GPU based Medical Imaging

강 성태
Computer Graphics & Medical Imaging Lab., Seoul National University

Overview

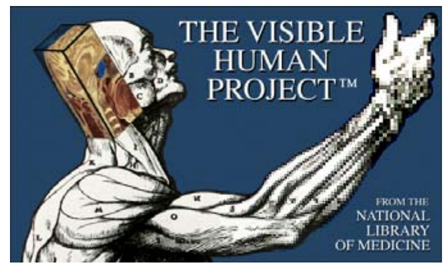
- GPUs and Medical Imaging
- Case Study : Cone-beam CT Reconstruction

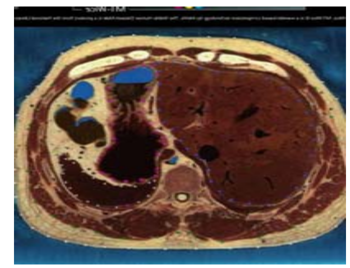
Medical Imaging

- Data Acquisition
  - CT, MRI, PET, SPECT, Ultrasound
  - Tomographic reconstruction of acquired data
- Image processing
  - Segmentation
  - Registration
- Visualization
  - Direct Volume Rendering
  - Maximum Intensity Projection

Issues on Medical Imaging Applications

- Storage and memory usage
  - Medical data is huge, and getting larger
    - Visible Human Project (1994)
      - 15GB
      - 65GB rescanned in 2000
    - Time-series data





  - The highest level of accuracy is required
- Performance
  - Most medical imaging applications are based on heavy-computational algorithms...
    - Frequency-domain analysis, filtering, optical integration, ...
  - ...Over large data

GPUs and Medical Imaging

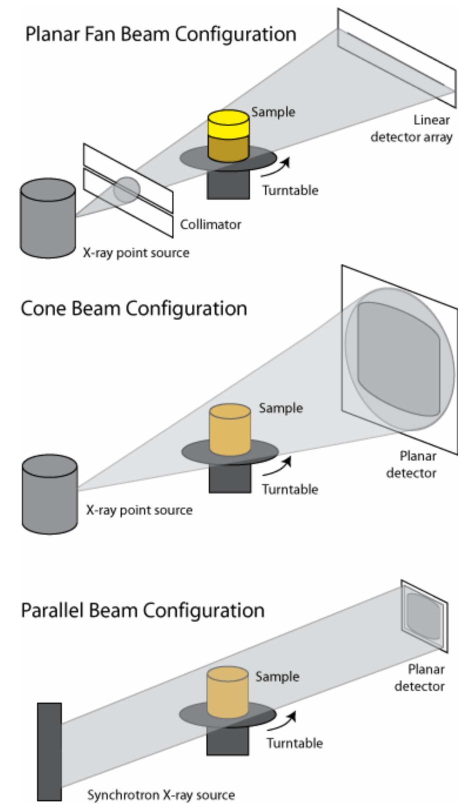
- Parallelism
  - Simple but heavy calculation over huge domain
  - No or less data dependency between output data
    - Filtering
    - Projection
- GPU-friendly operations
  - Interpolation
  - Blending

GPUs and Medical Imaging

- The first flexible shader (Robert L. Cook, 1984)
- Accelerating Volume Reconstruction with 3D Texture Hardware (T. Cullip and U. Neumann, 1993)
- Accelerated Volume Rendering and Tomographic Reconstruction using Texture Mapping Hardware (B. Cabral, N. Cam, and J. Foran, 1994)
- Programmable shader introduced to consumer H/W (Microsoft DirectX 8, 2000)

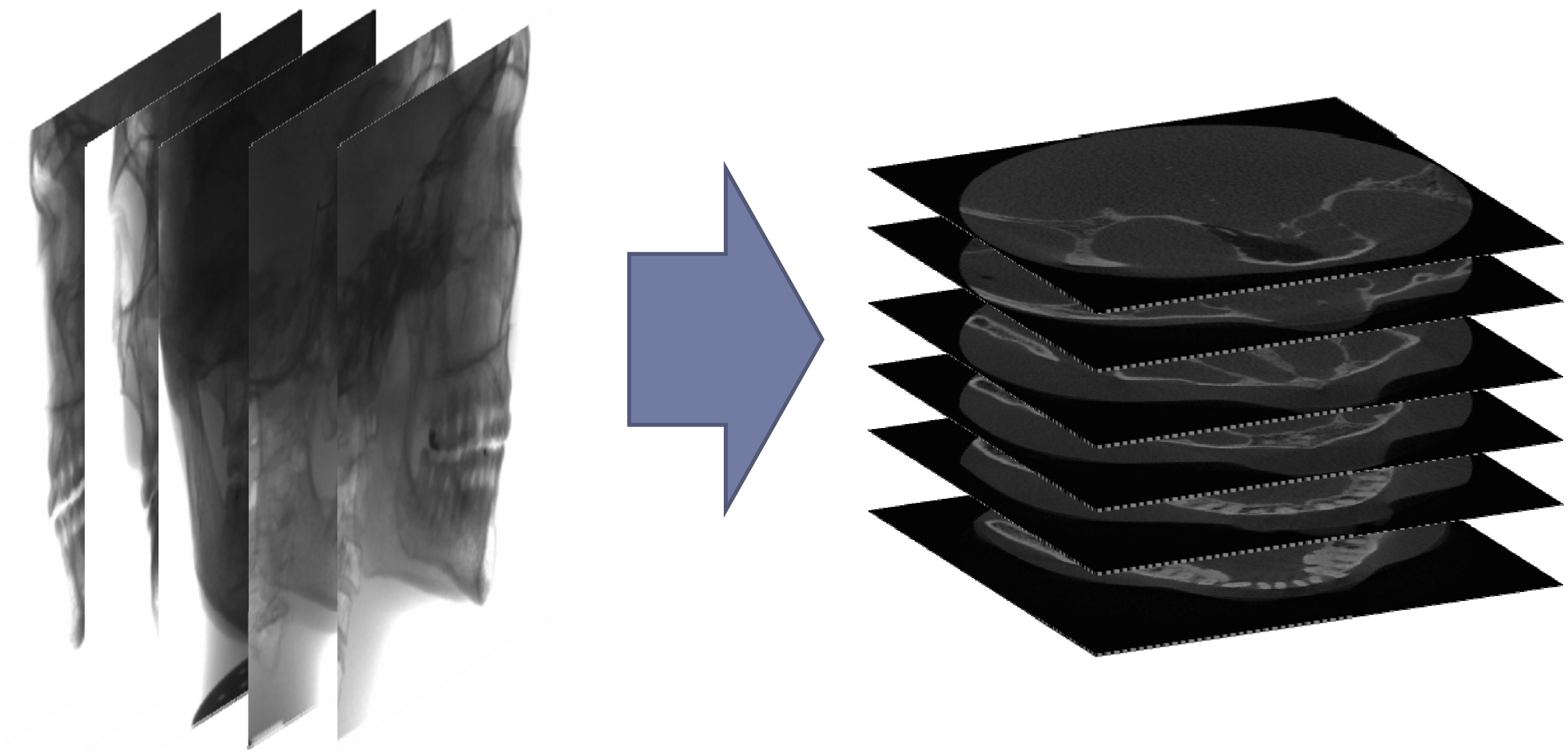
Case Study : CT Reconstruction

- Computed Tomography
  - 3D image of inside of an object from a large series of 2D X-ray images taken around a single axis of rotation



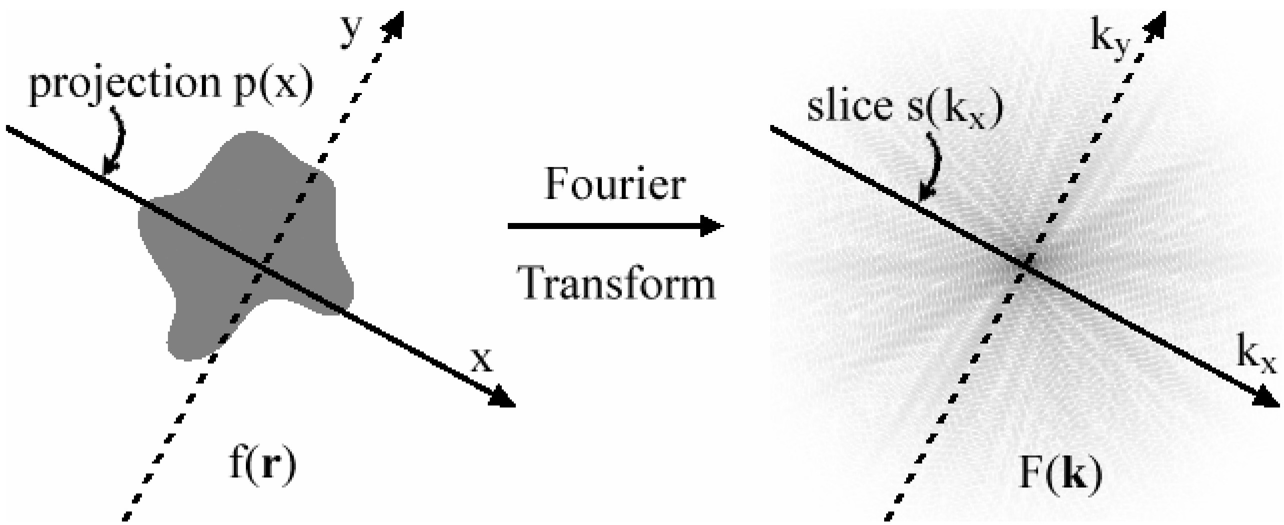
CT Reconstruction

- Reconstruction of 3D data from 2D projected images



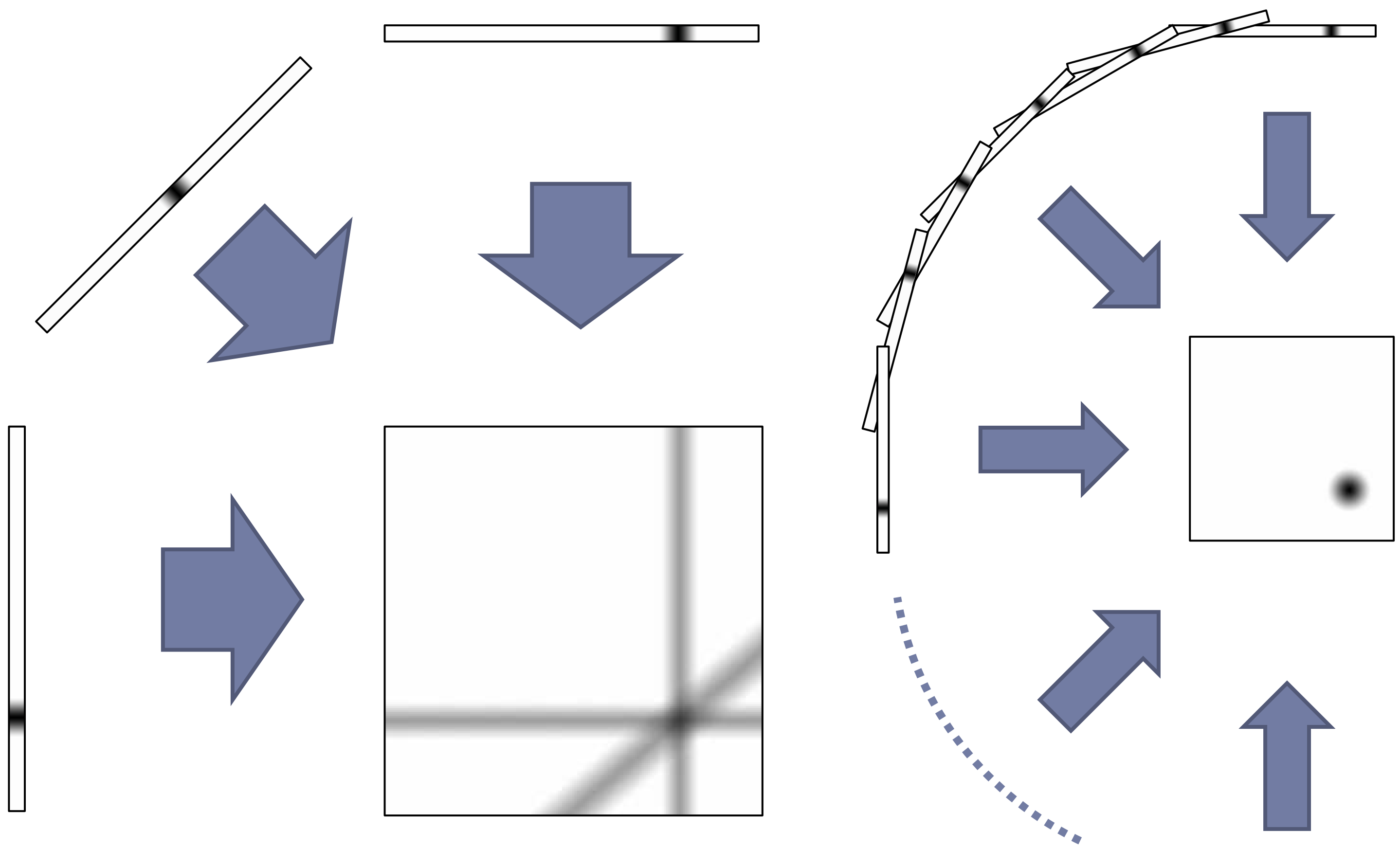
CT Reconstruction

- Fourier Slice Theorem
  - Fourier transform of the projection of an n-D function is equal to a slice of n-D Fourier transform of that function along the origin in the Fourier space



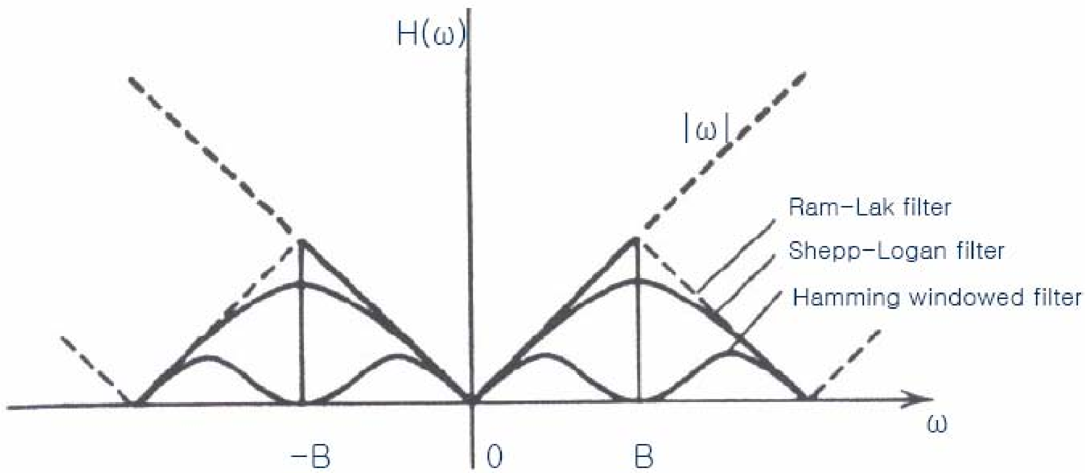
CT Reconstruction

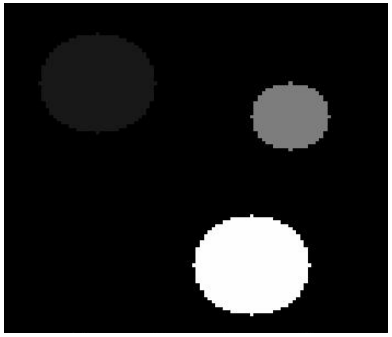
- Backprojection method



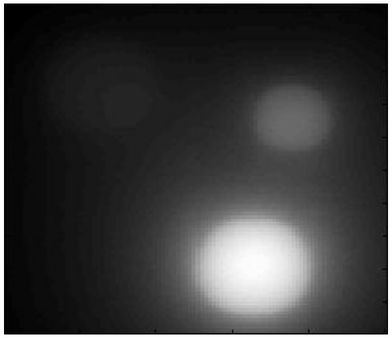
Filtered Backprojection

- Blurring Artifact
  - Overweighting on low-frequency area
- Filtering
  - high-pass filters



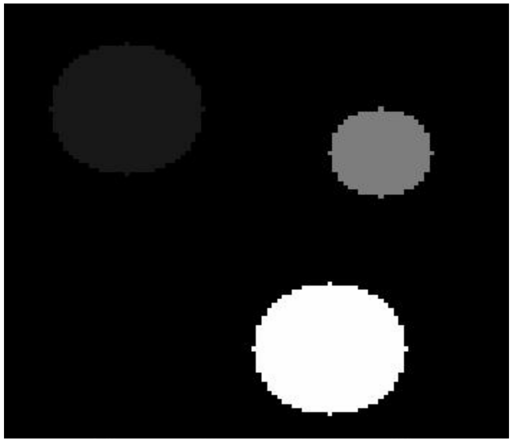


Source

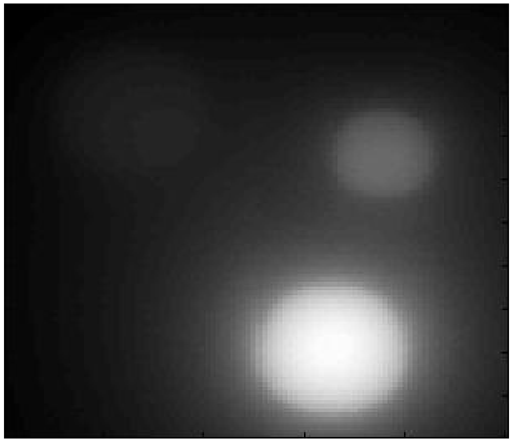


Backprojected

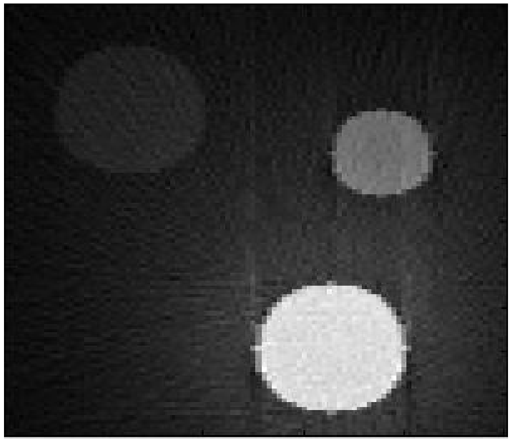
Filtering



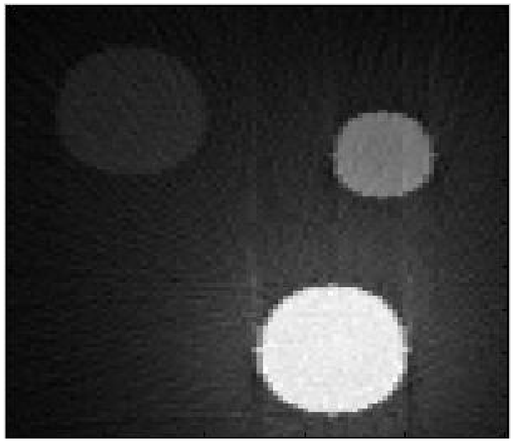
Source



No filtering



Ram-Lak filtered



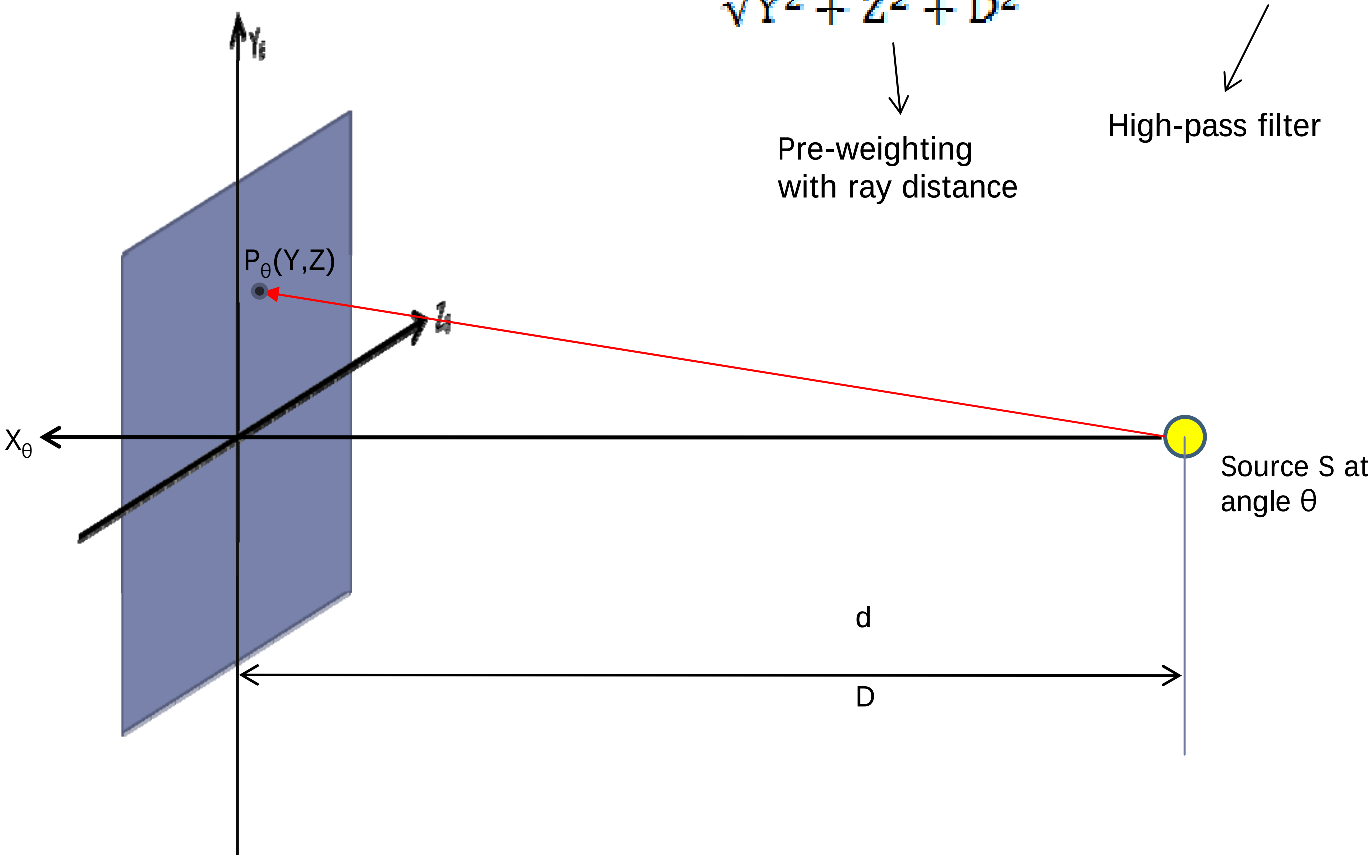
Shepp-Logan filtered

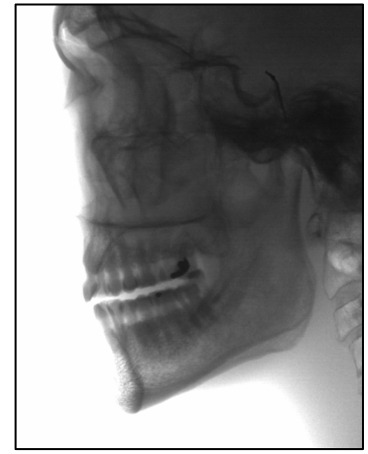
Filtering (Cone-beam)

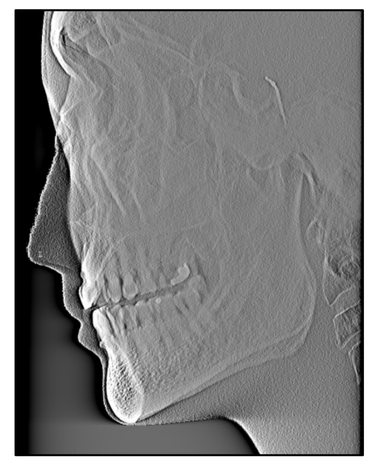
$$\hat{P}_\theta(Y,Z) = \frac{D}{\sqrt{Y^2+Z^2+D^2}} P_\theta(Y,Z) * h(Y)$$

Pre-weighting
with ray distance

High-pass filter







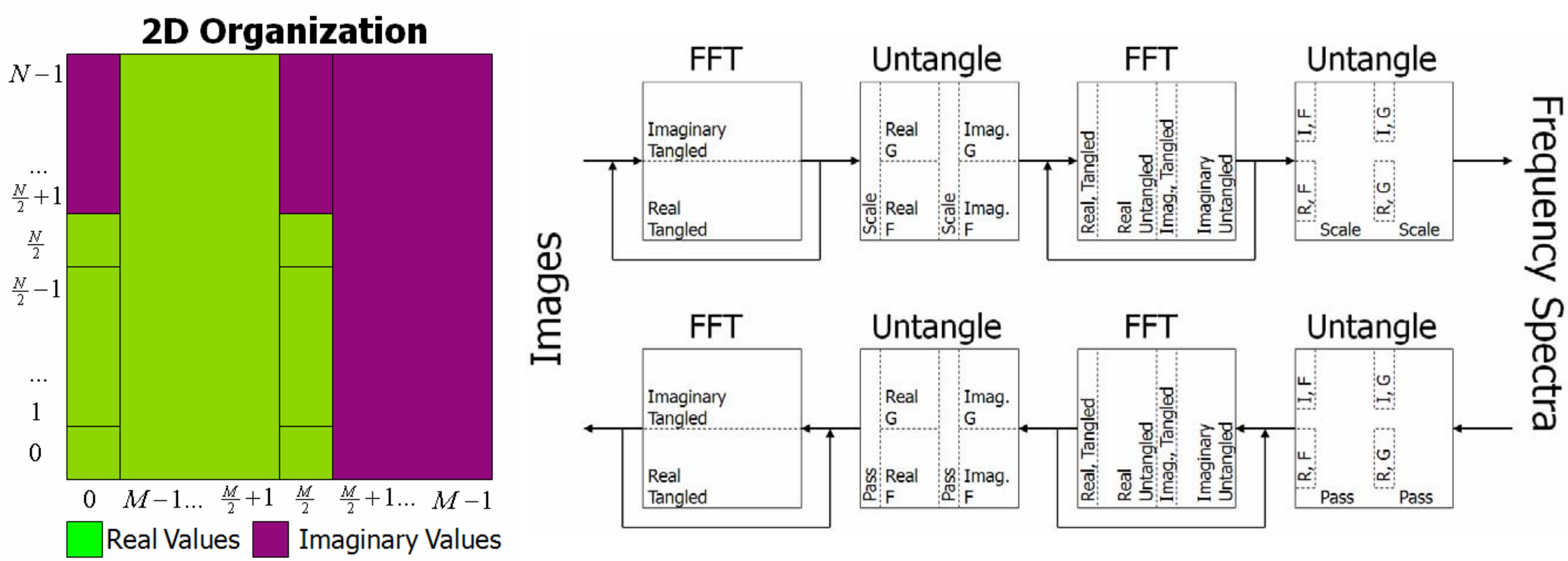
Filtering

- Brief algorithm

```
for each image in detected images
{
    FT
    for each pixel in the image
        multiply high-pass filter
    inverse FT
    for each pixel in the image
        multiply pre-weighting factor
}
```

GPU-based Filtering

- FFT on a GPU (K. Moreland and E. Angel, 2003)
  - Packing real values into real-imaginary pairs
  - Slower than CPU implementation at that time



GPU-based Filtering

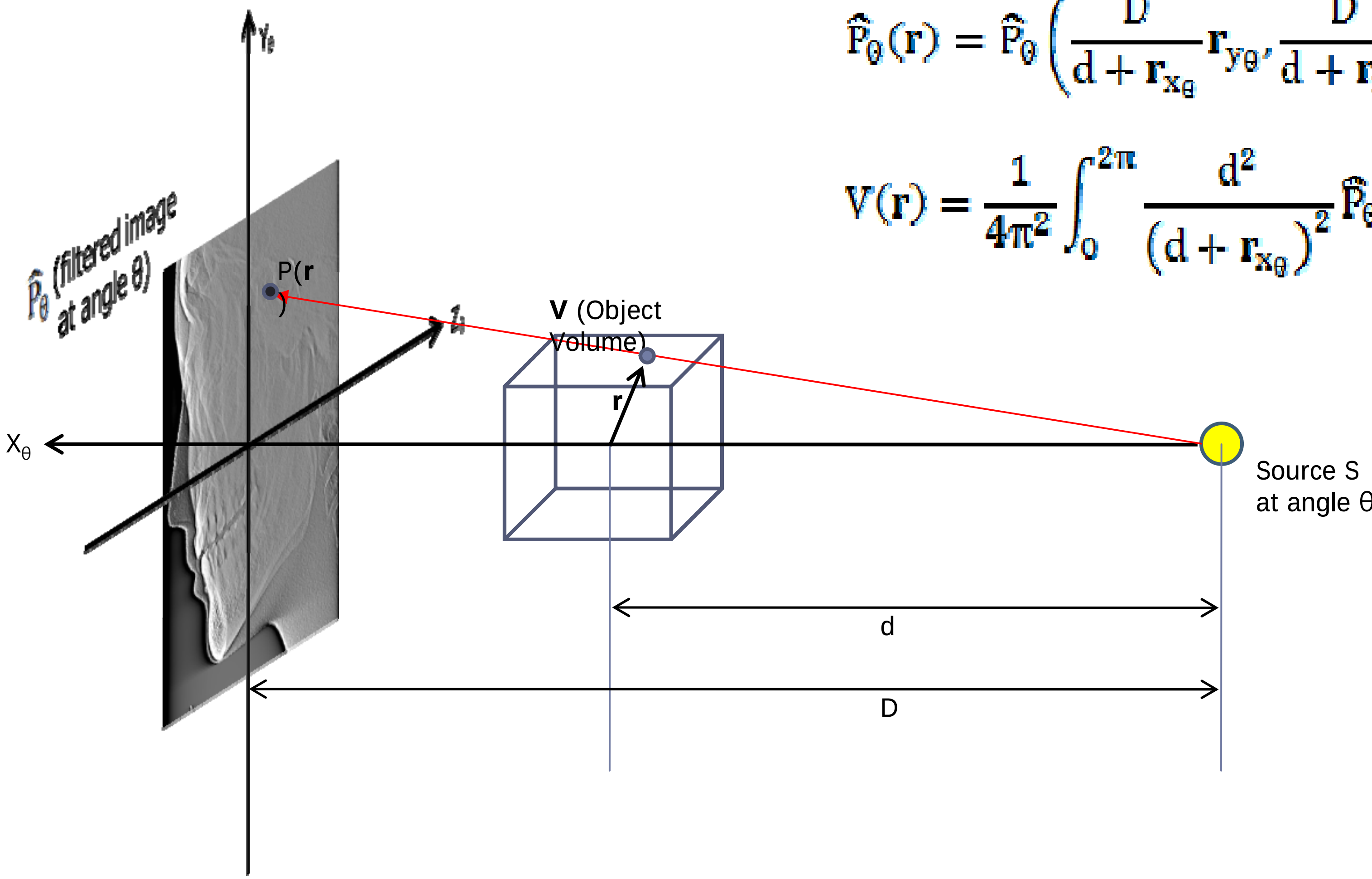
- FFT on a GPU (Sumanaweera and Liu, 2005)
  - 1.3~3 times faster than CPU based FFT
- GPU implementation was really complicated and tricky
- CUDA cufft library
  - Library for fast fourier transform using GPU resource
  - Simple function call
  - Nvidia's black-box algorithm

GPU-based Filtering

- CUDA-based implementation

```
__global__ void shepp_logan(cufftComplex sourceImage)
{
                              :
                              :
}

__host__ void filter(cuffReal **image, float2 size)
{

  Dim3 threadDim={BLOCK_SIZE, BLOCK_SIZE};
  Dim3 blockDim = {size.x/BLOCK_SIZE, size.y/BLOCK_SIZE};

  cufftPlan2d(&forward_plan, size.x, size.y, CUFFT_R2C);
  cufftPlan2d(&inverse_plan, size.x, size.y, CUFFT_C2R);

  for(int n=0;n<numOfDetectedImages;n++)
  {
     cufftExecR2C(forward_plan, image[i], freqImage);

     shepp_logan<<<blockDim, threadDim>>>(freqImage);

     cufftExecC2R(inverse_plan, freqImage, image[i]);
  }
                              :
                              :
```

FDK Backprojection (Cone-beam)



$$\hat{P}_\theta(\mathbf{r}) = \hat{P}_\theta\left(\frac{D}{d+\mathbf{r}_{x_\theta}}\mathbf{r}_{y\theta}, \frac{D}{d+\mathbf{r}_{x_\theta}}\mathbf{r}_{z\theta}\right)$$

$$\mathbf{V}(\mathbf{r}) = \frac{1}{4\pi^2}\int_0^{2\pi}\frac{d^2}{\left(d+\mathbf{r}_{x_\theta}\right)^2}\hat{P}_\theta(\mathbf{r})\,d\theta$$

FDK Backprojection

- Brief algorithm

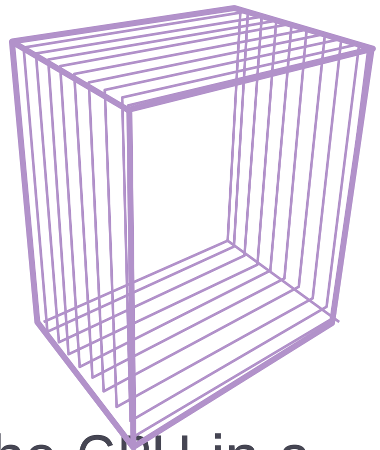
```
for each image I in detected images
{
    Calculate the transform matrix M
        which maps voxel coordinates to projected image coordinate
    for each voxel v with coordinate v of the volume
    {
        Calculate projected coordinate p = Mv
        Sample
        v += weight × I(p)
    }
}
```

GPU-Based Backprojection

- DX/OpenGL Programmable shader vs. CUDA
  - Accumulation of the value
    - CUDA doesn't support normal graphics raster operation like blending
    - Can be alternated with atomic operations, but limited in current version of CUDA
      - Only for CUDA 1.1 compatible H/W
      - Integer only
  - Manipulating volume data
    - Volume texture is not yet supported by CUDA
    - Can be implemented by combination of bilinear sampling

GPU-based Backprojection

- Reconstruction cube representation
  - Stack of 2D texture render targets
  - 3D texture
    - Supported by D3D10 compatible H/W



- Detected image representation
  - 2D texture
  - Only one detected image is loaded to the GPU in a rendering pass, for lack of GPU memory
    - Causes frequent context switching and CPU-GPU memory transfer

GPU-based Backprojection

- Calculating the sampling coordinate with transformation matrix
  - Mapping 3D voxel coordinates to 2D pixel coordinates of detected images
  - Rotation of the detector and perspective projection of cone-beam ray can be represented as composition of simple transform matrices
- Sampling a pixel is trivial
  - Hardware accelerated linear interpolation
- Accumulate weighted pixel to the voxel
  - Color/alpha blending

GPU-based Backprojection

- D3D implementation

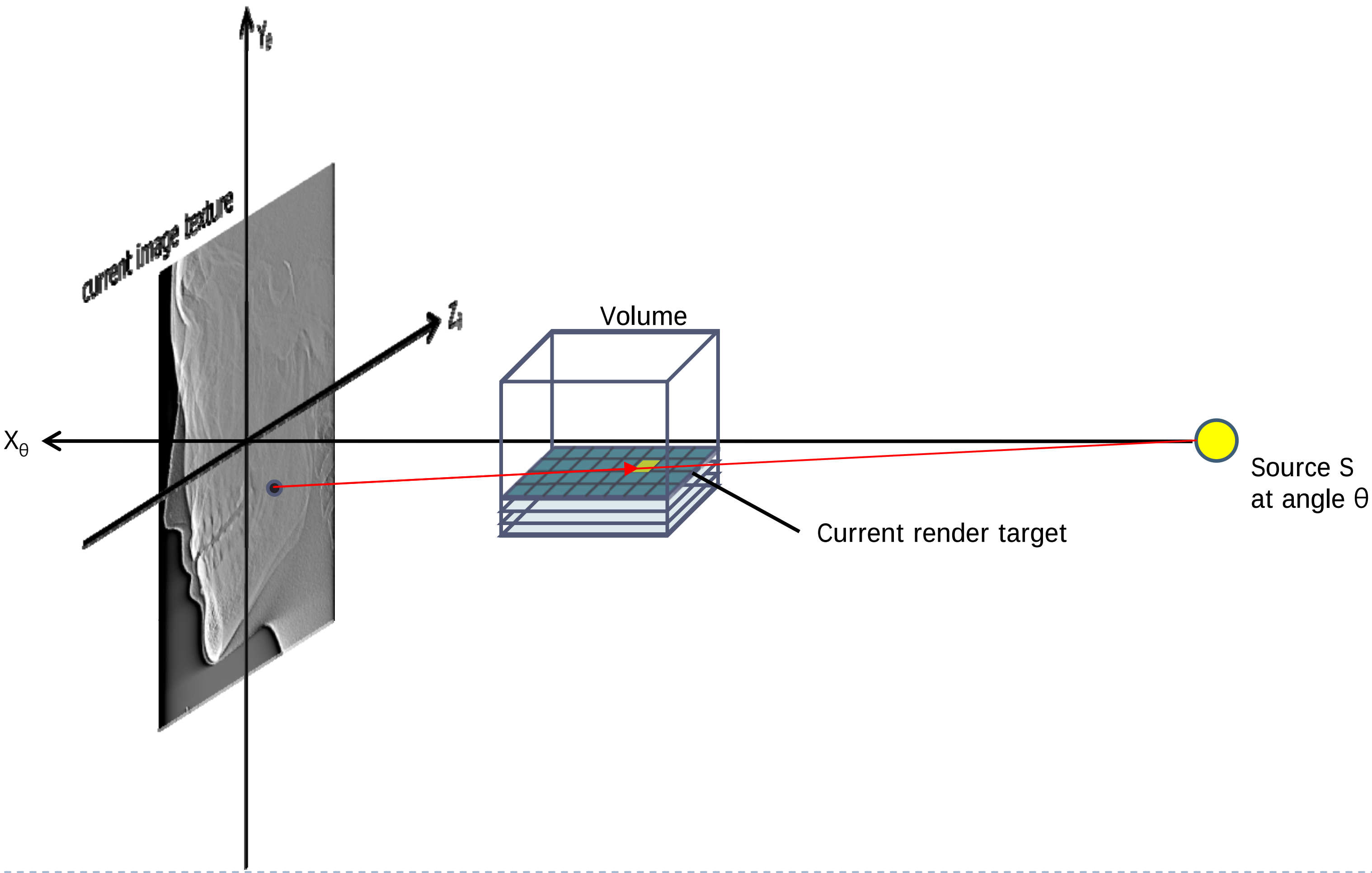
```
Prepare  (volume slice / 4) 4-channel render targets,
   each of those render targets represents 4 slices of volume
for each image I in detected images
{
        Load I to GPU memory (2D texture)
        for each render target R in the stack of render targets
        {
                Draw a rectangle to R
                [
                Vertex Shader :
                        Calculate projected coordinate p of four vertices
                Pixel Shader :
                        Sample the image I using raterized coordinate of p
                                            for current raster pixel, that is identical to a voxel
                        Output weighted sample value to output blender with ADD blend option
                ]
        }
}
```

CUDA-D3D9 Interoperability

- CUDA-D3D interoperability is limited to vertex buffers yet (1.1)
- Filtered image transfer from CUDA to D3D9
  - CUDA Array → CPU memory → D3D9 texture
- GPU→CPU transfer
  - cufft operations can hide upload latency
- CPU→GPU transfer
  - one image per rendering pass
  - Can be hidden with D3D9 reconstruction operations

GPU-based Backprojection

- Rendering pass



Result

- 712 detected images, 720x924 resolution
- 512x512x608 reconstructed cube
- Intel Core 2 Quad Q6600 CPU
- Nvidia GeForce 8800GTX GPU

	Filtering	Reconstruction	GPU-CPU transfer	Total
CPU (Multithreaded)	21.8 sec	352.3 sec	-	374.1 sec
GPU (CUDA+D3D9)	7.9 sec	6.2 sec	3.0 sec	17.1 sec

GPU based Medical Imaging Issues

- Memory size
  - GPU texture memory is not so large as main memory
    - Still insufficient for some medical data
    - 512~1GB for flagship model
      - Nvidia 8800GTX(768MB)
      - 128~256MB in general
- Memory transfer performance
  - Data transfer between GPU memory and system memory depends on bus bandwidth
    - AGP 8x : upload ~100MB/s, download ~2.1GB/s
    - PCI express 16x : up/down ~4GB/s

→ *just theoretical*

Experimental: up 700MB/s, down 1.2GB/s