# Real-Time Radon Transform via the GPU Graphics Pipeline

Christian B. Mendl

**Abstract**

Graphics processing units (GPUs) found in consumer hardware have emerged as first choice for computational-intensive image processing tasks, e.g., real-time computer tomography (CT) reconstruction. However, the GPU programming paradigm is rather intricate. To leverage the potential of GPUs for a broad audience, this paper is concerned with a self-contained, pedagogical implementation of the prototypical forward and inverse Radon transform. The full source code is available for download. Instead of utilizing the GPU merely as multiprocessor, the algorithm explicitly employs the graphics pipeline to take advantage of the dedicated GPU hardware components.

## 1. Introduction

Interactive computer tomographic (CT) imaging techniques in medical and physical sciences have recently become the focus of active research, for example, to enable assistance during surgery [1], for monitoring, or in (closed-loop) feedback experiments. The associated real-time data processing tasks can profit considerably from the broad availability and exponential performance growth of consumer graphics processing units (GPUs) [2, 3]. However, the GPU programming paradigm is still rather intricate as compared to traditional CPU implementations. To render this topic accessible to a broad audience, this paper provides a self-contained, pedagogical implementation of the prototypical forward and inverse Radon transform [4, 5] on GPUs. The full source code is available for download [6] and includes a real-time demonstration program.

The paper is organized as follows. Section 2 explains the theoretical background and contains a detailed description of our algorithmic implementation for both the forward and inverse Radon transform. We evaluate the performance of our program in section 3 and estimate the GPU speedup as compared to a fast CPU. Finally, section 4 draws concluding remarks.

*Email address:* christian_mendl@hotmail.com (Christian B. Mendl)

## 2. Implementation

This section contains the technical description of our implementation. It is subdivided into the forward and inverse Radon transform, which can be handled independently. In each subsection, we first state the well-known classical theory [4, 7, 5] and establish our notation, and then explain the mapping to the graphics pipeline of modern GPUs. There, the only essential hardware requirement is support for floating point arithmetic.

### *2.1. Forward Radon transform*

The forward Radon transform $R$ of a continuous function $f : \mathbb{R}^2 \to \mathbb{R}$ with bounded support may be defined by the line integral

$$(Rf)(\theta, t) := \int\limits_{\hat{n}_\theta \cdot x = t} f(x)\, \mathrm{d}m(x), \quad \theta \in [0, 2\pi), \quad t \in \mathbb{R} \tag{1}$$

with $\hat{n}_\theta := (\cos\theta, \sin\theta)$ and the 1-dimensional "hyperplane" measure $\mathrm{d}m$. In other words, the line contains the point $t\,\hat{n}_\theta$ and is perpendicular to $\hat{n}_\theta$. Figuratively, $R$ describes a set of parallel projections of $f$ at a certain angle $\theta$, as illustrated in Fig. 1.

The implementation we present here is a direct mapping of (1) to the graphics pipeline of modern GPUs, as illustrated in Fig. 2. The discretized version of $f$ is an image or "texture", which we assume

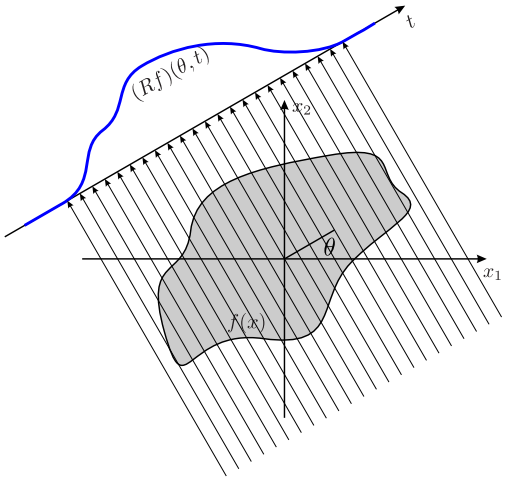


Figure 1: Parallel projections of an image $f : \mathbb{R}^2 \to \mathbb{R}$, for an angle $\theta$.

to be square for simplicity of presentation. Denote its width and height by $h$. Furthermore, we have to discretize the rotation angle $\theta$, i.e.,

$$\theta_i := \frac{i}{m_\theta}\,\pi, \quad i = 0, 1, \ldots, m_\theta - 1 \tag{2}$$

with some fixed integer $m_\theta > 0$. To accommodate all parallel projections of the input texture for any rotation angle requires line segments of length $\sqrt{2}\,h$. Thus, the output texture of the algorithm has dimensions $\lceil\sqrt{2}\,h\rceil \times m_\theta$, with row $i$ corresponding to rotation angle $\theta_i$. To register this texture as output of the graphics processing stages, we set it as *render target* of the GPU.

Our algorithm consists of the following steps: Allocate a vertex buffer containing $\lceil\sqrt{2}\,h\rceil$ "screen quads", i.e., pairs of triangles which form a rectangle. The vertex position coordinates are chosen such that each quad precisely covers the render target texture. Additionally, each vertex contains texture coordinates $(\text{tex.x}, \text{tex.y})$. These are used by the pixel shader to calculate the sampling coordinates $(u, v)$ for the input texture, as follows:

$$\begin{pmatrix} u \\ v \end{pmatrix} := \begin{pmatrix} \cos\theta_i & \sin\theta_i \\ -\sin\theta_i & \cos\theta_i \end{pmatrix} \begin{pmatrix} \text{tex.x} - 0.5 \\ \text{tex.y} - 0.5 \end{pmatrix} + \begin{pmatrix} 0.5 \\ 0.5 \end{pmatrix}.$$

The addition/subtraction of 0.5 shifts the rotation fixpoint to the center of the texture. The index $i$ is equal to the $y$-coordinate of the current render target pixel, and is thus available to the pixel shader.

To obtain the projection line integrals, we assign the same tex.y coordinate to the four vertices forming a single screen quad, such that the collection of all screen quads samples the projection lines. More precisely, for all four vertices of screen quad $k$,

$$\text{tex.y} := \frac{k}{\lceil\sqrt{2}\,h\rceil}, \quad k = 0, 1, \ldots, \left\lceil\sqrt{2}\,h\right\rceil - 1.$$

Then, the projection line integrals are effectively calculated by "blending" all screen quads together. In terms of graphics programming, this is effected by the ADD blendstate.

### *2.2. Inverse Radon transform*

First, we briefly recall the the theoretical background [4, 7, 5]. In what follows, $\mathcal{F}_n$ denotes the Fourier transform in $n$ dimensions. We treat $\theta$ as fixed parameter for now and set $P_\theta(t) := (Rf)(\theta, t)$. Then, the so-called "Fourier slice theorem" may be derived as follows. Let

$$\begin{aligned} S_\theta(\omega) &:= \sqrt{2\pi}\,\mathcal{F}_1 P_\theta = \int_{-\infty}^{\infty} \int_{\hat{n}_\theta \cdot x = t} f(x)\,\mathrm{d}m(x)\,\mathrm{e}^{-\mathrm{i}\omega t}\,\mathrm{d}t \\ &= \iint_{\mathbb{R}^2} f(x)\,\mathrm{e}^{-\mathrm{i}\omega \hat{n}_\theta \cdot x}\,\mathrm{d}^2x = 2\pi(\mathcal{F}_2 f)(\omega \hat{n}_\theta). \end{aligned} \tag{3}$$

That is, the 1-dimensional Fourier transform of $P_\theta$ yields the 2-dimensional Fourier transform of $f$.

Thus, applying the inverse Fourier transform recovers $f$. Using polar coordinates, we obtain

$$\begin{aligned} f(y) &= \frac{1}{(2\pi)^2} \int_0^{2\pi} \int_0^{\infty} S_\theta(\omega)\,\mathrm{e}^{\mathrm{i}\omega \hat{n}_\theta \cdot y}\,\omega\,\mathrm{d}\omega\,\mathrm{d}\theta \\ &= \frac{1}{(2\pi)^2} \int_0^{\pi} \int_{-\infty}^{\infty} S_\theta(\omega)\,\mathrm{e}^{\mathrm{i}\omega \hat{n}_\theta \cdot y}\,|\omega|\,\mathrm{d}\omega\,\mathrm{d}\theta, \end{aligned}$$

where we have used that $S_{\theta+\pi}(-\omega) = S_\theta(\omega)$. Defining

$$Q_\theta(t) := \frac{1}{2\pi} \int_{-\infty}^{\infty} S_\theta(\omega)\,|\omega|\,\mathrm{e}^{\mathrm{i}\omega t}\,\mathrm{d}\omega, \tag{4}$$

i.e., the 1-dimensional inverse Fourier transform of $S_\theta(\omega)\,|\omega|$ up to a constant prefactor, we thus obtain

$$f(y) = \frac{1}{2\pi} \int_0^{\pi} Q_\theta(\hat{n}_\theta \cdot y)\,\mathrm{d}\theta. \tag{5}$$

This is precisely the backprojection algorithm. Note that for fixed $\theta$, the contribution to the reconstructed image is constant along each line $\hat{n}_\theta \cdot y = \text{const}$.

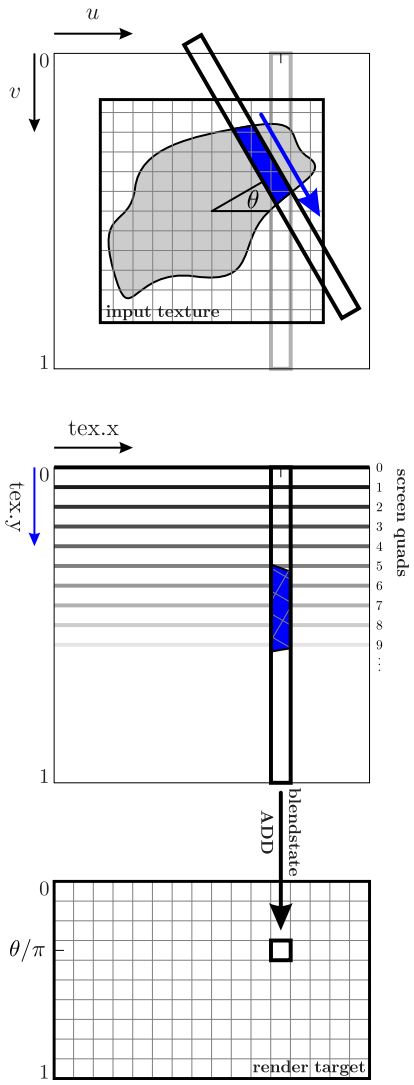


Figure 2: Illustration of the "rotated projections" rendering pass, which yields the forward Radon transform of the input texture using a single draw call. The highlighted rotated bar depicts a projection line along the input texture. The sampling of the texels within the bar (blue) is performed by "screen quads". Summing these texel values up is effectively implemented by the "ADD" blend state. Note that all four vertices forming a screen quad have the same tex.y coordinate. The rotation angle $\theta$ depends on the $y$-position of the current render target pixel.

Directly from the definitions in (3) and (4), we may write $Q_\theta$ concisely as

$$Q_\theta = \mathcal{F}_1^{-1}\left(\left(\mathcal{F}_1 P_\theta\right)(\omega)\,|\omega|\right). \tag{6}$$

In other words, $Q_\theta$ is a frequency domain filtering operation applied to $P_\theta$, with filter $|\omega|$.

The graphics pipeline has to carry out this filtering operation for each row of the output texture in Fig. 2, which serves as input to the inverse Radon transform algorithm. Zero-padding the rows to a power of 2 is required to prevent spatial domain aliasing, and also to speed up the FFT. We will not go into details here since there already exist FFT implementations specifically designed for graphics cards, i.e., the CuFFT [8] library. However, for demonstration purposes we have implemented the Cooley-Tukey FFT algorithm [9] in the provided source code.

In what follows, we show how the backprojection operation in (5) can be mapped to the graphics pipeline, as illustrated in Fig. 3. The input texture has the same structure as the output of the forward transform since the filtering operation (6) does not change dimensions. That is, row $i$ corresponds to rotation angle $\theta_i$ defined in (2). Backprojecting can be visualized as "smearing" the highlighted blue bar in Fig. 3 over the reconstruction plane, which is schematically indicated by the grayscale pixel lines. We employ a collection of $m_\theta$ rotated squares or "screen quads" such that all four vertices forming quad $i$ have the same tex.y texture coordinate,

$$\text{tex.y} := \frac{i}{m_\theta} = \frac{\theta_i}{\pi}, \quad i = 0, 1, \ldots, m_\theta - 1.$$

This quad is rotated by $\theta_i$ in accordance with (5).

## 3. Results

The following table summarizes the running times for a $256 \times 256$ input image and $m_\theta = 180$ rotation angles on a NVIDIA GeForce GTS 250 graphics card with 512 MB internal video memory:

| | |
|---|---|
| forward ($256 \times 256$ pixels) | 3.7 ms |
| inverse ($1024 \times 180$ pixels FFT filtering, $363 \times 180$ pixels backprojection) | 9.7 ms |

The overall system configuration consists of an Intel Core i7 CPU 920 running at 2.67 GHz (8 cores), 6 GB RAM, Windows 7 64-bit version with DirectX 11. For comparison, we have run the same program on the CPU only (DirectX 11 WRAP device) and obtained 112 ms for the forward and 71 ms for the inverse transform. Note that DirectX actually employs all 8 CPU cores simultaneously. Rather surprisingly, the CPU takes longer for the

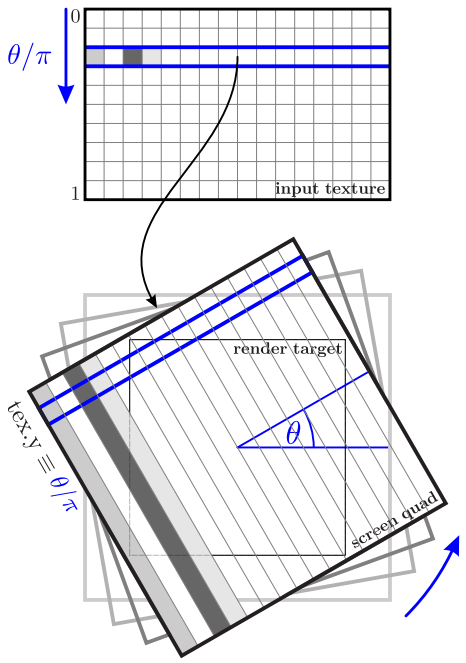


Figure 3: Illustration of the backprojection implementation using rotated screen quads. Each quad corresponds to a fixed angle $\theta$. Sampling the corresponding row in the input texture is effected via the tex.y texture coordinate, which has the same value for all four vertices within a quad.

forward than for the inverse transform. From the results, the GPU speedup factor equals 30 and 7.2, respectively. Note that we have not even employed the latest GeForce graphics card series (GeForce GTX 480 at the time of writing), or one of the NVIDIA Tesla computing processors. These would presumably lead to an even higher speedup factor.

Fig. 4 shows a screenshot of the demonstration program, which computes and visualizes the forward and inverse Radon transform in real-time. The output texture of the forward transform has dimensions $\lceil \sqrt{2}\,h \rceil \times m_\theta = 363 \times 180$ and is shown in the middle (rotated by 90°). As mentioned above, the filtering operation for the inverse transform requires zero-padding. Our implementation extends the width from 363 to 1024; thus, a forward and inverse 1-dimensional FFT of length 1024 has to be carried out 180 times.

## 4. Conclusion

As expected, the GPU implementation leads to a significant performance speedup as compared to modern CPUs, since graphics cards are specifically designed for texture operations like the Radon transform. Our contribution here is a detailed, pedagogical description of how to map the theoretical framework to the graphics pipeline. It is worth mentioning that except for the Fourier transform, it suffices to employ "classical" rendering techniques using vertex and pixel shaders. This provides additional benefits, like, for example, linear texture filtering, which is readily available in hardware.

[1] P. M. Novotny, J. A. Stoll, N. V. Vasilyev, P. J. del Nido, P. E. Dupont, T. E. Zickler, R. D. Howe, GPU based real-time instrument tracking with three-dimensional ultrasound, Medical Image Analysis 11 (5) (2007) 458 – 464. doi:10.1016/j.media.2007.06.009.

[2] F. Xu, K. Müller, Real-Time 3D Computed Tomographic Reconstruction Using Commodity Graphics Hardware, Physics in Medicine and Biology 52 (2007) 3405 – 3419.

[3] N. Neophytou, F. Xu, K. Müller, Hardware acceleration vs. algorithmic acceleration: Can GPU-based processing beat complexity optimization for CT?, SPIE Medical Imaging 6510 (2007) 65105F.

[4] J. Radon, Über die Bestimmung von Funktionen durch ihre Integralwerte längs gewisser Mannigfaltigkeiten, Sächsische Akademie der Wissenschaften, Leipzig 69 (1917) 262 – 277.

[5] A. C. Kak, M. Slaney, Principles of Computerized Tomographic Imaging, IEEE Press, 1988.

[6] C. B. Mendl, Forward and inverse Radon transform demonstration program (October 2010).
URL http://christian.mendl.net/software/iradon.zip

[7] J. Radon, P. Parks, On the Determination of Functions from Their Integral Values along Certain Manifolds, IEEE Transactions on Medical Imaging 5 (1986) 170 – 176.

[8] NVIDIA, CuFFT Library (August 2010).
URL http://developer.nvidia.com/object/gpucomputing.html

[9] J. W. Cooley, J. W. Tukey, An Algorithm for the Machine Calculation of Complex Fourier Series, Mathematics of Computation 19 (1965) 297 – 301.

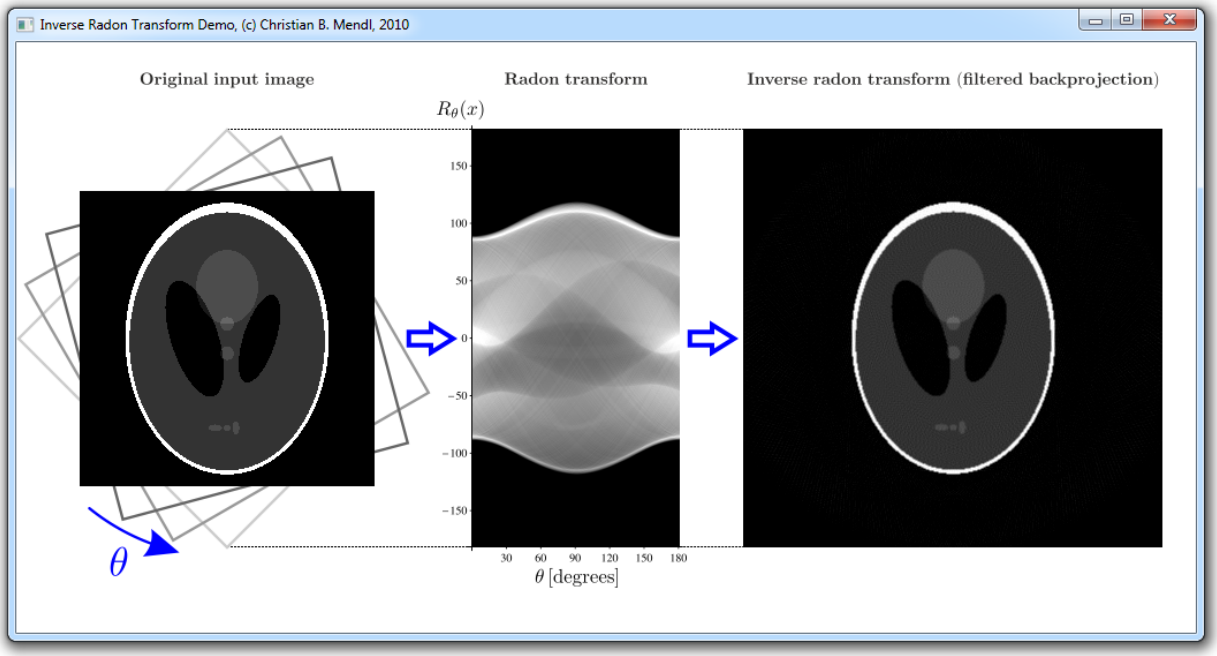


Figure 4: Screenshot of the animated forward and inverse Radon transform demonstration program, showing the Shepp-Logan phantom. Note that all computations are performed in real-time. The image in the middle shows the calculated forward transform and serves as input to the inverse transform. On the right is the reconstructed image after filtering (not shown) and backprojection.