Xync

Description of the Programming Interface to the X–cito Camera Tracking Program

Version 1.2.0

Sep. 1, 1998

Xync
Technopark
Rathausallee 10
D–53757 Sankt Augustin
Germany

phone: +49 2241 / 14 35 35
fax: +49 2241 / 14 35 36

E–mail: support@xync.com
WWW: http://www.xync.com

Contents

1 Document Version History

Version 1.0 (Nov. 5, 1996):

- The first version of this document and accompanying source code.

Version 1.0.1 (Nov. 7, 1996):

- Added a postscript version of this document to the distribution files.
- Added description of parameter binary to function `trkNetOpenWrite`.
- Error in file `write_struct.c` corrected. Camera position was sent incorrectly if format option `trkStudioY_Up` was used.
- Corrected a few typing errors.

Version 1.1.0 (Apr. 2, 1997):

- A new function `trkMemConfigure` has been introduced. See the description in subsection *"Data Transfer by Memory"* of section *"Interface Functions"*. If function `trkMemConfigure` is not used at all or if it is used with the default parameter value `trkMemDefault` then data transfer will behave exactly the same as before the introduction of function `trkMemConfigure`. Therefore programs using old source code which misses `trkMemConfigure` will still be able to communicate with programs using new source code which implements `trkMemConfigure`.

Version 1.2.0 (Sep. 1, 1998):

- The *X–cito camera tracking program* is the successor of the former *DMC* camera tracking program. This document is an adaption of the respective DMC document with all references to the company DMC substituted by references to *Xync* and *X–cito* accordingly.
- A new function `trkNetBufReadParams` has been introduced. It is intended to be a replacement for function `trkNetReadParams` in cases where data transfer across a network suffers from irregular delay intervals. See the description in subsection *"Data Transfer by Network"* of section *"Interface Functions"*.

2 Introduction

The *X–cito camera tracking program* uses data gathered from sensors which are connected to a TV camera to calculate a variety of camera parameters under real–time conditions. These parameters fully describe a so–called "virtual camera" which is used by computers in a Virtual Studio environment to render 3D images.

Xync presents a **public programming interface** consisting of a few source code files written in C which enables almost any program to receive camera parameters that were calculated by the camera tracking program. This document describes how to use this programming interface in order to obtain camera parameters and how to interpret these parameters.

Three different methods of passing camera parameters from one program to the other are supported. The first and probably most frequently used method uses **shared memory** on a UNIX platform to transfer data between two programs that are running concurrently on the same computer. The second method sends data in the shape of datagrams across a **network** from one computer to another using the **Internet UDP protocol**. Finally, the third method simply stores camera parameters in a **file** and later retrieves this data from the file again. This last method is of course only suitable for **post–production purposes**.

In the following sections of this document the camera model, camera parameters and additional information used by the tracking program will be discussed first, followed by a description of the source code files and the interface functions they supply.

The programming interface consists of a few global functions and type definitions. All of these declarations start with the prefix "`trk`" to avoid naming conflicts.

3 Camera Constants

In addition to supplying camera parameters, which may change at the field rate of the video signal, the *X–cito camera tracking program* offers information about various constants describing the type of camera that is being tracked. These constants might help programs in processing the camera parameters more adequately. Since they do not change during the tracking of camera movements they are sent only once at the start of each tracking process.

The programming interface stores camera constants in a structure of type `trkCameraConstants` which contains the following variables:

```
typedef struct
   {
   int      imageWidth, imageHeight;
   int      blankLeft, blankRight, blankTop, blankBottom;
   double   chipWidth, chipHeight, fakeChipWidth, fakeChipHeight;
   } trkCameraConstants;
```

`imageWidth`, `imageHeight`
: These variables specify the resolution of the digitized camera image in pixels. The ratio of width and height does not have to be the same as that of the CCD chip which produced the camera image! Standard values for image width and height for a D1 digital video signal are: `imageWidth` = 720, `imageHeight` = 576.

`blankLeft`, `blankRight`
: The conversion from an analog into a digital video signal may introduce **stripes of black pixels** at the left and right border of the camera image. These stripes do not overlay any image data. Instead, the actual camera image is slightly "compressed" horizontally and blank pixel columns are added on the left and right. The variables `blankLeft` and `blankRight` specify the number of blank pixel columns on each side. Typical values are between 5 and 15.

`blankTop`, `blankBottom`
: These variables are used similar to `blankLeft` and `blankRight` for the specification of blank pixel rows at the top and bottom of the camera image. The usual conversion from PAL or NTSC to D1 however does not introduce any blank pixel rows. Therefore, the values of `blankTop` and `blankBottom` are currently always zero.

`chipWidth`, `chipHeight`
: The optical image passing through the camera lens is converted into a video signal by the **CCD chip** inside the camera. The variables `chipWidth` and `chipHeight` specify the size of this chip or, to be more precise, the area of the chip which is read out to form the camera image. Both values are measured in MILLIMETERS! The chip size is important when it comes to calculating a field of view angle (see the next section *"Camera Model"*). The ratio of chip width and height is also called the **"aspect ratio"** of the camera image and is usually 4:3.

`fakeChipWidth`, `fakeChipHeight`

If a program ignores the fact that the digitized camera image contains columns (and possibly rows) of blank pixels at its borders then it must use the values stored in variables `fakeChipWidth` and `fakeChipHeight` as the CCD chip width and height. These values are slightly different from `chipWidth` and `chipHeight`. See the section *"Camera Parameters"* for more explanations.

4 Camera Model

The *X–cito camera tracking program* uses a specific camera model which is designed to approximate the "real" camera that is being tracked. This camera model is described in detail below in order to avoid a misinterpretation of camera parameters.

4.1 Pinhole Camera

The camera model is basically that of a pinhole camera with a few additions regarding lens distorsion and depth of field simulation. A pinhole camera projects a 3D point onto a 2D plane along the line passing through the point and the **center of projection**. The **projection plane** here lies between the 3D point and the center of projection, as opposed to the model of a pinhole camera used in physics, where the projection plane lies "behind" the center of projection and the camera image is created "upside down".

To describe the position, orientation and magnification of the pinhole camera the following coordinate system is introduced. The so–called **"camera coordinate system"** is a 3–dimensional right–handed cartesian coordinate system. Its origin is the center of projection. The projection plane lies parallel to the xz–plane (perpendicular to the y–axis) and intersects the y–axis at coordinate $(0, d, 0)$. The positive y–axis therefore plays the role of the **"optical axis"**. The distance d of the projection plane from the origin is called **"image distance"**.

The **projection area** which makes up the camera image is a rectangular area within the projection plane. Its edges are parallel to the x– and z–axis. Its geometrical center usually lies close to the intersection of the projection plane with the y–axis at coordinate (c_x, d, c_y). The displacement of the center of the projection area from the y–axis is a 2–dimensional vector (c_x, c_y) called **"center shift"**.

Positions in the camera image are expressed as 2–dimensional vectors within the projection plane, with the origin at the center of the projection area and the x– and y–axis pointing in the same direction as the camera coordinate system's x– and z–axis respectively. So the origin of the camera image lies is in the middle of the image, the x–axis points "to the right" and the y–axis points "upwards".

To sum it all up: a 3D point with coordinates (x, y, z) in the camera coordinate system is projected onto a point (x', y') in the camera image according to the following equations.

$$\begin{aligned} x' &= d\,\frac{x}{y} - c_x \\ y' &= d\,\frac{z}{y} - c_y \end{aligned}$$

If the width of the projection area (measured along the x–axis) is r_x and its height (measured along the z–axis) is r_y then the point (x', y') actually lies within the boundaries of the camera image if the following equations are both true:

$$\begin{aligned} -\frac{r_x}{2} &\le x' \le \frac{r_x}{2} \\ -\frac{r_y}{2} &\le y' \le \frac{r_y}{2} \end{aligned}$$

The section of 3D space that is visible in the camera image can also be specified by a so–called **"field of view"** angle. Either the horizontal angle a_h or the vertical angle a_v or the diagonal angle a_d is used and calculated like this:

$$\begin{aligned} \tan\left(\frac{a_h}{2}\right) &= \frac{r_x}{2\,d} \\ \tan\left(\frac{a_v}{2}\right) &= \frac{r_y}{2\,d} \\ \tan\left(\frac{a_d}{2}\right) &= \frac{\sqrt{{r_x}^2 + {r_y}^2}}{2\,d} \end{aligned}$$

4.2 Image Rasterization

The projection area in the pinhole camera model is the equivalent of the CCD chip in a real camera (with r_x being the chip width and r_y being the chip height). To convert the position of a point on the chip to a pixel position in the rasterized camera image, the information presented in the previous section *"Camera Constants"* has to be used. Assuming that the origin in the rasterized image is in the lower left corner of the image and furthermore assuming that the x–coordinates are counted from left to right and the y–coordinates from bottom to top, the point (x', y') on the chip (that is, on the projection plane) will be found at pixel position (x'', y'') in the rasterized image:

$$\begin{aligned} x'' &= m_x\,x' + b_x \\ y'' &= m_y\,y' + b_y \end{aligned}$$

with constants

$$\begin{aligned} m_x &= \frac{\texttt{imageWidth} - \texttt{blankLeft} - \texttt{blankRight}}{\texttt{chipWidth}} \\ m_y &= \frac{\texttt{imageHeight} - \texttt{blankBottom} - \texttt{blankTop}}{\texttt{chipHeight}} \end{aligned}$$

$$\begin{aligned} b_x &= \texttt{blankLeft} + \tfrac{1}{2}\,(\texttt{imageWidth} - \texttt{blankLeft} - \texttt{blankRight}) \\ b_y &= \texttt{blankBottom} + \tfrac{1}{2}\,(\texttt{imageHeight} - \texttt{blankBottom} - \texttt{blankTop}) \end{aligned}$$

4.3 Camera Position and Orientation

So far, the mapping of 3D points onto 2D points in the camera image has been described relative to the camera coordinate system. A global coordinate system called the **"studio coordinate system"** is necessary in order to be able to specify the position and orientation of the pinhole camera and its camera coordinate system in a studio environment.

As the name suggests, the studio coordinate system used by the tracking program is specified in relation to the studio surrounding the real camera. The studio floor represents the xy–plane of the coordinate system. The z–axis points upwards out of the floor towards the studio ceiling. The origin of the coordinate system is some

specially marked point on the studio floor. The x– and y–axis are picked such that the y–axis points into the studio towards its back wall and the x–axis points to the right wall.

Now the position and orientation of the pinhole camera can be described by the transformation of the studio coordinate system to the camera coordinate system or vice versa. The standard way of doing this in computer graphics is by using a **4x4 homogenous transformation matrix**. Alternatively, the transformation can be specified by the position of the center of projection in studio coordinates and the three so–called **"Euler angles" pan, tilt and roll**. Hereby, the transition from studio to camera coordinate system is done by first moving the origin to the center of projection, then rotating the coordinate system around its z–axis by the "pan" angle (also called "heading" or "yaw"), followed by rotating around the x–axis by the "tilt" angle (also called "pitch") and finally rotating around the y–axis by the "roll" angle (also called "twist"). A positive pan angle means a rotation to the left, a positive tilt angle represents a rotation which moves the camera lens upwards, and a positive roll angle results in a clockwise rotation around the optical axis when viewed from the position of the camera operator.

4.4 Lens Distorsion

The pinhole camera model is extended to allow the treatment of radial lens distorsion which is introduced by the camera lens. Taking lens distorsion into account by distorting computer generated images is not a standard practice in Virtual Studios at the moment, but it will be of importance in the future and will produce better results in the synthesis of real and virtual camera images.

The distorsion model of the *X–cito camera tracking program* uses two **coefficients** called k_1 and k_2 to describe image distorsion. The following equations explain how the position $(\tilde{x}, \tilde{y})$ of a distorted point on the CCD chip can be used to find the position (x', y') which this point would have if there was no lens distorsion:

$$\begin{aligned} x' &= u\,(\tilde{x} + c_x) - c_x \\ y' &= u\,(\tilde{y} + c_y) - c_y \end{aligned}$$

with

$$\begin{aligned} u &= 1 + R\,(k_1 + R\,k_2) \\ R &= (\tilde{x} + c_x)^2 + (\tilde{y} + c_y)^2 \end{aligned}$$

It is important to note that this distorsion correction is done using coordinates on the CCD chip (that is, on the projection area of the pinhole camera) and NOT using pixel coordinates in the rasterized camera image! This makes a difference because pixel coordinates have a different aspect ratio and the coefficients k_1 and k_2 would have to be rescaled to use a different measuring unit.

4.5 Depth of Field

While a pinhole camera always produces sharp images, a real camera may produce partially blurred images depending on the zoom, focus and aperture setting of the

camera lens. Two parameters are used to describe this behavior of real cameras.

The parameter **“focal distance”** specifies the distance from the center of projection at which real objects appear totally sharp in the camera image. The closer an object is to the camera or the farther away it is from the camera the more it appears blurred in the image. The parameter **“aperture”** is intended to describe the magnitude of this “bluriness”. Both parameters are needed in order to calculate the physically correct depth of field value.

There is no standard way of simulating depth of field in computer generated images at the moment. This means that, although the parameter “aperture” is one of the camera parameters offered by the *X–cito camera tracking program*, its interpretation depends on the type of simulation that is used. Anyway, the correct calculation of the “aperture” value requires the connection of a sensor to the aperture ring of the camera lens, which is currently not the case in most Virtual Studios.

A short–term solution to the aperture problem could be the manual setting of this parameter in the 3D rendering process of a Virtual Studio depending on the type of depth of field simulation that is being used. This would make sense because the aperture setting of the camera lens in practice usually remains constant during a recording and does not need to be tracked for every field of the video signal.

5 Camera Parameters

The programming interface to the *X–cito camera tracking program* stores all of the parameters of a virtual camera that were described in the previous section in a structure of type `trkCameraParams`. Unfortunately, there is no general agreement on the format that is used to specify some of these parameters. Therefore, the camera tracking program offers a number of **format options** which should enable any program to receive the camera parameters in the format it can work with best.

The parameters that are contained in a `trkCameraParams` structure are explained below with the help of the variables introduced in the previous section. The format options controlling the parameter format are mentioned in the explanation as well.

```
typedef struct
   {
   unsigned       format;
   trkTransform   t;
   double         fov, centerX, centerY, k1, k2, focdist, aperture;
   unsigned long  counter;
   } trkCameraParams;
```

`format` is an unsigned integer which contains a bit mask of format options. Every format option is a flag specified by a constant, and the constants of all the options that are required are logically ORed together to create the bit mask. The format variable can be used to verify that the parameters contained in the structure really have the format which is expected by the receiving program.

`t` describes the position and orientation of the camera in the studio. This variable is of type `trkTransform`, which is a union of a transformation matrix `t.m` and a structure `t.e` which contains a position vector and Euler angles. Using default option `trkMatrix` indicates that the matrix `t.m` is used, whereas using option `trkEuler` means that the position vector and Euler angles in structure `t.e` are used.

```
typedef union
   {
   double   m [4] [4];
   struct   { double x, y, z, pan, tilt, roll; } e;
   } trkTransform;
```

`t.m` is a 4x4 homogenous transformation matrix. "`t.m [j] [i]`" is the matrix element in row i, column j. If default option `trkCameraToStudio` is used then this matrix represents a transformation from the camera coordinate system to the studio coordinate system. That is, multiplying the matrix from the right with a 4–dimensional homogenous vector that contains the camera coordinates of a point yields the studio coordinates of the same point. Another way of looking at it is to say that the columns of the upper left 3x3 submatrix contain the axes of the camera coordinate system expressed in studio coordinates, and the upper three elements of the rightmost column contain the position of the projection center in studio coordinates. Such a transformation matrix can for example be used by SGI's Performer library.

If option `trkStudioToCamera` is used then the matrix contains the inverse of the matrix described above, that is, it describes a transformation from the studio coordinate system to the camera coordinate system. SGI's OpenGL library is an example of an application which could use a matrix like this.

By default the z–axes of the two coordinate systems point "upwards", which corresponds to using default options `trkCameraZ_Up` and `trkStudioZ_Up`. If the y–axis of the camera coordinate system is expected to be pointing upwards (OpenGL makes this assumption) then option `trkCameraY_Up` is to be used instead. In this case the x–axis remains the same, the z–axis becomes the y–axis and the former positive y–axis becomes the negative z–axis. Likewise, if virtual objects were designed using a studio coordinate system with its y–axis pointing upwards then option `trkStudioY_Up` has to be used.

`t.e` is a structure containing the position of the projection center in studio coordinates (variables `x, y, z`) and the three Euler angles which describe the orientation of the camera (variables `pan, tilt, roll`). The option `trkStudioY_Up` must be used if the y–axis of the studio coordinate system points upwards instead of the default z–axis, which is the case if default option `trkStudioZ_Up` is used.

Using option `trkStudioY_Up` only has an effect on variables `y` and `z` because the meaning of the Euler angles is unaffected by a switch of coordinate axes. The `pan` angle always represents a rotation around the vertical axis (z or y), the `tilt` angle is a rotation around the x–axis which points to the right wall in both cases, and the `roll` angle is always a rotation around the optical axis (y or –z). The options `trkCameraZ_Up` and `trkCameraY_Up` have no effect on the contents of the structure `t.e`.

`fov` contains either the image distance d (if default option `trkImageDistance` is used), or a field of view angle (if option `trkFieldOfView` is used). If the latter is used then one of the options `trkHorizontal` (default), `trkVertical` or `trkDiagnonal` is necessary in order to specify whether the field of view angle is the horizontal (a_h), vertical (a_v) or diagonal (a_d) angle.

The correct treatment of the fov value in a program depends on whether the program takes into account the blank borders in the digitized camera image (see section *"Camera Constants"*). Default option `trkConsiderBlank` indicates that the program knows about the existence of blank borders and handles them correctly, whereas option `trkIgnoreBlank` indicates that the program acts as if there were no blank borders.

The effect of using option `trkIgnoreBlank` is that a few values are changed such that the program which processes the camera parameters may still produce good results. To be more precise, the CCD chip size, specified as `chipWidth` and `chipHeight` in a `trkCameraConstants` structure, is slightly increased by the same amount as the digitized camera image (including the blank borders) is larger than the actual camera image (without the blank borders). This "fake" chip size is stored in variables `fakeChipWidth` and `fakeChipHeight` in a `trkCameraConstants` structure. Option `trkIgnoreBlank` indicates that the fake chip size is to be used instead of the actual chip size when it comes to calculating a field of view angle or interpreting an image distance or determining the aspect ratio.

If the program that receives camera parameters is not only incapable of handling the blank image borders but will also not work with an aspect ratio other than the standard 4:3 then option `trkKeepAspect` must be used instead of default option `trkAdjustAspect`. The `trkKeepAspect` option forces the ratio of the fake chip width and height to be equal to the ratio of the actual chip width and height. The ratio is kept equal by increasing the chip height by the same amount as the chip width. This will yield an erroneous fake chip height, but the error introduced here is assumed to be less noticeable than the error that would occur if the fake chip width was incorrect. The reason for this assumption is that pan movements of a camera are in practice more frequent and usually cover a wider range of angles than tilt movements.

The fake chip size can be expressed in terms of the variables of a `trkCameraConstants` structure:

$$\begin{aligned} \texttt{fakeChipWidth} &= f_x \times \texttt{chipWidth} \\ \texttt{fakeChipHeight} &= \begin{cases} f_y \times \texttt{chipHeight} & (\text{option } \texttt{trkAdjustAspect}) \\ f_x \times \texttt{chipHeight} & (\text{option } \texttt{trkKeepAspect}) \end{cases} \end{aligned}$$

with

$$\begin{aligned} f_x &= \frac{\texttt{imageWidth}}{\texttt{imageWidth} - \texttt{blankLeft} - \texttt{blankRight}} \\ f_y &= \frac{\texttt{imageHeight}}{\texttt{imageHeight} - \texttt{blankBottom} - \texttt{blankTop}} \end{aligned}$$

`centerX, centerY` are the coordinates of the center shift (c_x, c_y) introduced in the previous section *"Camera Model"*. The center shift specifies how far the geometrical center of the CCD chip is shifted away from intersection of the optical axis with the chip plane. Its value is not a constant but changes as the zoom and focus settings of the camera lens are altered.

If default option `trkShiftOnChip` is used then the center shift is measured on the CCD chip, as opposed to the use of option `trkShiftInPixels` which will cause the center shift to be expressed in image pixels.

Option `trkIgnoreBlank` also has an effect on the value of the center shift because it causes a constant offset to be added to the center shift in order to compensate asymmetrical widths of blank image borders.

Ignoring the center shift, that is, assuming it is zero, will often produce still acceptable program results, but using it will improve the results.

`k1, k2` are the distorsion coefficients k_1 and k_2 described in the previous section *"Camera Model"*.

`focdist, aperture` are the parameters "focal distance" and "aperture" required for depth of field simulation. These were explained in the previous section as well.

`counter` is an unsigned long integer which is incremented by one in every new set of camera parameters. The `counter` value can be used to find out if parameters were missed or if they arrived out of order, which might theoretically happen when sending camera parameters in datagrams across a network.

The following measuring units are used for the camera parameters:

- meters for all distances outside of the camera, that is, for the elements of matrix `t.m` and variables `t.e.x, t.e.y, t.e.z` and `focdist`,
- millimeters for all distances inside the camera, that is, for the variables `fov` (if option `trkImageDistance` is used), and `centerX, centerY` (if option `trkShiftOnChip` is used),
- mm^{-2} for `k1` and mm^{-4} for `k2`,
- pixels for variables `centerX` and `centerY` if option `trkShiftInPixels` is used,
- degrees for all angles, that is, for the variables `t.e.pan, t.e.tilt, t.e.roll` and `fov` (if option `trkFieldOfView` is used).

6 Source Code

6.1 Files

The following source code files constitute the programming interface to the *X–cito camera tracking program*. They can be compiled on various UNIX operating systems using a C or C++ compiler.

`trk_struct.h`	Header file containing the declarations of types `trkCameraParams` and `trkCameraConstants` and of the format option and error constants as well as a few help functions which are used by other files.
`trk_struct.c`	Implementation of the help functions declared in `trk_struct.h`.
`trk_mem.h`	Declarations of the functions which are transfering camera parameters to and from another process on the same computer by the use of shared memory.
`trk_mem.c`	Implementation of functions declared in `trk_mem.h`. On a computer running SGI's IRIX operating system the implementation can utilize memory from a so–called "shared arena", whereas on other UNIX derivates it uses System V style shared memory.
`trk_net.h`	Declarations of the functions which are transfering camera parameters across a network from one computer to another.
`trk_net.c`	Implementation of functions declared in `trk_net.h` using BSD sockets and the Internet UDP protocol to transfer data in the shape of datagrams.
`trk_file.h`	Declarations of the functions which store camera parameters in a file and retrieve parameters from a file.
`trk_file.c`	Implementation of functions declared in `trk_file.h` using standard I/O library functions.
`read_struct.h`	Header file used by test programs `read_mem`, `read_net` and `read_file`.
`read_struct.c`	Source code used by test programs `read_mem`, `read_net` and `read_file`.
`write_struct.h`	Header file used by test programs `write_mem`, `write_net` and `write_file`.
`write_struct.c`	Source code used by test programs `write_mem`, `write_net` and `write_file`.

`read_mem.c`	Test program which reads camera parameters from shared memory.
`write_mem.c`	Test program which writes camera parameters to shared memory.
`read_net.c`	Test program which receives camera parameters coming in from the network.
`write_net.c`	Test program which sends camera parameters onto the network.
`read_file.c`	Test program which reads camera parameters from a file.
`write_file.c`	Test program which writes camera parameters to a file.
`Makefile`	A very simple makefile which can be used to compile the source code files listed above.

Part of the source code in the files `trk_struct.c`, `trk_mem.c`, `trk_net.c` and `trk_file.c` is only needed by tracking programs which need to send away camera parameters. The source code for sending parameters can be left out when compiling by calling the compiler with the additional command line option `-DONLY_READ`. This will cause the object files to be smaller, but will make compilation of the test programs `write_mem`, `write_net` and `write_file` impossible. See the makefile for an example of how to use the option `-DONLY_READ`.

6.2 Test Programs

The test programs `read_mem`, `read_net` and `read_file` demonstrate how to use the programming interface for receiving camera parameters. After their successful initialization (`read_mem` connects itself to a shared memory buffer, `read_net` connects to a UDP port and `read_file` opens a file) they will print camera parameters onto standard output. Programs `read_mem` and `read_net` will run for thirty seconds and then terminate. During this half minute the programs will poll the shared memory buffer or the UDP port once every second to see if there is a new set of camera parameters available. If this is the case they will print the new parameters. Program `read_file` will keep on reading and printing parameters until the end of the input file is reached.

The test programs `write_mem`, `write_net` and `write_file` are the counterparts of the three test programs described above. They take on the role of a camera tracking program and write camera parameters which the `read_...` programs or other programs wanting camera parameters are supposed to read afterwards. The `write_...` programs test the most important camera parameters one by one by changing the value of one parameter a couple of times and leaving all other parameters constant. The parameters are tested in this order: x–, y– and z–position, pan, tilt and roll angle, field of view, center shift in x– and y–direction. All other parameters are not tested.

All test programs require command line parameters which are needed for the initialization of the data transfer. Calling the programs without any command line parameters will bring up a message about what parameters are required. The three `write_...` programs also accept a list of identifiers on the command line which correspond to the format options explained in the section *"Camera Parameters"*. The recognized identifiers for the format options are listed below together with

their corresponding option constant. The options are arranged in pairs, the default option is named first and the alternative option second.

option constant	**identifier**
`trkMatrix`	`matrix`
`trkEuler`	`euler`
`trkCameraToStudio`	`camera_to_studio`
`trkStudioToCamera`	`studio_to_camera`
`trkCameraZ_Up`	`camera_z_up`
`trkCameraY_Up`	`camera_y_up`
`trkStudioZ_Up`	`studio_z_up`
`trkStudioY_Up`	`studio_y_up`
`trkImageDistance`	`image_distance`
`trkFieldOfView`	`field_of_view`
`trkHorizontal`	`horizontal`
`trkVertical`	`vertical`
`trkDiagonal`	`diagonal`
`trkConsiderBlank`	`consider_blank`
`trkIgnoreBlank`	`ignore_blank`
`trkAdjustAspect`	`adjust_aspect`
`trkKeepAspect`	`keep_aspect`
`trkShiftOnChip`	`shift_on_chip`
`trkShiftInPixels`	`shift_in_pixels`

Important Note:

The basis for the transfer of camera parameters from the *X–cito camera tracking program* to any other program is a shared data format that is expected in a shared memory buffer, in a UDP datagram or in a file. Anyone unwilling to use the source code files described in this section could just as well implement source code on his/her own if he/she studies the source code listed above to find out about the data format that is being used. Nevertheless, using the original source code presented here has the advantage of having a convenient interface at hand right away and of being able to adapt to changes in future versions very quickly.

7 Interface Functions

The functions of the programming interface can be grouped into three categories according to the data transfer method they are implementing: functions starting with `trkMem...` make use of shared memory, functions starting with `trkNet...` handle the transmission of data across a network, and functions starting with `trkFile...` read from and write to a file.

Each type of data transfer has to be initiated by calling an initialization function and has to be terminated by calling a termination function. Functions for reading and writing camera parameters complement the interface.

7.1 Handles

All initialization functions require a parameter which contains the address of a variable of type `trkHandle`. A new value will be assigned to this variable if the initialization was successful. From now on it serves as a handle and has to be used when calling any function that transfers data or terminates a data connection. The type `trkHandle` is actually a pointer to "void", that is, a pointer which points to some opaque data that is used internally by the implementation.

7.2 Error Codes

Most functions in the programming interface return a variable of enumeration type `trkError`. The value of this variable is used to specify what type of error occured during the execution of the function. Below is a list of the declared error constants and their meanings. The function `trkErrorMessage` can be used to convert an error code into a message string. `trkErrorMessage` expects a variable of type `trkError` as its parameter and returns a pointer to a string which briefly describes the error.

`trkOK`	The desired operation was completed successfully.
`trkFailed`	The desired operation failed. If this error is returned by an initialization function then the initialization was unsuccessful and data transfer cannot be started. If this error is returned by a function which reads or writes data then the data transfer should be stopped.
`trkInvalidHandle`	The function was called with an invalid handle parameter. Only handles which were assigned by an initialization function may be used, and only as long as they have not been invalidated by a termination function.
`trkInvalidVersion`	The desired operation could not be completed because data of an invalid format was encountered, either in a shared memory buffer, in a UDP datagram or in a file. This reason for this might be a software version conflict.

`trkUnavailable` There is no new data available. This error is only returned by functions which read data. The functions for reading from shared memory and from the network will not wait for new data to arrive and return immediately, so this error just means "Try again later!". The function reading from a file will return this error when the end of the input file is encountered.

7.3 Data Transfer by Memory

It is possible to track the movements of several cameras simultaneously on one computer by using one or several camera tracking processes. Therefore, a program which wants to obtain parameters through shared memory has to specify which camera's parameters it wants to receive. Integer numbers called "slot values" are used for the purpose of camera identification. The shared memory initialization function `trkMemOpen` takes such a slot value as one of its parameters.

Every camera tracking process which is configured to send camera parameters into a shared memory buffer using the public programming interface reads a certain slot value for each camera it is supposed to track from its setup file. The slot values of the cameras that are being tracked at the same time all have to be different so they can be distinguished.

While the processes that are sending camera parameters have to use unique slot values, the interface implementation allows any number of programs to use the same slot value for reading the camera parameters of the same camera out of the shared memory buffer.

The following functions are used to transmit camera parameters with the help of shared memory:

`trkMemConfigure (unsigned flags)`

stores configuration options which will be used with all following calls of function `trkMemOpen`. If the operating system supplies different types of shared memory usage — as the IRIX operating system does — the type of shared memory that `trkMemOpen` will request can be chosen by using the appropriate configuration constants for the `flags` parameter. Currently there are only the following choices:

`trkMemConfigure (trkMemDefault);`

makes `trkMemOpen` use the default type of shared memory, which is a shared arena for IRIX and System V style shared memory for all other UNIX derivates.

`trkMemConfigure (trkMemSysV);`

makes `trkMemOpen` use System V style shared memory if this is supplied by the operating system.

`trkMemConfigure (trkMemArena);`

makes `trkMemOpen` use a shared arena if the operating system is IRIX.

```
trkMemConfigure (trkMemDefault | trkMemRecreate);
```

is an example of how the bitwise OR operation of one of the constants mentioned above with the constant `trkMemRecreate` can be used to tell `trkMemConfigure` that all shared memory which is opened from now on will be handled in the following way: if the number of processes that are connected to the shared memory buffer drops down to one then the last process using the memory will destroy the shared memory buffer and recreate a new one immediately. This feature has been introduced in order to avoid problems occuring when processes do not seem to be able to connect and disconnect from the same shared memory buffer any number of times.

`trkMemOpen (int slot, trkHandle*)`

creates a shared memory buffer or attaches to an already existing one. The meaning of the `slot` parameter is described above.

`trkMemClose (trkHandle)`

disconnects from a shared memory buffer. If the process calling this function is the last one connected to the buffer then the shared memory will be destroyed automatically. Failing to call `trkMemClose` at program end prevents this automatic destruction and the shared memory will keep on existing until the computer is rebooted.

`trkMemCloseAll ()`

calls `trkMemClose` for all currently handles that are currently in use. This function is also registered with the standard `atexit` function so it will always be called after program termination. Calling `trkMemCloseAll` several times will do no harm.

`trkMemReadParams (trkHandle, trkCameraParams*)`

reads camera parameters out of the shared memory buffer and stores them in the specified structure. The function will not wait for new data to arrive. If there are no camera parameters available it will return immediately with the error code `trkUnavailable`.

`trkMemReadConstants (trkHandle, trkCameraConstants*)`

reads camera constants out of the shared memory buffer and stores them in the specified structure. The function will not wait for new data to arrive. If there are no camera constants available it will return immediately with the error code `trkUnavailable`.

`trkMemWriteParams (trkHandle, const trkCameraParams*)`

writes camera parameters which are contained in the specified structure into the shared memory buffer. This function is only needed by tracking programs like the *X–cito camera tracking program*.

`trkMemWriteConstants (trkHandle, const trkCameraConstants*)`

writes camera constants which are contained in the specified structure into the shared memory buffer. This function is only needed by tracking programs like the *X–cito camera tracking program*.

7.4 Data Transfer by Network

If camera parameters are to be sent across a network using the UDP protocol then the sending and receiving process have to agree upon the UDP port of the receiver. This port number can be picked rather freely as long as it does not conflict with other programs which use the same port number and are running at the same time. The *X–cito camera tracking program* is informed abouth the host name and port number to which it is expected to send parameters through a specification in its setup file.

The following functions are used to transmit camera parameters across a network:

`trkNetOpenRead (int localPort, trkHandle*)`

allocates the UDP port specified by parameter `localPort` for the reception of datagrams containing camera parameters.

```
trkNetOpenWrite (const char* remoteHost, int remotePort,
                 int binary, trkHandle* handle)
```

allocates any UDP port on the local computer and remembers `remoteHost` and `remotePort` as the address to which camera parameters will be sent from the allocated port. This function is only needed by tracking programs like the *X–cito camera tracking program*.

If parameter `binary` is not zero then data will be sent in binary format, that is, the functions `trkNetWriteParams` and `trkNetWriteConstants` will copy binary images of variables of types `trkCameraParams` and `trkCameraConstants` into UDP datagrams and send them away. The receiving computer will not be able to interpret these binary images correctly unless it uses the same data sizes, data alignment and byte order as the sending computer.

If parameter binary is zero then data will be sent in ASCII format, that is, `trkCameraParams` and `trkCameraConstants` variables will be stored in datagrams using an ASCII representation. This ASCII format is understood by all computers, but it takes a little bit longer to read and write data this way. The functions `trkNetReadParams` and `trkNetReadConstants` automatically determine whether a UDP datagram contains data in binary or ASCII format and extract the data accordingly.

`trkNetClose (trkHandle)`

releases the allocated UDP port and thus terminates network data transfer.

`trkNetReadParams (trkHandle, trkCameraParams*)`

checks if a new UDP datagram has arrived on the allocated port. If this is the case then camera parameters are extracted out of this datagram and stored in the specified structure. If no datagram has arrived then the function returns immediately with the error code `trkUnavailable`.

```
trkNetBufReadParams (trkHandle, trkCameraParams*,
                     int numPassed, int numTolerated)
```

works similarly to `trkNetReadParams` and should be used instead of it in cases where data transfer across a network suffers from irregular delay intervals, producing a noticeable jitter in the time intervals between successively received camera parameters.

As described above, each camera parameter structure contains a `counter` value which is incremented by one for each new set of camera parameters. Function `trkNetBufReadParams` ensures that the difference between the counter of the camera parameters that are returned by the function call and the counter of those parameters that were returned by the last successful function call is exactly equal to the value of the function parameter `numPassed`. This is accomplished by utilizing a ring buffer to store camera parameters which are taken from several successive UDP datagrams.

New camera parameters usually arrive either at video field rate or at twice this rate, depending on the tracking hardware and the configuration of the *X–cito camera tracking program*. Therefore, the value of parameter `numPassed` should be chosen such that it reflects the number of video fields that have passed since the last successful `trkNetBufReadParams` function call (usually equal to one for a real–time 3D rendering process), possibly multiplied by two if parameters arrive at twice the video field rate. (Please note that the *X–cito camera tracking program* can always be configured such that it sends camera parameters only at video field rate.)

The drawback of using function `trkNetBufReadParams` instead of `trkNetReadParams` is that is might introduce an additional delay between the creation of camera parameters and their arrival at the process requesting them. In order to avoid a gradual buildup of such a delay, function parameter `numTolerated` is used to specify the maximum number of sets of camera parameters that are stored "in advance" within the ring buffer. Using a value for `numTolerated` that is too large might introduce a large delay, while using a value that is too small might eliminate the effects of using a ring buffer and might therefore produce the same results as using function `trkNetReadParams`. The recommended value for `numTolerated` which should work in most cases is two.

`trkNetReadConstants (trkHandle, trkCameraConstants*)`

checks if new camera constants have arrived in a UDP datagram. Datagrams containing camera constants may be sent anywhere between datagrams containing camera parameters. This function will store the last constants that have been received in the specified structure, or it will return with error code `trkUnavailable` if no camera constants were received since the last call of this function.

`trkNetWriteParams (trkHandle, const trkCameraParams*)`

wraps the camera parameters which are contained in the specified structure into a UDP datagram and sends them to the host and port that were named as parameters in function `trkNetOpenWrite`. This function is only needed by tracking programs like the *X–cito camera tracking program*.

`trkNetWriteConstants (trkHandle, const trkCameraConstants*)`

wraps the camera constants which are contained in the specified structure into a UDP datagram and sends them to the host and port that were named as parameters in function `trkNetOpenWrite`. This function is only needed by tracking programs like the *X–cito camera tracking program*.

7.5 Data Transfer by File

If camera parameters are stored in a file then exactly one set of parameters will be stored for every field of the video signal. In order to be able to match each set of parameters to a camera image, the time code of the first set of parameters is stored in the file as well. This time code is stored as a string in the format `HH:MM:SS:FF:D`, with `HH` denoting hours, `MM` minutes, `SS` seconds, `FF` frames and `D` fields. All values are stored as 2–digit decimals with the exception of `D` which is either `1` or `2`, indicating the first or the second field of the frame.

The storage of camera parameters in a file only makes sense if these parameters were all recorded for the same camera, that is, all camera parameters refer to the same camera constants. Therefore, a set of camera constants is only stored once at the head of a file which contains camera parameters.

The following functions are used to store and retrieve camera parameters within a file:

```
trkFileOpenRead (const char* fileName,
                 char* timeCode,
                 trkCameraConstants*,
                 trkHandle*)
```

opens the file specified by parameter `fileName` for reading. If this file does not exist or cannot be opened then error code `trkFailed` is returned. After opening the file, the function retrieves the time code of the first set of camera parameters as well as the camera constants from the file.

```
trkFileOpenWrite (const char* fileName,
                  const char* timeCode,
                  const trkCameraConstants*,
                  trkHandle*)
```

creates the file specified by parameter `fileName` for writing, hereby erasing a file of that name if one already exists. If the file cannot be created then error code `trkFailed` is returned. After creating the file, the function stores the time code of the first set of camera parameters as well as the camera constants in the file. Function `trkFileOpenWrite` is only needed by tracking programs like the *X–cito camera tracking program*.

```
trkFileClose (trkHandle)
```

closes the file that was opened by `trkFileOpenRead` or `trkFileOpenWrite`.

```
trkFileReadParams (trkHandle, trkCameraParams*)
```

reads camera parameters from the file and stores them in the specified structure. If there are no more camera parameters available because the end of the file is reached then error code `trkUnavailable` is returned.

```
trkFileWriteParams (trkHandle, const trkCameraParams*)
```

writes camera constants which are contained in the specified structure into the file. This function is only needed by tracking programs like the *X–cito camera tracking program*.

This concludes the description of the programming interface. Please feel free to contact Xync if there are any questions left unanswered of if you feel that some things have to be clarified. Any corrections and suggestions are certainly welcomed as well.