL_RADIUS – j] [KERNEL_RADIUS – i];
             float   value              = tex2D(texSrc,  x + offset.x,  y + offset.y);
            sum+=
             value*rangeWeight*tex1D((texSignalDepWeightLUT,fabs(value-centerValue));
    }
}
```

2D texture convolution sampling



- Example: Using a neighbor distance of 1.5-2.0 samples
  - Covering a 9x9 neighborhood with just 25 samples

Example : Photo noise reduction



Branching

- Despite computational benefits, texture fetch rate is limited compared to the other computational resources, especially on newer GPUs

- Branching can be used to avoid unnecessary texture fetches

- In most real cases, image data is locally coherent

- Data dependent conditional branching is therefore often effective

Example : Shadow area image processing

```
float centerValue = tex2D(texSrc,x,y);
 float sum = 0.0F;
If (centerValue<shadowAreaLimit) { // i.e 0.2
      for(int j = -KERNEL_RADIUS; j < KERNEL_RADIUS;  j++) {
            for(int i = -KERNEL_RADIUS; i < KERNEL_RADIUS;  i++) {
                float2 offset              = c_Offset[KERNEL_RADIUS – j] [KERNEL_RADIUS – i];
                float   rangeWeight = c_Kernel[KERNEL_RADIUS – j] [KERNEL_RADIUS – i];
                float   value               = tex2D(texSrc,  x + offset.x,  y + offset.y);
                sum+= value*rangeWeight*tex1D((texSignalDepWeightLUT,fabs(value-centerValue));
            }
      }
}
else
     sum=centerValue;
```

Speedup on typical images (9x9 lookups): 3-4 x (Fermi GPUs)

Examples of using shared memory

Result of a column-wise 1D filter may be written to shared memory:

- Column-wise filtering with use of texturing, results put in shared memory
- Row-wise filtering done from shared memory
- Reduces global memory bandwidth needs.

Exercise: Change SDK Example: «convolutionTexture»

Spatially invariant filtering can benefit greatly from shared memory when coefficients can be assumed to be constant locally

- Filter coefficients read into shared memory

  or
- Filter coefficient indices read into shared memory

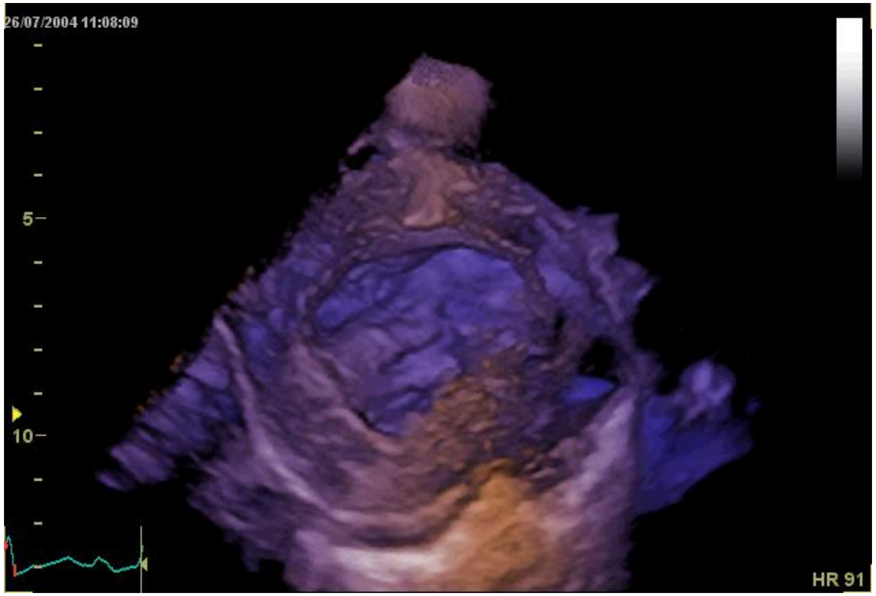
Example: Spatially variant filtering

Spatially variant filtering often useful (i.e lense-specific deblurring of photos)

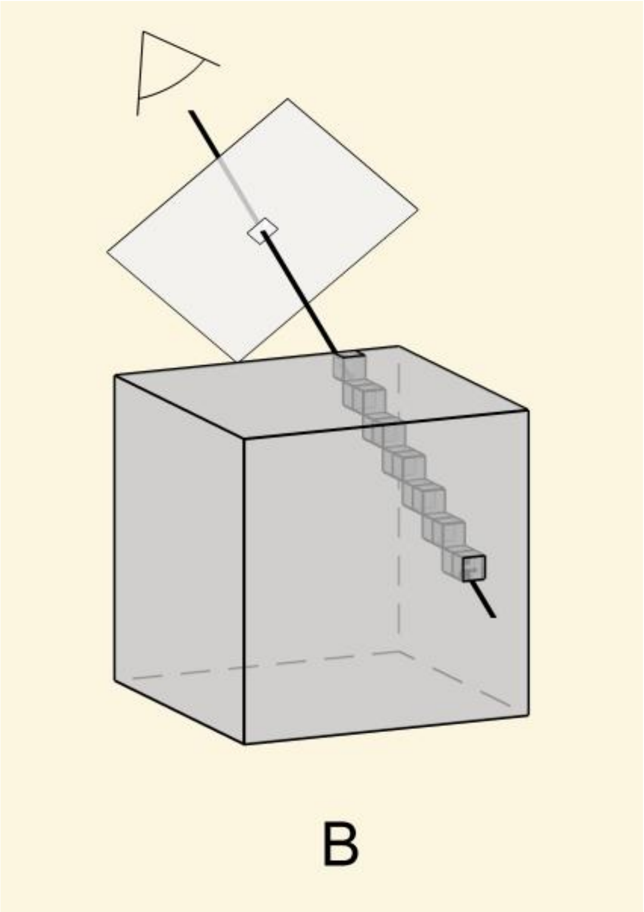
```
.....
// Assuming one index per threadBlock available in global memory and a limited number of coefficient sets in constant mem.
 __shared__ uint s_ind1;
__shared__  uint s_ind2;
 if (threadIdx.x==0 && threadIdx.y==0) {
    s_ind1 = globalOffsetIndexTable[blockIdx.x][blockIdx.y];
    s_ind2 = globalKernelIndexTable[blockIdx.x][blockIdx.y]
  }
  __syncthreads();


   ....
  for(int k = -KERNEL_RADIUS k <= KERNEL_RADIUS; k++)
      sum += tex2D(texSrc, x + c_Offset[s_ind1][KERNEL_RADIUS – k], y) * c_Kernel[s_ind2][KERNEL_RADIUS - k];
  .....
```

Case 2: Volume rendering

Use of 3D texturing and branching



3D rendering principle



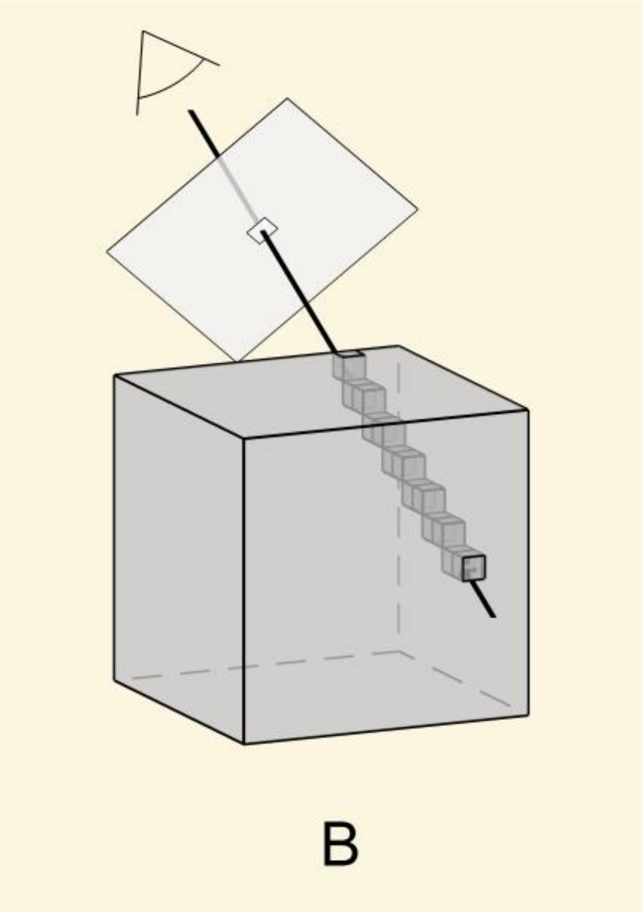
The rendered image is a type of projection of an entire volume.

In its simplest form it can be thought of as a digital x-ray image.

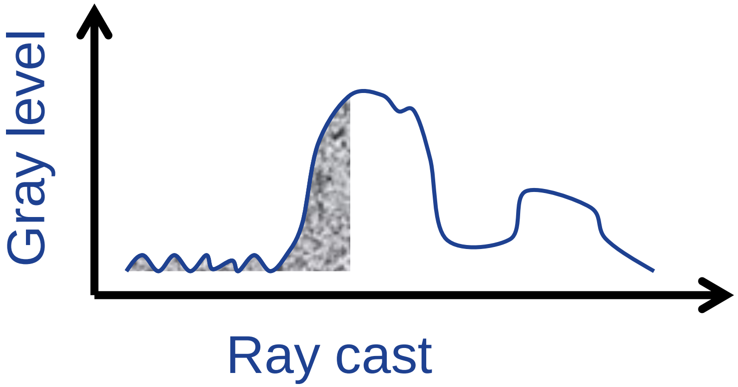
In medical imaging we are usually interested in displaying surfaces of anatomical structures.

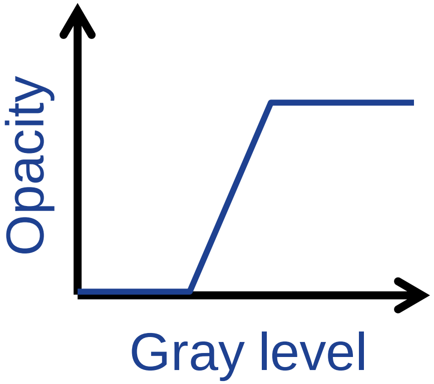
The rendering should be robust and give a good depth perception

Opacity function controls rendering appearance



Weighted integration along rays:





Ray casting loop (Nvidia SDK sample)

SDK Example: «volumeRender»

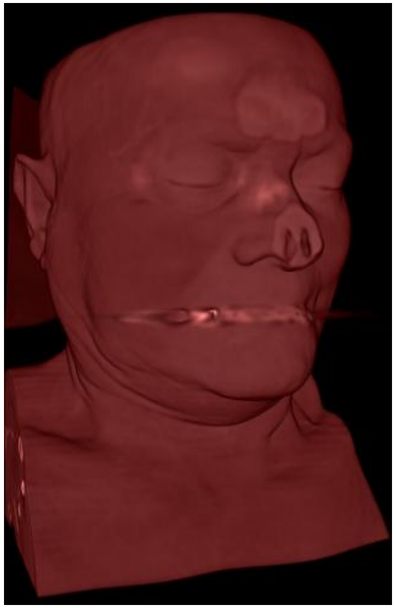
```
float4 sum = make_float4(0.0f);
for (i=0;i<maxSteps;i++) {
{
    // read from 3D texture with remapped positions to [0,1] coordinates
    float3 posNorm = pos*0.5f+make_float3(0.5f,0.5f,0.5f);
    float sample = tex3D(tex, posNorm.x,posNorm.y,posNorm.z);

    // lookup in transfer function texture
    float4 col = tex1D(transferTex, (sample-transferOffset)*transferScale);

    // pre-multiply alpha
    col.x *= col.w; col.y *= col.w; col.z *= col.w;

    // "over" operator for front-to-back blending
    sum = sum + col*(1.0f - sum.w);

    // exit early if opaque
    if (sum.w >= opacityThreshold) break;

    t += tstep; pos += step;
}
```

A simple gradient shading scheme

- Selectively modifying each voxel value using gradient shading
- Each gradient computation uses 6 additional texture fetches
- Gradients normalized and used in lighting calculation

Simple shading

```
__device__ inline float ComputeShadeValue(
        const float3 &pN,
        const float3 &dirX,
        const float3 &dirY,
        const float3 &dirZ,
        const float3 &lightDir)
{
    // Get 6 neighbors in directions aligned with viewing plane
    float v1,v2,v3,v4,v5,v6;
    v1=tex3D(tex,pN.x-dirX.x,pN.y-dirX.y,pN.z-dirX.z);
    v2=tex3D(tex,pN.x+dirX.x,pN.y+dirX.y,pN.z+dirX.z);
    ......

    float3 gradUnNormalized;
    gradUnNormalized.x=v2-v1;
    gradUnNormalized.y=v4-v3;
    gradUnNormalized.z=v6-v5;
    // 25 % ambient and 75 % diffuse reflection
    return
    0.25f+0.75f*fabsf(dot(lightDir,normalize(gradUnNormalized))
    );
}
```



No shading

Simple Gradient based shading

Ray casting loop (w/ simple shading)

```
// read from 3D texture with remapped positions to [0,1] coordinates
float3 posNorm = pos*0.5f+make_float3(0.5f,0.5f,0.5f);

float sample      = tex3D(tex, posNorm.x,posNorm.y,posNorm.z);

// lookup in transfer function texture
float4 col        = tex1D(transferTex, (sample-transferOffset)*transferScale);

if (sample>threshold) {        // Selective shading
    float shadeVal      =  ComputeShadeValue(posNorm,eyeRay.dx,eyeRay.dy,eyeRay.dz,c_lightDir);
    col.x*=shadeVal;
    col.y*=shadeVal;
    col.z*=shadeVal;
}
```

The actual threshold value may impact performance substantially !

Questions ?