ICONS 2012, Saint Gilles, Reunion Island, IARIA, ISBN: 978-1-61208-184-7, pp. 193-198, 2012

Interpolation and Intersection Algorithms and GPU

Vaclav Skala
Department of Computer Science
VSB-Technical University
Ostrava, Czech Republic
Vaclav.Skala@vsb.cz

Abstract — Interpolation and intersection methods are closely related and used in computer graphics, visualization, computer vision etc. The Euclidean representation is used nearly exclusively not only in computational methods, but also in education despite it might lead to instability in computation in many cases. The projective geometry, resp. projective extension of the Euclidean space, offers many positive features from the computational and educational points of view with higher robustness and stability of computation. This paper presents simple examples of projective representation advantages, especially from the educational point of view. In particular, how interpolation and intersection can be applied to fundamental algorithms, which are becoming more robust, stable and faster due to compact formulation. Another advantage of the proposed approach is a simple implementation on vector-vector architectures, e.g. GPU, as it is based on matrix-vector operations.

Keywords - Interpolation; intersection; principle of duality; barycentric coordinates; cross-product; linear systems of equations.

I. INTRODUCTION

Algorithm efficiency and robustness are key points of research activities in computer graphics [7], [1], computer vision [6], [4], texture mapping [19] etc. Due to many items to be processed, a strong requirement for speed arises; also hardware architecture needs to be considered. However, speed and robustness requirements are usually in contradiction, especially if the Euclidean representation is used.

Nevertheless, some other approaches like projective or conformal geometries can be used to overcome selected problems. As the projective representation is widely used in computer graphics, a simple modification of interpolation and intersection algorithms will be introduced and simple examples presented for demonstration.

II. PROJECTIVE REPRESENTATION

Projective representation uses homogeneous coordinates for computations and geometric transformations. A point $\boldsymbol{X} = (X, Y)$ in E^2 (the Euclidean space) can be represented as $\boldsymbol{x} = [x, y: w]^T$ in P^2 (the projective extension of the Euclidean space). Mutual conversion is defined as:

$$X = {}^{x}/_{w} \quad Y = {}^{y}/_{w} \quad w \neq 0 \quad \text{(origin excluded)}$$

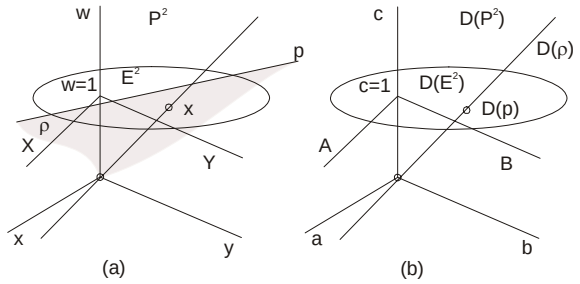

Figure 1. Geometric interpretation of a dual space

One parametric set of points in P^2 representing a unique point in E^2, see [14]. A significant advantage of the projective representation is a possibility to use principle of duality. In the above case, a point is dual to a line and vice versa, etc. which might lead to new algorithms [2], [13], [18].

Similarly, the concept of the projective extension of the Euclidean space can be extended to n-dimensional space, especially to E^3 used in computer graphics and vision.

This simple formulation shows, that many computations, not necessarily only geometric transformations, can be made using homogeneous coordinates.

It means that "*projective scalar*", i.e. $\boldsymbol{x} = [\bar{x}: w]^T$ or "*projective vector*" $\boldsymbol{x} = [\bar{\boldsymbol{x}}: w]^T = [x, y: w]^T$, where $\bar{\boldsymbol{x}}$ is a vector, can form an input or output of the processing pipeline, e.g. interpolation and intersection computation.

Projective representation can also help to explain many geometrical problems in a simple way, e.g. line intersection [13], area or volume computation [14], solution of linear homogeneous systems and computation of barycentric coordinates [12].

III. PRINCIPLE OF DUALITY

Principle of duality is an essential principle and especially in computer graphics and vision can bring quite a new way how to handle and solve non-trivial problems. The principle states that any theorem remains true when we interchange the words "point" and "line", "lie on" and "pass through", "join" and "intersection" and so on. Once the theorem has been established, the dual theorem is obtained as described above, see [5].

The advantages of the projective geometry and principle of duality use can be demonstrated on very simple examples, e.g. a point as an intersection of two lines and a line as a join of two points.

Let two points $\mathbf{x}_1$ and $\mathbf{x}_2$ be given in the projective space. Then the coefficients of the line $\mathbf{p}$, which is defined by those two points, are determined as the of their homogeneous coordinates.

$$\mathbf{p} = \mathbf{x}_1 \times \mathbf{x}_2 \qquad \text{i.e.} \qquad \mathbf{p} = \det\begin{bmatrix} \mathbf{i} & \mathbf{j} & \mathbf{k} \\ x_1 & y_1 & w_1 \\ x_2 & y_2 & w_2 \end{bmatrix}$$

where: $\mathbf{p} = [a,b:c]^T$

If the principle of duality is used, it is possible to write computation of an intersection of two lines as:

$$\mathbf{x} = \mathbf{p}_1 \times \mathbf{p}_2 \qquad \text{i.e.} \qquad \mathbf{x} = \mathbf{p}_1 \times \mathbf{p}_2 = \det\begin{bmatrix} \mathbf{i} & \mathbf{j} & \mathbf{k} \\ a_1 & b_1 & c_1 \\ a_2 & b_2 & c_2 \end{bmatrix}$$

where: $\mathbf{x} = [x,y:w]^T$

A computation of an intersection point of two lines or a computation of a line if two points are given is made by the same sequence using the principle of duality and no division operation is needed.

In the case of E^3 point is dual to a plane and vice versa, i.e. if three points are given a computation of a plane is the same, in the sense of duality, as intersection computation of three planes, i.e. a plane is computed by the generalized cross-product as:

$$\mathbf{x}_1 \times \mathbf{x}_2 \times \mathbf{x}_3 = \det\begin{bmatrix} \mathbf{i} & \mathbf{j} & \mathbf{k} & \mathbf{l} \\ x_1 & y_1 & z_1 & w_1 \\ x_2 & y_2 & z_2 & w_2 \\ x_3 & y_3 & z_3 & w_3 \end{bmatrix}$$

and an intersection of three planes is computed as:

$$\boldsymbol{\rho}_1 \times \boldsymbol{\rho}_2 \times \boldsymbol{\rho}_3 = \det\begin{bmatrix} \mathbf{i} & \mathbf{j} & \mathbf{k} & \mathbf{l} \\ a_1 & b_1 & c_1 & d_1 \\ a_2 & b_2 & c_2 & d_2 \\ a_3 & b_3 & c_3 & d_3 \end{bmatrix}$$

It should noted that a division operation is not needed and the computational sequence is the same for both the cases.

IV. LINEAR INTERPOLATION

Linear interpolation is frequently used method not only in computer graphics. Let us consider a simple case of linear interpolation, when we want to interpolate on a line $\boldsymbol{p}$ or on a surface $\boldsymbol{\rho}$, i.e.

$$\boldsymbol{p}: \boldsymbol{X}(t) = \boldsymbol{X}_A + \boldsymbol{S}_1 t \quad \text{or}$$

$$\boldsymbol{\rho}: \boldsymbol{X}(u,v) = \boldsymbol{X}_A + \boldsymbol{S}_1 u + \boldsymbol{S}_2 v$$

where: $\boldsymbol{S}_1 = \boldsymbol{X}_B - \boldsymbol{X}_A$ and $\boldsymbol{S}_2 = \boldsymbol{X}_C - \boldsymbol{X}_A$

These are well-known formulas, of course. But what happens if points are given in homogeneous coordinates?

From the teaching experience, the approach is a conversion of points to the Euclidean space followed by the "standard" linear interpolation. It means that in the first case 6 divisions and in the second case 9 divisions are needed with all consequences, including precision and stability issues.

However, there is a possibility to make a linear interpolation **directly** in homogeneous coordinates as:

$$\boldsymbol{p}: \boldsymbol{x}(t) = \boldsymbol{x}_A + \boldsymbol{s}_1 t$$

where: $\boldsymbol{s_1} = \boldsymbol{x}_B - \boldsymbol{x}_A$

$$= [x_B - x_A,\ y_B - y_A : w_B - w_A]^T$$

or

$$\boldsymbol{\rho}: \boldsymbol{x}(\xi,\eta) = \boldsymbol{x}_A + \boldsymbol{s}_1\xi + \boldsymbol{s}_2\eta$$

where: $\boldsymbol{s_1} = \boldsymbol{x}_B - \boldsymbol{x}_A$

$$= [x_B - x_A,\ y_B - y_A, z_B - z_A : w_B - w_A]^T$$

and

$$\boldsymbol{s_2} = \boldsymbol{x}_C - \boldsymbol{x}_A = [x_C - x_A,\ y_C - y_A,\ z_C - z_A : w_C - w_A]^T$$

In both cases, the following conditions apply:

$$w_A > 0, \qquad w_B > 0, \qquad w_C > 0$$

As in the projective space the metric is not generally defined, there must be some different behavior of such interpolation. Note that there is a direct connection to interpolation and projection operation in the graphical pipeline.

Basic property of the interpolation in the projective space is a **non-linear monotonic parameterization**, i.e. for $\tau = 1/2$ the center of the segment $\boldsymbol{X}_A\boldsymbol{X}_B$ in the Euclidean space is not obtained in general. It is well known problem of determining z-coordinate after projection operation. It means that we have a linear interpolation with:

- Linear parameterization in the Euclidean space
- Non-linear parameterization, but with a **monotonic** parameterization, in the projective space. This fundamental property is needed when comparison of $t_1 < t_2$, resp. $\tau_1 < \tau_2$, is required for a decision, e.g. which object is closer etc.

In both cases, division operation can be avoided by "hiding" denominator to the homogeneous coordinate, i.e.

$$\boldsymbol{x}(t) = [w_B\overline{\boldsymbol{x}}_A + (w_A\overline{\boldsymbol{x}}_B - w_B\overline{\boldsymbol{x}}_A)t : w_A w_B]^T$$

In this case, the parameterization is linear, of course.

It should be noted that barycentric coordinates can be computed directly in homogeneous coordinates without division operations as well, see [12], using generalized cross-product.

The above presented approach is quite simple for understanding projective space principles.

V. BARYCENTRIC COORDINATES

Barycentric coordinates are very often used not only in computer graphics, computer graphics and visualization. It is known that computation of barycentric coordinates leads to solution of linear system of equations (LSE). A solution of LSE is equivalent to the generalized cross-product. This

result to computation of the barycentric coordinates directly using generalized cross-product without use of division operation and therefore the computation is more robust in general. The barycentric coordinates are computed as

$$\mathbf{b} = \boldsymbol{\xi} \times \boldsymbol{\eta} \times \mathbf{w} \qquad \boldsymbol{\tau}^T \mathbf{b} = 0$$

$$\mathbf{b} = [b_1, b_2, b_3, b_4]^T \quad \boldsymbol{\xi} = [x_1, x_2, x_3, x]^T \quad \mathbf{w} = [1, 1, 1, 1]^T$$

$$\boldsymbol{\eta} = [y_1, y_2, y_3, y]^T \quad det \begin{vmatrix} \tau_1 & \tau_2 & \tau_3 & \tau \\ x_1 & x_2 & x_3 & x \\ y_1 & y_2 & y_3 & y \\ 1 & 1 & 1 & 1 \end{vmatrix} = 0$$

The barycentric coordinates of the point **x** are then given as

$$a_1 = -\frac{b_1}{b_4},\ a_2 = -\frac{b_2}{b_4},\ a_3 = -\frac{b_3}{b_4}$$

The above formulas shows that the computation of the barycentric coordinates is quite simple. If hardware acceleration using matrix-vector operation is used, the computation is very fast. It is important to note that the similar scheme for the barycentric coordinates computation is valid for homogeneous coordinates as the determinant is multi-linear.

VI. INTERSECTION COMPUTATIONS

A. Line-plane intersection

There is lot of algorithms based on line intersections, like ray-tracing, line clipping etc. Let us consider a simple case of intersection of a line in a parametric form with a plane, which is the fundamental principle of many algorithms, e.g. Cyrus-Beck's (CB) line clipping, see Fig. 2 for E^2 analogy (planes are "degenerated" to edges).

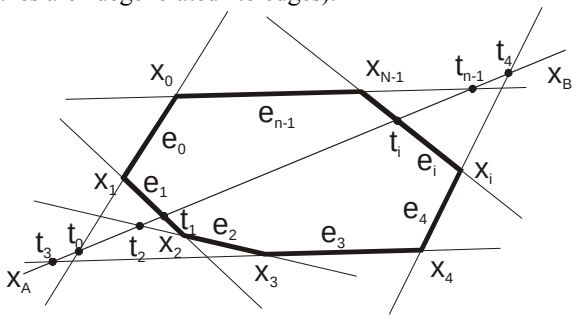

Figure 2. Line clipping by a convex polygon

Let us consider two planes $\boldsymbol{\rho}_1$ and $\boldsymbol{\rho}_2$ and a line $\boldsymbol{p}$ in a parametric form given in homogeneous coordinates by two points $\boldsymbol{x}_A$ and $\boldsymbol{x}_B$.

In many algorithms including CB algorithm the relation $t_1 < t_2$, needs to be evaluated, e.g. to get an order of intersection points. For this *only* ***monotonic parameterization*** on the line $\boldsymbol{p}$ is needed. It means that the linear interpolation with non-linear parameterization presented above can be used efficiently.

The CB algorithm is based on an intersection solution of a line $\boldsymbol{p}$ given in a parametric form and a plane $\boldsymbol{\rho}$ in E^3 (or a line in E^2) given in the implicit form as follows:

$$\boldsymbol{\rho}: \boldsymbol{a}^T \boldsymbol{x} = 0 \qquad \boldsymbol{p}: \boldsymbol{x}(\tau) = \boldsymbol{x}_A + \boldsymbol{s}\tau$$

It should be noted that all vectors are vectors of the projective space, i.e. they have homogeneous coordinates. Therefore, it is easy to compute the intersection point as

$$\boldsymbol{a}^T \boldsymbol{x}_A + \boldsymbol{a}^T \boldsymbol{s}\tau = 0 \qquad \text{then} \qquad \tau = -\boldsymbol{a}^T \boldsymbol{x}_A / \boldsymbol{a}^T \boldsymbol{s}$$

The parameter can be represented by a "*projective scalar*" as

$$\tau = [-\boldsymbol{a}^T \boldsymbol{x}_A : \boldsymbol{a}^T \boldsymbol{s}]^T = [\bar{\tau} : \tau_w]^T$$

Then the CB's algorithm can be modified as follows:

$\tau_{min} = [-\infty : 1]^T; \qquad \tau_{max} = [\infty : 1]^T$

for i:=1 **to** N_planes **do**

{ $\boldsymbol{\tau} := [-\boldsymbol{a}^T \boldsymbol{x}_A : \boldsymbol{a}^T \boldsymbol{s}]^T$; # $\boldsymbol{\tau} = [\bar{\tau} : \tau_w]^T$ #

if $\tau_w < 0$ **then** $\boldsymbol{\tau} := -\boldsymbol{\tau}$;

τ_w coordinate needs to be non-negative

if $\bar{\tau} < 0$ **then** $\boldsymbol{\tau}_{min} = \max(\tau_{min}, \tau)$

else $\boldsymbol{\tau}_{max} = \min(\tau_{max}, \tau)$

}

if NON-EMPTY ($\boldsymbol{\tau}_{min}$, $\boldsymbol{\tau}_{max}$) = **true then**

#equivalent test to $t_1 < t_2$ #

{ $\boldsymbol{x}_{A\_new} = \cdots\ldots; \boldsymbol{x}_{B\_new} = \cdots$ }

Algorithm 1

The above shows that no division operation is needed. Experiments made proved a slight speed-up for the case when the points are given in the Euclidean space (the algorithm has been simplified as w_A, $w_B = 1$ of course) and significant speed-up for the case when the points of the clipped line are given in the homogeneous coordinates. As N_planes, the number of planes, is usually higher, the speed-up will grow with the number of planes of the given convex polyhedron.

B. Intersection of two planes

Intersection of two planes is another case very often solved in computer graphics and vision. Unfortunately in many cases available solutions are not robust or formula are neither simple nor convenient for GPU use.

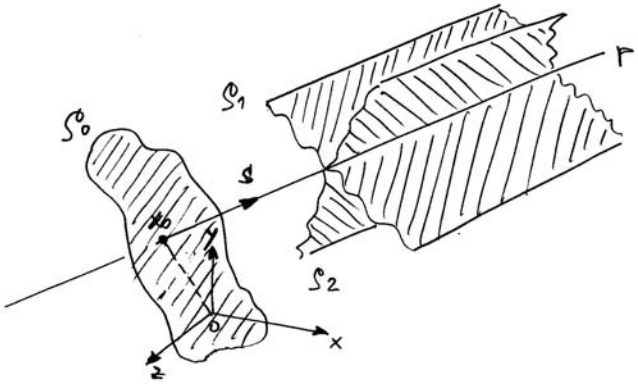

Figure 3. Intersection of two planes

If the projective space is used, the solution is quite simple. Let us consider two planes $\boldsymbol{\rho}_1$ and $\boldsymbol{\rho}_1$ given as

$$\boldsymbol{\rho}_1 = [a_1, b_1, c_1 : d_1]^T \qquad \boldsymbol{\rho}_2 = [a_2, b_2, c_2 : d_2]^T$$

It means that normal vectors of those planes are

$$\boldsymbol{n}_1 = [a_1, b_1, c_1]^T \, \boldsymbol{n}_2 = [a_2, b_2, c_2]^T$$

It is obvious that a directional vector of a line is determined as an intersection of two planes $\boldsymbol{\rho}_1$ and $\boldsymbol{\rho}_1$ given as

$$\boldsymbol{s} = \boldsymbol{n}_1 \times \boldsymbol{n}_2$$

However, the "starting" point $\boldsymbol{x}_0$ of the line is determined in quite complicated ways, sometimes even not robustly enough and based on a user choice of some value, or proposes solution of a system of linear equations [Gol90], [20].

The following "standard" formula can typically be found:

$$\boldsymbol{n}_3 = \boldsymbol{n}_1 \times \boldsymbol{n}_2$$

$$x_0 = \frac{d_2 \begin{vmatrix} b_1 & c_1 \\ b_3 & c_3 \end{vmatrix} - d_1 \begin{vmatrix} b_2 & c_2 \\ b_3 & c_3 \end{vmatrix}}{DET}$$

$$y_0 = \frac{d_2 \begin{vmatrix} a_3 & c_3 \\ a_1 & c_1 \end{vmatrix} - d_1 \begin{vmatrix} a_3 & c_3 \\ a_2 & c_2 \end{vmatrix}}{DET}$$

$$z_0 = \frac{d_2 \begin{vmatrix} a_1 & b_1 \\ a_3 & b_3 \end{vmatrix} - d_1 \begin{vmatrix} a_2 & b_2 \\ a_3 & b_3 \end{vmatrix}}{DET}$$

$$DET = \begin{vmatrix} a_1 & b_1 & c_1 \\ a_2 & b_2 & c_2 \\ a_3 & b_3 & c_3 \end{vmatrix}$$

The formula is quite "horrible" one and for students not acceptable as it is too complex and they do not see from the formula comes from.

As outline below, there is a quite simple geometrical explanation and solution. So the first question is how to find the "starting" point $\boldsymbol{x}_0$ of the line $\boldsymbol{p}$ given by two planes $\boldsymbol{\rho}_1$ and $\boldsymbol{\rho}_2$. If a robust solution is required a user should be prevented from a selection of some "parameters".

Let us imagine that there exists a plane $\boldsymbol{\rho}_0$, whose normal vector is given as $\boldsymbol{s} = \boldsymbol{n}_1 \times \boldsymbol{n}_2$.

It means that its position needs to be "fixed" in the space. As there is no other requirement on this plane, we can "fix" it so it passes through the origin of the coordinate system, i.e. the plane $\boldsymbol{\rho}_0$ is given as

$$\boldsymbol{\rho}_0 = [a_0, b_0, c_0 : 0]^T$$

and the line $\boldsymbol{p}$ is orthogonal to the plane $\boldsymbol{\rho}_0$ – resulting in a robust geometric position.

Now, the intersection point of three planes is the point $\boldsymbol{x}_0$ we are looking for. Coordinates of the point $\boldsymbol{x}_0$ are determined by generalized cross-product as

$$\boldsymbol{x}_0 = \boldsymbol{\rho}_1 \times \boldsymbol{\rho}_2 \times \boldsymbol{\rho}_0$$

As this formula is very compact and the cross-product is a GPU instruction, it is suitable for GPU use. See the Appendix for the extended cross-product GPU implementation.

From the formulation presented above, it can be seen that it is not only very simple, easy to understand and remember, but also easy to implement. It is obvious that the point $\boldsymbol{x}_0$ is also the closest point on the line to the origin, too. As a result the Plücker coordinates formulation of this problem solution is not needed when looking for such properties.

VII. WINDOW CLIPPING

Line or line segment clipping against rectangular window or convex polygon in E^2 is a basic operation in computer graphics. There are well-known Cohen-Sutherland algorithm and many other algorithms. Some of these well-known algorithms are not easy to implement due to their complexity. However, there is a simple and effective solution based on projective representation and the line clipping algorithm can be described using 7 lines only.

The algorithm is based on classification of the window vertices resulting in a binary code which is the address to TAB_1 and TAB_2 tables, where indices of intersected edges are stored. Coordinates of intersection points are computed as a cross-product of the given line and intersected edges.

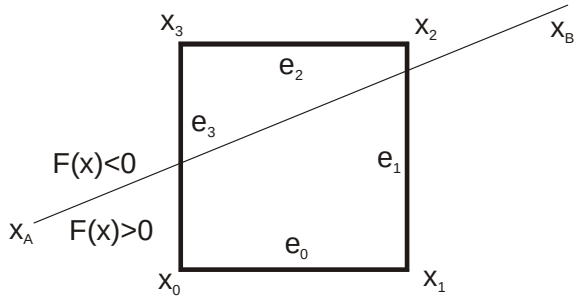

Figure 4. Line clipping by a rectangular window

procedure CLIP_L; { **input**: $\boldsymbol{x}_A$, $\boldsymbol{x}_B$ }
$\boldsymbol{x}_A=[x_A,y_A:w_A]^T$, $\boldsymbol{x}_B=[x_B,y_B:w_B]^T$
$\boldsymbol{x}_A$, $\boldsymbol{x}_B$ –in homogeneous coordinates
the **EXIT** statement ends the procedure
{ $\boldsymbol{p}$:= $\boldsymbol{x}_A$ x $\boldsymbol{x}_B$; { ax+by+c = 0; $\boldsymbol{p}$ = $[a,b,c]^T$ }
for k:=0 **to** N-1 do # $x_k=[x_k,y_k:w_k]^T$ #
if $\boldsymbol{p}^T\boldsymbol{x}_k \geq 0$ **then** $\mathbf{c}_k$:=1 **else** c_k:=0;
if $\mathbf{c} = [0000]^T$ **or** $\mathbf{c} = [1111]^T$ **then EXIT**;
i:= TAB_1[**c**]; j:= TAB_2[**c**];
$\boldsymbol{x}_A$:= $\boldsymbol{p}$ x $\boldsymbol{e}_i$; $\boldsymbol{x}_B$:= $\boldsymbol{p}$ x $\boldsymbol{e}_j$;
DRAW ($\boldsymbol{x}_A$; $\boldsymbol{x}_B$)
}

Algorithm 2

Where N is a number of edges of the clipping window TAB_1 and TAB_2 are constant tables with window edges classifications, for details see [15].

For the situation at Fig. 4, CODE=[0011]=3, TAB_1=1 and TAB_2=3. The algorithm itself is easy to explain and implement, too.

The line segment clipping algorithm is a little bit longer, but still easy to implement, see [15], and to modify for line or line segment clipping by a convex polygon as well. It should be noted that due to the principle of duality, line clipping by a convex polygon is dual to a point-in-polygon test, which is of $O(lgN)$ complexity. Line clipping algorithm of $O(lgN)$ complexity is more complex, see [16].

VIII. CONCLUSION

Application of projective geometry principles to the computational pipeline, especially in the field of geometry and computer graphics can bring new algorithms that are more robust and faster even for the Euclidean space. Due to the formulation, is very convenient for vector-vector or matrix-vector architectures, like GPU and a significant speed-up can be expected as well. Projective geometry can easily explain several methods in a more simple way and also provide new formula and geometric representation which contributes to students' better understanding.

ACKNOWLEDGMENT

The author would like to thank students and colleagues at the VSB-Technical University and University of West Bohemia for discussions and suggestions, reviewers for their critical comments and constructive recommendations.

This work was partially supported by SGS, VSB-Technical University of Ostrava, Czech Republic, under the grant No. SP2011/163.

REFERENCES

[1] D. van Arsdale, "Homogeneous Transformation Matrices for Computer Graphics," Computers & Graphics, Vol.18, No. 2, March-April 1994, pp. 177-191, 1994.

[2] M.M.S. Coxeter, "Projective Geometry," Toronto: University of Toronto, 2nd edition, 1974.

[3] R. Goldman, "Intersection of Three Planes," Graphics Gems (Ed. A.Glassner-), Academic Press, pp. 305-310, 1990.

[4] R. Hartley and A. Zisserman, "Multiple View Geometry in Computer Vision," Cambridge University Press, 2000.

[5] M. Johnson, "Proof by Duality: or the Discovery of "New" Theorems," Mathematics Today, December, 1996.

[6] M.E. Loaiza, A.B. Raposo and M. Gattass, "Multi-camera Calibration Based on an Invariant Pattern," Computers & Graphics, Vol. 35, Issue 2, pp. 198-207, 2011.

[7] J.R. Miller, "Vector Geometry for Computer Graphics," IEEE Computer Graphics and Applications, Vol. 19, No. 3., 1999.

[8] V. Skala, "Geometric Computation, Duality and Projective Space," IW-LGK workshop proceedings, pp.105-111, Dresden University of Technology, 2011.

[9] V. Skala and V. Ondracka, "A Precision of Computation in the Projective Space," Recent Researchers in Computer Science, WSEAS, pp. 35-40, 2011.

[10] V. Skala, "Duality and Intersection Computation in Projective Space with GPU support," ASM 2010 Conf., pp. 66-71, NAUN, 2010.

[11] V. Skala, "Computation in Projective Space," MAMETICS 2009 Conf., pp. 152-157, WSEAS, 2009.

[12] V. Skala, "Barycentric Coordinates Computation in Homogeneous Coordinates," Computers & Graphics, Elsevier, Vol. 32, No. 1, pp. 120-127, 2008.

[13] V. Skala, "Intersection Computation in Projective Space using Homogeneous Coordinates," International Journal on Image and Graphics, Vol. 8, No. 4, pp. 615-628, 2008.

[14] V. Skala, "Length, Area and Volume Computation in Homogeneous Coordinates," International Journal of Image and Graphics, Vol. 6., No. 4, pp. 625-639, 2006.

[15] V. Skala, "A New Approach to Line and Line Segment Clipping in Homogeneous Coordinates," The Visual Computer, Vol.21, No.11, pp.905-914, Springer Verlag, 2005.

[16] V. Skala, "O(lg N) Line Clipping Algorithm in E^2," Computers & Graphics, Pergamon Press, Vol.18, No.4, 1994.

[17] J. Stolfi, "Oriented Projective Geometry," Academic Press, 2001.

[18] F. Yamaguchi, "Computer-Aided Geometric Design," Springer Verlag, 2002.

[19] Y. Yu, "Efficient Visibility Processing for Projective Texture Mapping," Computers & Graphics, Vol. 23, No. 2, pp. 245-253, 1999.

WEB references

[20] Softsurfer http://softsurfer.com/ <retrieved on 2011-12-05>

APPENDIX

The cross-product in 4D defined as

$$\boldsymbol{x_1} \times \boldsymbol{x_2} \times \boldsymbol{x_3} = det \begin{vmatrix} \boldsymbol{i} & \boldsymbol{j} & \boldsymbol{k} & \boldsymbol{l} \\ x_1 & y_1 & z_1 & w_1 \\ x_2 & y_2 & z_2 & w_2 \\ x_3 & y_3 & z_3 & w_3 \end{vmatrix}$$

can be implemented in Cg/HLSL on GPU as follows:

```
float4 cross_4D(float4 x1, float4 x2, float4 x3)
{
        float4 a;
        a.x=dot(x1.yzw, cross(x2.yzw, x3.yzw));
        a.y=-dot(x1.xzw, cross(x2.xzw, x3.xzw));
        // or a.y=dot(x1.xzw, cross(x3.xzw, x2.xzw));
        a.z=dot(x1.xyw, cross(x2.xyw, x3.xyw));
        a.w=-dot(x1.xyz, cross(x2.xyz, x3.xyz));
        // or a.w=dot(x1.xyz, cross(x3.xyz, x2.xyz));

        return a;
}
```

or more compactly

```
float4 cross_4D(float4 x1, float4 x2, float4 x3)
{
        return ( dot(x1.yzw, cross(x2.yzw, x3.yzw)),
        -dot(x1.xzw, cross(x2.xzw, x3.xzw)),
        dot(x1.xyw, cross(x2.xyw, x3.xyw)),
        -dot(x1.xyz, cross(x2.xyz, x3.xyz)) );
}
```