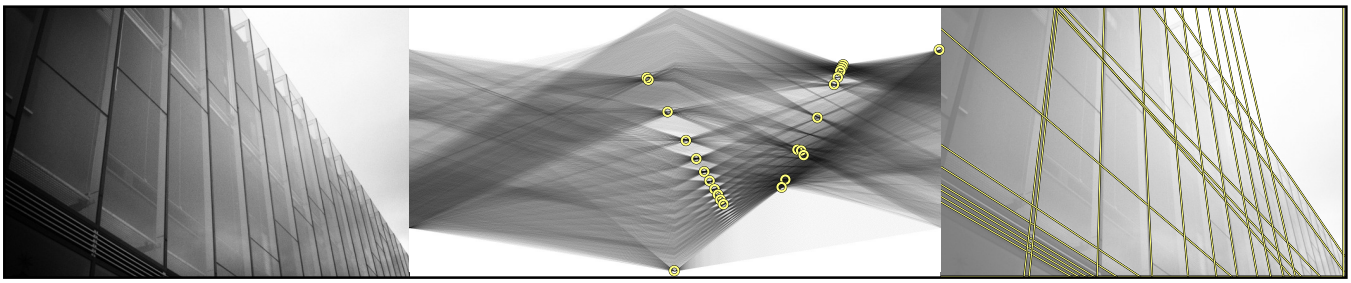
# Real-Time Detection of Lines using Parallel Coordinates and OpenGL

Markéta Dubská*
Graph@FIT
Brno University of Technology

Jiří Havel†
Graph@FIT
Brno University of Technology

Adam Herout‡
Graph@FIT
Brno University of Technology



## Abstract

Line detection in raster images is frequently performed using the Hough Transform. Hough Transform for line detection is difficult to accelerate using the GPU because it essentially requires rasterization of sinusoids into a high-resolution raster of accumulators, which is not a suitable task for GPU. This paper presents a GPU implementation of the PClines – a new parameterization of lines for the Hough Transform. PClines are a point-to-line-mapping and thus the detection of lines uses the graphics processor to rasterize lines into a rectangular frame buffer which is a task very natural and effective on the GPU. The OpenGL 3.3 pipeline is used to efficiently perform the whole of the PClines-based Hough Transform on the GPU. Experimental evaluation shows that even for high-resolution input images with complicated content, the line detector performs easily in real time, which allows for different practical applications.

**CR Categories:** I.4.8 [Image Processing]: Scene Analysis; I.5.0 [Pattern Recognition]: General

**Keywords:** Hough Transform, Parallel Coordinates, OpenGL, Line Detection, PClines

## 1 Introduction

The Hough transform is a well-known tool for detecting shapes and objects in raster images. Originally, Hough [1962] defined the transformation for detecting lines; later it was extended for more complex shapes and arbitrary patterns [Ballard 1987].

*e-mail: idubska@fit.vutbr.cz
†e-mail: ihavel@fit.vutbr.cz
‡e-mail: herout@fit.vutbr.cz

When used for detecting lines in 2D raster images, the Hough transform is defined by a *parameterization* of lines: each line is described by two parameters. The input image is preprocessed and for each pixel which is likely to belong to a line, voting accumulators corresponding to lines which could be coincident with the pixel are increased. Next, the accumulators in the parameter space are searched for local maxima above a given threshold, which correspond to likely lines in the original image.

Hough [1962] parameterized the lines by their *slope* and $y$-axis *intercept*. Using this parameterization, Hough space must be infinite and the same is true for any *point-to-line mapping* (PTLM) where a point in the source image corresponds to a line in the Hough space; and a point in the Hough space represents a line in the $x$-$y$ image space [Bhattacharya et al. 2002]. However, for any PTLM, a complementary PTLM can be found so that the two mappings define two finite Hough spaces containing all lines possible in the $x$-$y$ image space. Some naturally bounded parameterizations exist, such as the very popular $\theta$-$\varrho$ parameterization introduced by Duda and Hart [1972], which is based on the line equation in the normal form. Other bounded parameterizations were introduced by Wallace [1985], Natterer [1986], Eckhardt and Mederlechner [1988], and Forman [1986]. As the line's parameters, intersections with a bounding rectangle or circle were used, together with angles defined by these intersections and the input point.

The majority of currently used implementations seems to be using the $\theta$-$\varrho$ parameterization [Duda and Hart 1972]. In this parameterization, for each input pixel, a sinusoid curve must be rasterized which makes this method very computationally complex. That is why several research groups invested great effort in order to deal with these undesirable properties. Different methods focus on special data structures, non-uniform resolution of the accumulation array or special rules for picking points from the input image. O'Rourke and Sloan [1984] developed two special data structures: *dynamically quantized spaces* (DQS) [O'Rourke 1981] and a *dynamically quantized pyramid* (DQP) [Sloan 1981]. Both these methods use splitting and merging cells of the space represented as a binary tree or possibly a quadtree. A typical method using a special picking rule is the Randomized Hough Transform (RHT) [Xu et al. 1990]. This method is based on the idea that each point in an $n$-dimensional Hough space of parameters can be exactly defined by an $n$-tuple of

points from the input raster image. Instead of an accumulation of a hypersurface in the Hough space for each point, $n$ points are randomly picked and one corresponding accumulator in the parameter space is increased. Approaches based on repartitioning the Hough space can be represented by the Fast Hough Transform (FHT) [Li et al. 1986]. The algorithm assumes that each edge point in the input image defines a hyperplane in the parameter space. These hyperplanes recursively divide the space into hypercubes and perform the Hough transform only on the hypercubes with votes exceeding a selected threshold.

The Hough transform also provides space for parallel implementations using special hardware, such as a distributed memory multiprocessor [Underhill et al. 1999], graphics hardware [Strzodka et al. 2003], pyramid multiprocessors [Atiquzzaman 1994], or reconfigurable architectures [Pavel and Akl 1996]. For more information about different existing modifications and implementations of the Hough transform, one can see a comprehensive overview by Illingworth and Kittler [1988].

Line detection is an integral part of many image processing tasks such as camera calibration, object detection or marker localization (bar codes, QR codes, augmented reality markers), and many more. Because some of these tasks are performed online, fast line detection is not only desirable but sometimes necessary. This paper is about real-time detection of lines based on a parameterization using parallel coordinates (PC) and implemented on GPU using OpenGL shading language. Other GPU implementations of the Hough Transform exist, however the PClines parametrization is perfectly suitable for OpenGL implementation because it requires only line rasterization. The goal of the presented research is to maximally utilize contemporary graphics chips for the task of detecting straight lines in raster images.

Basic information on the parallel coordinates and the "PClines" line parameterization is reviewed in Section 2. The real-time algorithm for line-detection designed for today's graphics chips is presented in Section 3. Section 4 contains an experimental evaluation. Conclusions and ideas for further development are given in Section 5.

## 2 PClines: Line Detection using Parallel Coordinates

Parallel coordinates (PC) were invented in 1885 by Maurice d'Ocagne [d'Ocagne 1885] and further studied and popularized by Alfred Inselberg [2009]. The coordinate system used for representing geometric primitives in parallel coordinates is defined by mutually parallel axes. Each $N$-dimensional vector is represented by $(N-1)$ lines connecting the axes – see Fig. 1. In this text, we will be using an Euclidean plane with a $u$-$v$ Cartesian coordinate system to define positions of points in the space of parallel coordinates. For defining these points, a notation $(u, v, w)_{\mathbb{P}^2}$ will be used for homogeneous coordinates in the projective space $\mathbb{P}^2$ and $(u, v)_{\mathbb{E}^2}$ will be used for Cartesian coordinates in the Euclidean space $\mathbb{E}^2$.

In the two-dimensional case, points in the $x$-$y$ space are represented as lines in the space of parallel coordinates. Representations of collinear points intersect at one point – the representation of a line (see Fig. 2). Based on this relationship, it is possible to define a point-to-line mapping between

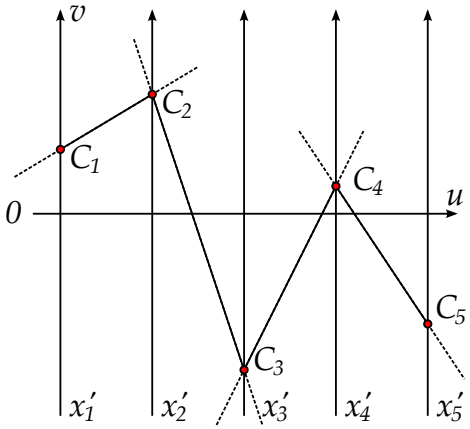



Figure 1: Representation of a 5-dimensional vector in parallel coordinates. The vector is represented by its coordinates $C_1, \ldots, C_5$ on axes $x'_1, \ldots, x'_5$, connected by a complete polyline (composed of 4 infinite lines).

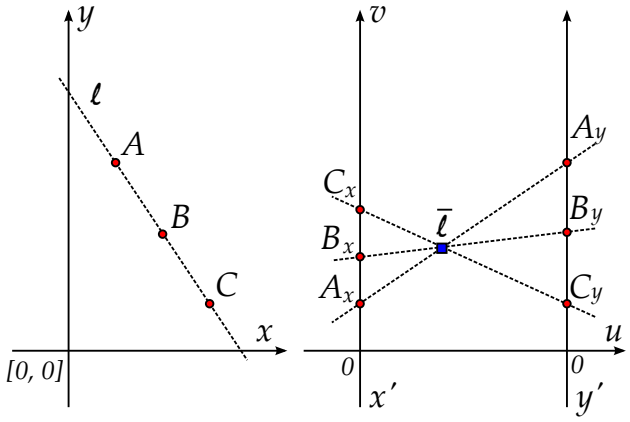



Figure 2: Three collinear points in parallel coordinates: (left) Cartesian space and (right) space of parallel coordinates. Line $\ell$ is represented by point $\overline{\ell}$ in parallel coordinates.

the original $x$-$y$ space and the space of parallel coordinates. For some cases, such as line $\ell : y = x$, the corresponding point $\overline{\ell}$ lies in infinity (it is an ideal point). Projective space $\mathbb{P}^2$ (contrary to the Euclidean $\mathbb{E}^2$ space) provides coordinates for these special cases. A relationship between line $\ell : ax + by + c = 0$ (denoted as $[a, b, c]$) and its representing point $\overline{\ell}$ can be defined by mapping:

$$\ell : [a, b, c] \rightarrow \overline{\ell} : (db, -c, a + b)_{\mathbb{P}^2}, \qquad (1)$$

where $d$ is the distance between parallel axes $x'$ and $y'$.

### 2.1 Parameterization "PClines" for Line Detection

This section gives a brief overview of the "PClines" parameterization introduced in [Dubská et al. 2011]. In the following text, we will use the intuitive slope-intercept line equation $y = mx + b$. Using this parameterization, the corresponding point $\overline{\ell}$ in the parallel space has coordinates $(d, b, 1 - m)_{\mathbb{P}^2}$. The line's representation $\overline{\ell}$ is between the

axes $x'$ and $y'$ if and only if $-\infty < m < 0$. For $m = 1$, $\overline{\ell}$ is an ideal point (a point in infinity). For $m = 0$, $\overline{\ell}$ lies on the $y'$ axis, for vertical lines ($m = \pm\infty$), $\overline{\ell}$ lies on the $x'$ axis.

Besides this space defined by parallel axes $x', y'$ (further referred to as *straight*, $\mathcal{S}$), we propose using a *twisted* ($\mathcal{T}$) system $x', -y'$, which is identical to the straight space, except that the $y'$ axis is inverted. In the twisted space, $\overline{\ell}$ is between the axes $x'$ and $-y'$ if and only if $0 < m < \infty$. By combining the *straight* and *twisted* spaces, the whole $\mathcal{TS}$ plane can be constructed, as shown in Fig. 3. Figure 3 (top)

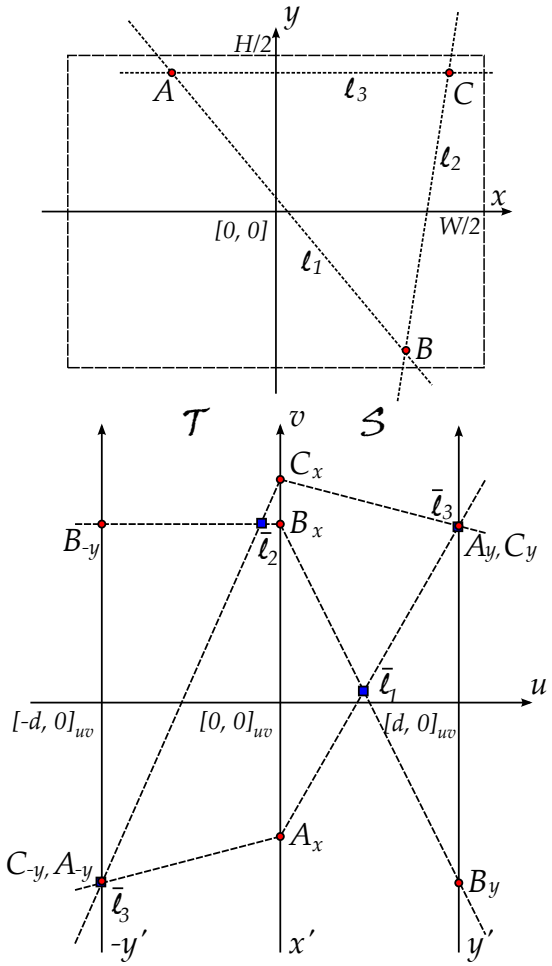



Figure 3: (top) Original $x$-$y$ space and (bottom) its PClines representation – the corresponding $\mathcal{TS}$ space.

shows the original $x$-$y$ image with three points $A$, $B$, and $C$ and three lines $\ell_1$, $\ell_2$, and $\ell_3$ coincident with the points. The origin of $x$-$y$ is placed into the middle of the image for the convenience of the figures. Figure 3 (bottom) depicts the corresponding $\mathcal{TS}$ space. It should be noted that a finite part of the $u$-$v$ plane is sufficient:

$$\begin{array}{rcl} -d & \leq u \leq & d, \\ -\max(\frac{W}{2}, \frac{H}{2}) & \leq v \leq & \max(\frac{W}{2}, \frac{H}{2}), \end{array} \tag{2}$$

where $W$ and $H$ are the width and height of the input raster image, respectively.

Any line $\ell : y = mx + b$ is now represented either by point $\overline{\ell}_{\mathcal{S}}$ in the *straight* half or by $\overline{\ell}_{\mathcal{T}}$ in the *twisted* part of the $u$-$v$ plane:

$$\begin{array}{l} \overline{\ell}_{\mathcal{S}} = \big(d, b, 1-m\big)_{\mathbb{P}^2}, -\infty \leq m \leq 0, \\ \overline{\ell}_{\mathcal{T}} = \big(-d, -b, 1+m\big)_{\mathbb{P}^2}, 0 \leq m \leq \infty. \end{array} \tag{3}$$

Consequently, any line $\ell$ has exactly one image $\overline{\ell}$ in the $\mathcal{TS}$ space; except for cases that $m = 0$ and $m = \pm\infty$, when $\overline{\ell}$ lies in both spaces either on axis $y'$ or $x'$. That allows the $\mathcal{T}$ and $\mathcal{S}$ spaces to be "attached" to one another. Figure 3 illustrates the spaces attached along the $x'$ axis. Attaching also the $y'$ and $-y'$ axes results in an enclosed Möbius strip.

This parameterization can be used for detecting lines using the standard Hough transform procedure, as depicted in Algorithm 1. The space $S(u, v)$ is discretized directly according

**Algorithm 1** Detection of lines using parallel coordinates.

**Input:** Input image $I$ with dimensions $W$, $H$
**Output:** Detected lines $L = \{(m_1, b_1), \ldots\}$
1: $S(u, v) \leftarrow 0, \forall u \in \{-d, \ldots, d\}, v \in \{v_{min}, \ldots, v_{max}\}$
2: **for all** $x \in \{1, \ldots, W\}, y \in \{1, \ldots, H\}$ **do**
3: **if** $I(x, y)$ **is an edge then**
4: **rasterize line in the $\mathcal{S}$ space**
5: **rasterize line in the $\mathcal{T}$ space**
6: **end if**
7: **end for**
8: $L \leftarrow \{\}$
9: $L = \big\{(m(u), b(u, v)) | u \in \{-d, \ldots d\} \wedge$
$v \in \{v_{min}, \ldots, v_{max}\} \wedge S(u, v)$ is a high local max.$\big\}$

to Eq. (2); other discretizations – denser or sparser – would be possible by just linearly mapping the $u$ and $v$ coordinates used in the algorithm. The condition used in codeline 3 is application-specific and it typically involves an edge detection operator and thresholding. The lines rasterized in codeline 4 and 5, in fact, constitute a two-segment polyline defined by three points: $(-d, -y) - (0, x) - (d, y)$. Codeline 9 scans the space of accumulators $S$ for local maxima above a given threshold – this is a standard Hough transform step.

## 3 Real-Time Line Detection using OpenGL

In the presented GLSL implementation, the following shader programs are used:

**Image preprocessing** program in case the input image requires preprocessing, namely conversion to greyscale. This program is optional.

**Accumulation** program for accumulating the edges' votes from the input image to the $\mathcal{TS}$ space.

**Detection** program for detecting local maxima in the $\mathcal{TS}$ space.

Both the image preprocessing and the $\mathcal{TS}$ space accumulation programs are implemented via rendering to a texture. The (optional) preprocessing step is done by simple screen quad rendering.

Most of the $\mathcal{TS}$ space accumulation is done by a geometry shader. A point for every pixel of the input image is rendered by geometry instancing. Builtin variables `gl_VertexID` and `gl_InstanceID` specify the point coordinates. The geometry shader reads the input image at the specified coordinates

and thresholds the value to determine whether it is an edge. The output of the geometry shader is a three-point line strip that is rasterized to the $\mathcal{TS}$ space. The $u$ coordinates of the points are fixed $\{-1, 0, 1\}$ and the $v$ coordinates are based on the input point coordinates $(x, y)$ (see Section 2.1). The $\mathcal{TS}$ space is accumulated using additive blending into a floating-point texture.

The maxima detection is also performed by a geometry shader. A point is rendered for each pixel of the $\mathcal{TS}$ space (stored in the texture) and the geometry shader checks the small neighbourhood of this pixel to see whether it is a local maximum. In that case, the detected line is returned by the *transform feedback*. The maxima detection could be implemented separably using two passes and one temporary texture. However, experiments have shown that the single pass detection performed faster for all used neighbourhood sizes.

### 3.1 Harnessing the Edge Orientation

O'Gorman and Clowes [1976] improve the basic $\theta$-$\varrho$ parameterization with their idea of not accumulating values for all discretized values of $\theta$ but for a single value of $\theta$, instead. The appropriate $\theta$ for a point can be obtained from the gradient of the detected edge present at this point [Shapiro and Stockman 2001].

One common way to calculate the local gradient direction of the image intensity is by using the Sobel operator. Sobel kernels for convolution are as follows: $S_x = [1, 2, 1]^T \cdot [1, 0, -1]$ and $S_y = [1, 0, -1]^T \cdot [1, 2, 1]$. Using these convolution kernels, two gradient values $G_x$ and $G_y$ can be obtained for any discrete location in the input image. Based on these, the gradient's direction is $\theta = \arctan(G_y/G_x)$. The line's inclination in the slope-intercept parameterization $m$-$b$ is related to $\theta$:

$$m = -\tan\frac{1}{\theta}. \tag{4}$$

The slope $m$ of line $\ell$ defines the $u$ coordinate of the line's image $\overline{\ell}$ in the $\mathcal{TS}$ space: $u = d/(1-m)$ for $\mathcal{S}$ space and $u = -d/(1+m)$ for $\mathcal{T}$ space. When $\overline{\ell}$ is in the $\mathcal{S}$ space,

$$u_{\mathcal{S}} = d\frac{1}{1-m} = d\frac{1}{1+\tan\theta^{-1}} = d\frac{G_y}{G_y+G_x} \tag{5}$$

and similarly in the $\mathcal{T}$ space

$$u_{\mathcal{T}} = d\frac{G_y}{-G_y+G_x}. \tag{6}$$

The $u$ coordinate can be expressed independently of the location of $\overline{\ell}$ as

$$u = d\frac{G_y}{(\operatorname{sgn} G_y)(\operatorname{sgn} G_x)G_y + G_x}. \tag{7}$$

It should be noted that contrary to the "standard" $\theta$-$\varrho$ parameterization, no goniometric operation is needed to compute the horizontal position of the ideal gradient in the accumulator space. In order to avoid errors caused by noise and the discrete nature of the input image, accumulators within a suitable interval $\langle u-r, u+r\rangle$ around the calculated angle (or more precisely $u$ position) are also incremented. That – unfortunately – introduces a new parameter of the method – radius $r$. However, experiments show that neither the robustness nor the speed is affected notably by the selection of $r$.

The algorithm of line detection taking into account the edge orientation is depicted by Alg. 2. Although the $\mathcal{TS}$ space is a plane, three dimensional space is involved. The third coordinate is used in one special case illustrated by Figure 4. It is the situation which occurs if the rendered part of the polyline around the estimated $u$ is outside of interval $[-d, d]$. Such a situation results in the necessity of rendering two separate lines. Instead of calculating all four endpoints of the lines, only three vertices are emitted with different $z$-coordinates and the back clipping plane of the OpenGL view frustum is used to clip the polyline.

**Algorithm 2** Geometry Shader Using Edge Orientation

**Input:** Image $I$ with dimensions $W, H$, radius $r$ and $d$=1
**Output:** Accumulator space $S$ (refer to Alg. 1)

**for all** $x \in \{1, \ldots, W\}, y \in \{1, \ldots, H\}$ **do**
  $G_x = (I * S_x)(x, y)$
  $G_y = (I * S_y)(x, y)$
  $u = \dfrac{G_y}{\operatorname{sgn}(G_x)\operatorname{sgn}(G_y)G_y + G_x}$
  **if** $G_x^2 + G_y^2 > \tau$ **then**
    $u_L = u + r$, $u_R = u - r$
    **if** $u_L <= 0 \wedge u_R <= 0$ **then**
      **emit vertex** $(u_L, (y+x) * u_L + x, 0)$
      **emit vertex** $(u_R, (y+x) * u_R + x, 0)$
    **else if** $u_L >= 0 \wedge u_R >= 0$ **then**
      **emit vertex** $(u_L, (y-x) * u_L + x, 0)$
      **emit vertex** $(u_R, (y-x) * u_R + x, 0)$
    **else if** $u_L < -1 \vee u_R > 1$ **then**
      **if** $u_L < -1$ **then**
        $u_L = u_L + 2$
      **end if**
      **if** $u_R > 1$ **then**
        $u_R = u_R - 2$
      **end if**
      $z = (1-r)/r$
      **emit vertex** $(-1, -y, (z(1+u_R)-1)/u_R)$
      **emit vertex** $(0, x, z)$
      **emit vertex** $(1, y, (1-z(1-u_L))/u_L)$
    **else**
      **emit vertex** $(u_L, (y+x) * u_L + x, 0)$
      **emit vertex** $(0, x, 0)$
      **emit vertex** $(u_R, (y-x) * u_R + x, 0)$
    **end if**
  **end if**
**end for**

## 4 Experimental Results

This section presents the experimental evaluation of the proposed algorithm. Though the aim of this paper is mainly the GPU speed-up, it is important to mention the accuracy of PClines detection – referred to in Section 4.1. Section 4.2 contains the results achieved by the GLSL implementation of PClines presented in this paper.

The following hardware was used for testing (in bold face is the identifier used in the text):

**GTX 480** – NVIDIA GTX 480 in a computer with Intel Core i7-920, 6GB 3×DDR3-1066(533MHz) RAM;

**GTX 280** – NVIDIA GTX 280 in a computer with Intel Core i7-920, 6GB 3×DDR3-1066(533MHz) RAM;

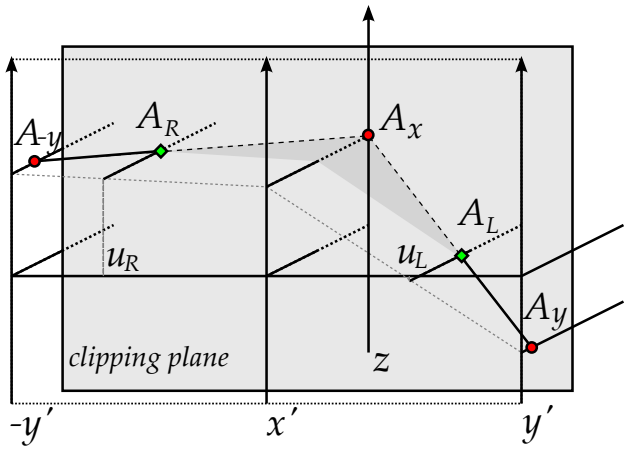



Figure 4: Three vertices used for rendering two separate line segments. The middle point $A_x$ has its $z$-coordinate calculated as $(1-r)/r$, where $r$ is the radius rendered around the predicted $u$ (this restricts $r$ to be smaller than $d$). The depths of vertices $A_{-y}$ and $A_y$ are calculated according to the required lengths of the line segments.

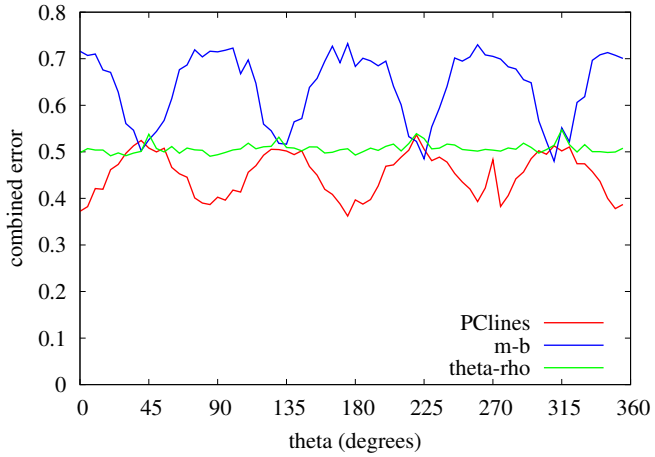



Figure 5: Line localization error as it depends on the lines' slope. For $x$ on the horizontal scale, the lines' slope in degrees is at interval $\langle x, x+5)$. Red: PClines; Green: $\theta$-$\varrho$; Blue: $m$-$b$. Average error of the 5 least accurate lines, (i.e. a pessimistic error estimation).

**GT 130M** – NVIDIA GT 130M mobile GPU in a laptop computer with Intel Core 2 DUO T6500, 2× 2GB DDR2 399MHz RAM;

**HD 5970-1** – AMD Radeon HD5970 (single core used) in a computer with Intel Core i5-660, 4GB 3×DDR3-1066(533MHz) RAM;

**HD 5970-2** – AMD Radeon HD5970 (both cores used) in a computer with Intel Core i5-660, 4GB 3×DDR3-1066(533MHz) RAM; and

**i7-920** – Intel Core i7-920, 6GB 3×DDR3-1066(533MHz) RAM – the same computer is used for testing the GTX 480 and GTX 280.

### 4.1 Accuracy Evaluation

The line localization error was measured on automatically generated data and calculated as the Euclidean distance from the ground truth. For comparison, Hough transform using $\theta$-$\rho$ and $m$-$b$ (slope-intercept) parametrizations was used. Figure 5 shows the dependency of the detection error on the line's slope for all the three parameterizations. The test was performed on images 512 × 512 pixels and lines generated with random $\theta$ in 5° intervals. The measurements show that the $\theta$-$\varrho$ parameterization discretizes the space evenly; PClines are about as accurate as $\theta$-$\varrho$ for $\theta \in \{45°, 135°, 225°, 315°\}$ and more accurate at $\theta \in \{0°, 90°, 180°, 270°\}$; the $m$-$b$ parameterization is the least accurate of the three evaluated methods. One should refer to [Dubská et al. 2011] for more details.

### 4.2 Performance Evaluation on Real-Life Images

As the dataset for this test real photographs with different amounts of edge points and different dimensions were used – see Figure 6. The images are sorted according to the number of edge points detected by the Sobel filter.

The presented algorithm (further referred to as **PClines**) was compared to a software implementation of the "standard" $\theta$-$\varrho$ based Hough transform taken from the **OpenCV** library[1] and parallelized by OpenMP and slightly optimized.

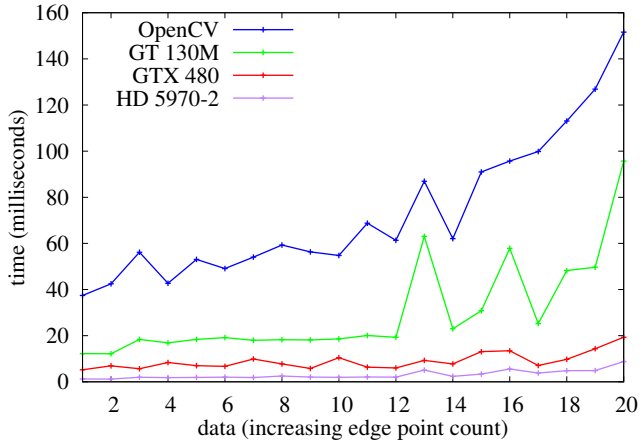



Figure 7: Performance evaluation of computational complexity tested on real-world images. The GLSL implementation is compared to a parallelized OpenCV implementation (using all cores of the i7-920). Figure 6 shows the individual images (horizontal axis of this graph).

The results are reported in Figure 7. The measurements verify that the computational complexity depends mostly on the number of edge points extracted from the input image and the edge-detection phase is linearly proportional to the image resolution, which causes nonlinearity in the graph. The GPU-accelerated implementations are notably faster than the software implementation. A detailed comparison of the GPU-accelerated implementations is shown in Figure 8.

[1] http://opencv.willowgarage.com


Figure 6: Images used in the test. The number in the top-left corner of each thumbnail image is the image ID – used on the horizontal axis in Figure 7 and 8. The bottom-left corner of each thumbnail image states the number of edge points and pixel resolution of the tested image.

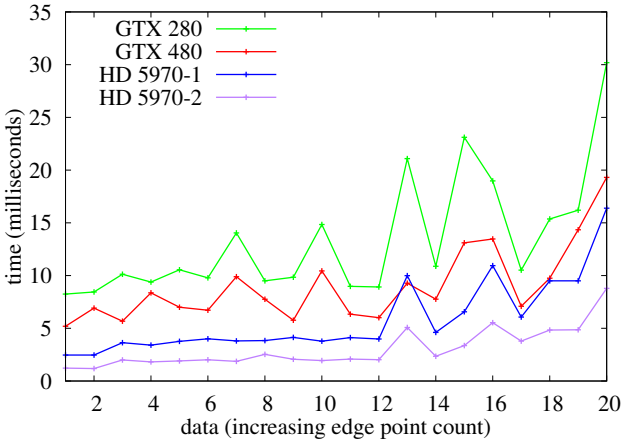



Figure 8: Performance evaluation of the GLSL implementation using different high-end graphics hardware.

## 5 Conclusion

This paper presents an OpenGL 3.3 implementation of the PClines line detector. Contrary to the "standard" $\theta$-$\varrho$ parameterization which requires rasterization of sinusoids into the Hough accumulator space, PClines are a point-to-line-mapping. That allows for a very efficient use of the GPU for accumulation of the votes in the Hough space.

The results show that PClines can be used for real-time detection of lines, even in complex images of high resolutions. The accuracy of the parameterization notably outperforms the original Hough's slope-intercept parameterization and it is equal or more accurate than the commonly used $\theta$-$\varrho$ parameterization. Together with its ability to directly detect sets of parallel or mutually coincident lines, PClines seem very attractive for use in various applications. One advantage of the presented solution is its ability to avoid using of CUDA or OpenCL which are still facing compatibility issues.

The simplicity of the algorithm is not only suitable for implementation in OpenGL which is presented here. In the near future, we are considering experiments on programmable hardware (FPGA). Another interesting topic of future study can be porting the algorithm presented here to older versions of the OpenGL pipeline. Such ports can be welcome in contemporary smartphones and other mobile and ultramobile devices supporting OpenGL ES.

## Acknowledgements

This research was supported by the EU FP7-ARTEMIS project no. 100230 SMECY, by the research project CEZMSMT, MSM0021630528, and by the BUT FIT grant FIT-10-S-2.

## References

Atiquzzaman, M. 1994. Pipelined implementation of the multiresolution Hough transform in a pyramid multiprocessor. *Pattern Recognition Letters 15*, 9, 841 – 851.

Ballard, D. H. 1987. Generalizing the Hough transform to detect arbitrary shapes. 714–725.

Bhattacharya, P., Rosenfeld, A., and Weiss, I. 2002. Point-to-line mappings as Hough transforms. *Pattern Recognition Letters 23*, 14, 1705–1710.

d'Ocagne, M. 1885. *Coordonnées parallèles et axiales. Méthode de transformation géométrique et procédé nouveau de calcul graphique déduits de la considération des coordonnées parallèlles.* Gauthier-Villars.

Dubská, M., Herout, A., and Havel, J. 2011. PClines – line detection using parallel coordinates. In *Proceedings of CVPR 2011*.

Duda, R. O., and Hart, P. E. 1972. Use of the Hough transformation to detect lines and curves in pictures. *Commun. ACM 15*, 1, 11–15.

Eckhardt, U., and Maderlechner, G. 1988. Application of the projected Hough transform in picture processing. In *Proceedings of the 4th International Conference on Pattern Recognition*, Springer-Verlag, London, UK, 370–379.

Forman, A. V. 1986. A modified Hough transform for detecting lines in digital imagery. In *Applications of artificial intelligence III*, 151–160.

Hough, P. V. C., 1962. Method and means for recognizing complex patterns, Dec. U.S. Patent 3,069,654.

Illingworth, J., and Kittler, J. 1988. A survey of the Hough transform. *Comput. Vision Graph. Image Process. 44*, 1, 87–116.

Inselberg, A. 2009. *Parallel Coordinates; Visual Multidimensional Geometry and Its Applications*. Springer. ISBN: 978-0-387-21507-5.

Li, H., Lavin, M. A., and Le Master, R. J. 1986. Fast Hough transform: A hierarchical approach. *Comput. Vision Graph. Image Process. 36* (November), 139–161.

Natterer, F. 1986. *The mathematics of computerized tomography*. Wiley, John & Sons, Incorporated. ISBN 9780471909590.

O'Gorman, F., and Clowes, M. B. 1976. Finding picture edges through collinearity of feature points. *IEEE Trans. Computers 25*, 4, 449–456.

O'Rourke, J., and Sloan, K. R. 1984. Dynamic quantization: Two adaptive data structures for multidimensional spaces. *Pattern Analysis and Machine Intelligence, IEEE Transactions on PAMI-6*, 3 (May), 266 –280.

O'Rourke, J. 1981. Dynamically quantized spaces for focusing the Hough transform. In *Proceedings of the 7th international joint conference on Artificial intelligence - Vol. 2*, Morgan Kaufmann Publ. Inc., San Francisco, CA, USA, 737–739.

Pavel, S., and Akl, S. 1996. Efficient algorithms for the Hough transform on arrays with reconfigurable optical buses. In *Parallel Processing Symposium, 1996., Proceedings of IPPS '96, The 10th International*, 697 –701.

Shapiro, L. G., and Stockman, G. C. 2001. *Computer Vision*. Tom Robbins.

Sloan, K. R. 1981. Dynamically quantized pyramids. In *In Proc. International Joint Conference on Artificial Intelligence (IJCAI*, Kaufmann, 734–736.

Strzodka, R., Ihrke, I., and Magnor, M. 2003. A graphics hardware implementation of the Generalized Hough Transform for fast object recognition, scale, and 3D pose detection. In *Proceedings of IEEE International Conference on Image Analysis and Processing (ICIAP'03)*, 188–193.

Underhill, A., Atiquzzaman, M., and Ophel, J. 1999. Performance of the Hough transform on a distributed memory multiprocessor. *Microprocessors and Microsystems 22*, 7, 355 – 362.

Wallace, R. 1985. A modified Hough transform for lines. In *Proceedings of CVPR 1985*, 665–667.

Xu, L., Oja, E., and Kultanen, P. 1990. A new curve detection method: Randomized Hough Transform (RHT). *Pattern Recognition Letters 11* (May), 331–338.