MIXED REALITY
SOFTWARE DEVELOPMENT KIT
(MXR TOOLKIT)

January 2004
http://mixedreality.nus.edu.sg

The GNU General Public License (GPL)

TERMS AND CONDITIONS FOR COPYING, DISTRIBUTION AND MODIFICATION

0. This License applies to any program or other work which contains a notice placed by the copyright holder saying it may be distributed under the terms of this General Public License. The "Program", below, refers to any such program or work, and a "work based on the Program" means either the Program or any derivative work under copyright law: that is to say, a work containing the Program or a portion of it, either verbatim or with modifications and/or translated into another language. (Hereinafter, translation is included without limitation in the term "modification".) Each licensee is addressed as "you".

Activities other than copying, distribution and modification are not covered by this License; they are outside its scope. The act of running the Program is not restricted, and the output from the Program is covered only if its contents constitute a work based on the Program (independent of having been made by running the Program). Whether that is true depends on what the Program does.

1. You may copy and distribute verbatim copies of the Program's source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice and disclaimer of warranty; keep intact all the notices that refer to this License and to the absence of any warranty; and give any other recipients of the Program a copy of this License along with the Program.

You may charge a fee for the physical act of transferring a copy, and you may at your option offer warranty protection in exchange for a fee.

2. You may modify your copy or copies of the Program or any portion of it, thus forming a work based on the Program, and copy and distribute such modifications or work under the terms of Section 1 above, provided that you also meet all of these conditions:

a) You must cause the modified files to carry prominent notices stating that you changed the files and the date of any change.

b) You must cause any work that you distribute or publish, that in whole or in part contains or is derived from the Program or any part thereof, to be licensed as a whole at no charge to all third parties under the terms of this License.

c) If the modified program normally reads commands interactively when run, you must cause it, when started running for such interactive use in the most ordinary way, to print or display an announcement including an appropriate copyright notice and a notice that there is no warranty (or else, saying that you provide a warranty) and that users may redistribute the program under these conditions, and telling the user how to view a copy of this License. (Exception: if the Program itself is interactive but does not normally print such an announcement, your work based on the Program is not required to print an announcement.)

These requirements apply to the modified work as a whole. If identifiable sections of that work are not derived from the Program, and can be reasonably considered independent and separate works in themselves, then this License, and its terms, do not apply to those sections when you distribute them as separate works. But when you distribute the same sections as part of a whole which is a work based on the Program, the distribution of the whole must be on the terms of this License, whose permissions for other licensees extend to the entire whole, and thus to each and every part regardless of who wrote it.

Thus, it is not the intent of this section to claim rights or contest your rights to work written entirely by you; rather, the intent is to exercise the right to control the distribution of derivative or collective works based on the Program.

In addition, mere aggregation of another work not based on the Program with the Program (or with a work based on the Program) on a volume of a storage or distribution medium does not bring the other work under the scope of this License.

3. You may copy and distribute the Program (or a work based on it, under Section 2) in object code or executable form under the terms of Sections 1 and 2 above provided that you also do one of the following:

a) Accompany it with the complete corresponding machine-readable source code, which must be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,

b) Accompany it with a written offer, valid for at least three years, to give any third party, for a charge no more than your cost of physically performing source distribution, a complete machine-readable copy of the corresponding source code, to be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,

c) Accompany it with the information you received as to the offer to distribute corresponding source code. (This alternative is allowed only for noncommercial distribution and only if you received the program in object code or executable form with such an offer, in accord with Subsection b above.)

The source code for a work means the preferred form of the work for making modifications to it. For an executable work, complete source code means all the source code for all modules it contains, plus any associated interface definition files, plus the scripts used to control compilation and installation of the executable. However, as a special exception, the source code distributed need not include anything that is normally distributed (in either source or binary form) with the major components (compiler, kernel, and so on) of the operating system on which the executable runs, unless that component itself accompanies the executable.

If distribution of executable or object code is made by offering access to copy from a designated place, then offering equivalent access to copy the source code from the same place counts as distribution of the

source code, even though third parties are not compelled to copy the source along with the object code.

4. You may not copy, modify, sublicense, or distribute the Program except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense or distribute the Program is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

5. You are not required to accept this License, since you have not signed it. However, nothing else grants you permission to modify or distribute the Program or its derivative works. These actions are prohibited by law if you do not accept this License. Therefore, by modifying or distributing the Program (or any work based on the Program), you indicate your acceptance of this License to do so, and all its terms and conditions for copying, distributing or modifying the Program or works based on it.

6. Each time you redistribute the Program (or any work based on the Program), the recipient automatically receives a license from the original licensor to copy, distribute or modify the Program subject to these terms and conditions. You may not impose any further restrictions on the recipients' exercise of the rights granted herein. You are not responsible for enforcing compliance by third parties to this License.

7. If, as a consequence of a court judgment or allegation of patent infringement or for any other reason (not limited to patent issues), conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot distribute so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not distribute the Program at all. For example, if a patent license would not permit royalty-free redistribution of the Program by all those who receive copies directly or indirectly through you, then the only way you could satisfy both it and this License would be to refrain entirely from distribution of the Program.

If any portion of this section is held invalid or unenforceable under any particular circumstance, the balance of the section is intended to apply and the section as a whole is intended to apply in other circumstances.

It is not the purpose of this section to induce you to infringe any patents or other property right claims or to contest validity of any such claims; this section has the sole purpose of protecting the integrity of the free software distribution system, which is implemented by public license practices. Many people have made generous contributions to the wide range of software distributed through that system in reliance on consistent application of that system; it is up to the author/donor to decide if he or she is willing to distribute software through any other system and a licensee cannot impose that choice.

This section is intended to make thoroughly clear what is believed to be a consequence of the rest of this License.

8. If the distribution and/or use of the Program is restricted in certain countries either by patents or by copyrighted interfaces, the original copyright holder who places the Program under this License may add an explicit geographical distribution limitation excluding those countries, so that distribution is permitted only in or among countries not thus excluded. In such case, this License incorporates the limitation as if written in the body of this License.

9. The Free Software Foundation may publish revised and/or new versions of the General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

Each version is given a distinguishing version number. If the Program specifies a version number of this License which applies to it and "any later version", you have the option of following the terms and conditions either of that version or of any later version published by the Free Software Foundation. If the Program does not specify a version number of this License, you may choose any version ever published by the Free Software Foundation.

10. If you wish to incorporate parts of the Program into other free programs whose distribution conditions are different, write to the author to ask for permission. For software which is copyrighted by the Free Software Foundation, write to the Free Software Foundation; we sometimes make exceptions for this. Our decision will be guided by the two goals of preserving the free status of all derivatives of our free software and of promoting the sharing and reuse of software generally.

NO WARRANTY

11. BECAUSE THE PROGRAM IS LICENSED FREE OF CHARGE, THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE PROGRAM "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

12. IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MAY MODIFY AND/OR REDISTRIBUTE THE PROGRAM AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE PROGRAM (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER PROGRAMS), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

END OF TERMS AND CONDITIONS

I

USER'S GUIDE

MXR SOFTWARE DEVELOPMENT KIT (MXRTOOLKIT)

The MXR Software Development Kit (or MXRToolkit) consists of a library of routines to help with all aspects of building mixed reality applications. The philosophy of the SDK is to keep the interface extremely simple. All of the code is called using 'c' style function calls and 'c' style structures so that an understanding of C++ syntax is not required.

We anticipate two major types of users:

- Users primarily concerned with developing mixed reality applications – the SDK contains a large number of routines to make camera tracking and model rendering very easy, so the developers can concentrate on the application and content rather than low-level programming.
- Users primarily concerned with developing new low-level tracking solutions using computer vision techniques – the SDK provides access to many of the estimation, maths, optimization, networking and geometry routines that underlie the higher level functions in the library. These will prove invaluable to AR developers.

The MXRToolkit is under constant revision and development and we encourage all of our users to send feedback about the product, especially concerning which features they would like to see included in the next revision.

INTRODUCTION TO MIXED REALITY

Mixed reality refers to the combination of computer graphics and real-world objects. This encompasses both augmented reality, which involves placing computer graphics objects into the real world and augmented virtuality, which involves placing real-world objects into virtual environments. If we consider a continuum of experience with the real world at one end and virtual reality at the other, mixed reality forms the intermediate space.

A wide range of mixed reality applications have been proposed. These include simple annotation of the world, where text labels are superimposed upon the viewer's scene providing information about the environment. An example would be a navigation application where textural information is superimposed telling the user which direction to move to reach his target. More sophisticated applications involve placing realistic three-dimensional objects into the real world so that they are indistinguishable from real parts of the scene. One example might be entertainment applications in which 3-dimensional cartoon characters could be drawn into the real world. Other proposed applications include education, visualization of complex data and medical visualization. At the most realistic end of the spectrum we might aim to capture a live three-dimensional person at a remote location and render him into the user's world so realistically that the user could not tell whether he was really there or not.

This toolkit is primarily concerned with one particular type of application, which is known as "video see-through" augmented reality. In this paradigm the user wears a head-mounted display with a camera attached to the front. The data from the camera is passed to the computer, modified and displayed in the head-mounted display in real-time.

In order to realistically integrate objects into the world, we must determine the exact position and orientation of the camera relative to the objects for each frame in the video stream. In order to do this, we must "track" features in the real world. The easiest way to do this is to place known objects into the scene. These take the form of two-dimensional fiducial markers.

There are two interesting aspects to this type of mixed reality which are not immediately obvious:

- *collaborative reality* : mixed reality allows users to interact with graphical objects while still allowing users to see the real world. This allows several users to naturally collaborate with a computer generated objects as if they were real world items. This provides a substrate for human-human interaction that is far more powerful than a number of people crowding around a computer screen.
- *Tangible interaction:* The tracking routines in this library allow computer graphical objects to be attached to real-world physical objects. By manipulating the position of the real object, we can correspondingly manipulate the position of the virtual object. This is known as "tangible interaction". It provides an extremely natural interface for manipulating three-dimensional objects which does not need to be learnt by the user.

COMPONENTS OF MIXED REALITY

The aim of this section is to discuss the components of a mixed reality system. This is done with reference to the components of the MXRToolkit. Even readers familiar with mixed reality applications should read this section to gain familiarity with the terminology used in the SDK. Figure [1] illustrates the major components of a typical mixed reality system. The viewer sees the real world through a head-mounted display. The head mounted display has a camera (1) mounted to the front. The computer captures (2) a video image (3) from the camera and analyses it. When the computer identifies a marker (4) in the scene, it estimates the exact transformation (5) between the marker and the camera. This transformation comprises a translation and rotation. We then render the virtual media objects (6) into a 3-dimensional frustum (7) so that they appear to be standing on the marker. Both the original video image and the graphics object are then presented to the user in the head-mounted display.

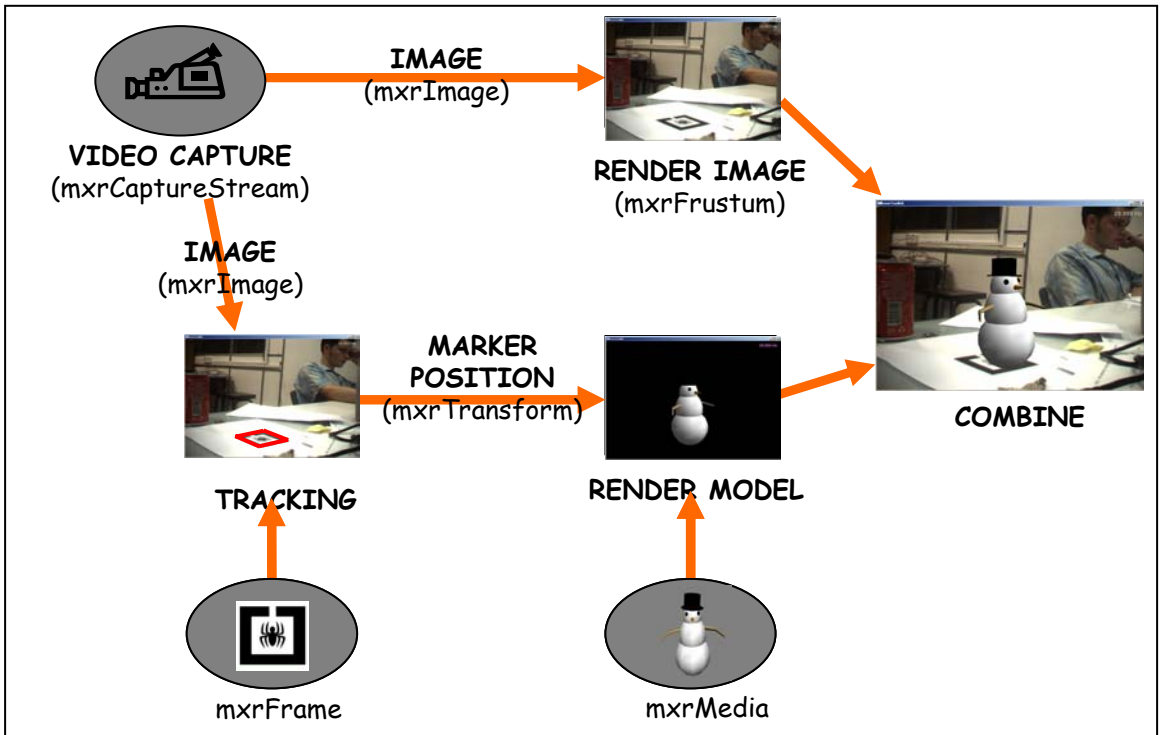


Figure [1] Components of Mixed Reality System. The image is captured and displayed in the background (top path). The image is also analysed – the machine looks for a 2D marker in the scene (mxrFrame). It then renders a model into the so that it appears to sit on the marker.

Each of the seven components listed above and detailed in Figure [1] corresponds to an important part of the MXRToolkit. Each has a number of routines and a data structure associated with them. The names of these data structures are:

Structure Name	Purpose
mxrCamera	Describes real-world camera model parameters e.g. field of view
mxrCaptureStream	Holds information about the video capture source – height and width
mxrImage	Holds a single image from the capture stream
mxrFrame	Describes the marker or object in the world to be tracked
mxrTransform	Describes relative position and orientation of marker and camera
mxrMedia	Holds information about 3d models / media clips
mxrFrustum	Describes rendering volume that matches the real camera parameters

In the next section we discuss each of the parts of the mixed reality in turn and introduce the data structures associated with this part of the process. A complete list of all data structures and software routines used by MXRToolkit is presented in Part II.

1. Camera Models – mxrCamera

The starting point of the mixed reality system is the camera which is generally mounted to the front of a head mounted display. Ultimately, we wish to use the images from the camera to make measurements of object positions in the world, and then to introduce new items into the scene realistically. In order to make such measurements, we must know something about the camera which produced the images. In particular, we need to know exactly where in the image (in pixels) any given point in three-dimensional space will project to. It transpires that this relationship can be described using a simple model with nine parameters. These parameters are kept inside the mxrCamera structure, which is a simple 'c' structure defined as:

```
struct mxrCamera{
        double          fx;                             // x focal length
        double          fy;                             // y focal length
        double          ox;                             // x offset in pixels
        double          oy;                             // y offset in pixels
        double          skew;                           // skew parameter
        double          r2;                             // 2nd order radial distortion
        double          r4;                             // 4th order ratial distortion
        double          xy1;                            // tangential distortion
        double          xy2;                            // tangential distortion #2
        …
};
```

The ellipsis (…) denotes the fact that there are also some more members of this structure which we do not discuss here for simplicity. The complete description of the camera model including the interpretation of all of these parameters is provided at the start of the Camera Library part of the Reference Section. However, for those of you who are wondering what type of parameters might describe a camera, simple examples might be the horizontal field of view in degrees, or the aspect ratio.

The key message of this section is: *In order to do mixed reality, we must know the values of these nine parameters for our camera*. The process of finding these cameras is termed "camera calibration," and is carried out prior to starting the mixed reality application. A separate Calibration Tool is provided to help the user easily determine these values and save them to a file that can be loaded in at run-time.

2. Camera Capture - mxrCaptureStream

The second part of the mixed reality process concerns the capturing of images from the camera attached to the computer. The 'c' structure which describes the capturing process is termed mxrCaptureStream, and has the following members:

```
struct mxrCaptureStream{
        mxrCamera               cam;                            // camera structure
        int                     imX;                            // width of stream in pixels
        int                     imY;                            // height of stream in pixels
        mxrImFormat             format;                         // pixel format of stream
        float                   frameRate;                      // nominal frame rate in Hz
        …
};
```

The first field simply reflects the camera parameters describing the capturing device as described above. The remaining parameters describe the height and width of the captured images, and the pixel format of the captured images (e.g. RGB, BGR). When capturing starts these fields are initialized. If multiple mxrCameraStream structures are declared an initialized, one can capture frames from multiple cameras simultaneously. One circumstance in which this might be useful would be for stereoscopic mixed reality applications where two cameras capture the world from slightly different viewpoints and relay their images to the two eyes of the user.

3. Images - mxrImage

The result of the capturing process is an image which is stored in the mxrImage structure, which contains the pixel color data, and some basic information about how to interpret it. The mxrImage structrure is defined as:

```
struct mxrImage{
        int             x;                                      // width of image in pixels
        int             y;                                      // height of image in pixels
        mxrImFormat     format;                                 // image format
        mxrCamera       cam;                                    // camera data
        unsigned char*  ubData;                                 // pointer to image data
        …
};
```

The first three fields refer to the height, width, and format of the stream as described for the capture device. The fourth field contains information about the camera that was used to capture the image. The field ubData is a pointer to the actual data of the image, which is stored in a "row-first" fashion.

All operations on images in the mxrSDK use this basic structure to contain information about the image.

4. Tracking Markers - mxrFrame

It is possible to track individual markers. However, there are many cases where we know a-priori that several markers are physically connected together (i.e. fixed to one piece of card). If this is the case, then we only need to establish the position of one of the markers to identify where the object is. In a sense, what we are trying to estimate is the transformation from the "frame of reference" of the to the marker. Hence, we denote the tracking object "mxrFrame". Roughly speaking, a frame is a marker or collection of markers or other tracking objects that are known to be rigidly connected together. After initializing the frame with data about the patterns that are being searched for and their relative positions, we search for the frame in the image. On return, the mxrFrame structure provides information about the success of this search via the following elements:

```
struct mxrFrame{
        bool                    foundFlag;              // was frame found in the image?
        mxrTransform            T;                      // position and orientation of frame
        …
};
```

5. Transformations – mxrTransform

The result of a successful tracking operation is a transformation matrix – this encompasses all the information about the position and orientation of the object relative to the marker. In precise terms it is the Euclidean transformation matrix which transforms points in the frame of reference of the tracking frame, to points in the frame of reference in the camera. The full structure is defined as:

```
struct mxrTransform{
        mxrFloat        r11;
        mxrFloat        r12;
        mxrFloat        r13;
        mxrFloat        r21;
        mxrFloat        r22;
        mxrFloat        r23;
        mxrFloat        r31;
        mxrFloat        r32;
        mxrFloat        r33;
        mxrFloat        tx;
        mxrFloat        ty;
        mxrFloat        tz;
};
```

The first nine elements form a 3x3 rotation matrix and describe the orientation of the object. The last three-elements represent the position of the object relative to the camera. Hence, as the "tz" component gets larger, the object moves further from the camera.

6. 3D Media Objects – mxrMedia

The final two components of the system concern the virtual objects and how to draw them. The mxrMedia structure holds information about the three-dimensional model – it includes data concerning the animation speed and status of the object, the position of the model

relative to the centre of the marker. The mxrMedia structure contains the following elements.

```
struct mxrMedia {
       float                    speed;
       mxrLoopStatus            loopStatus;
       mxrMediaHandle           handle;
       float                    startFrame;
       float                    currentFrame;
       float                    endFrame;
       int                      numOfFrames;
       mxrTransform             T;
       …
};
```

The most important components are the "handle" field which is a pointer to the data that allows the three-dimensional model to be drawn, and the transformation field which provides information about the current position and orientation of the object in the current co-ordinate frame

7. Rendering Volume - mxrFrustum

If one considers any three-dimensional application, there is a "virtual camera" which views the 3-D world. This has many of the properties shared by real world cameras – for example, it has a constant field of view. In order to display virtual objects so that they appear to be superimposed upon objects in the real world, we must ensure that the parameters of this "virtual camera" exactly match the parameters of the physical camera attached to the front of the HMD.

In computer graphics, the properties of the "virtual camera" are encompassed by the viewing frustum. The viewing frustrum is the volume of space in which graphical objects are drawn and mapped to the screen of three-dimensional space. It usually takes the form of a truncated pyramid, where the tip of the pyramid would lie at the position of the camera. The mxrFrustum structure contains the following elements:

```
struct mxrFrustum{
       mxrMatrix          projection;         // projection matrix
       mxrMatrix          invProjection;      // inverse of projection matrix
       mxrPoint3D         vertexCoords[MXR_TEX_RES][MXR_TEX_RES];
       mxrPoint2D         texCoords[MXR_TEX_RES][MXR_TEX_RES];
       …
}
```

The fields projection and invProjection contain 4x4 matrices which describe the shape of the frustum. The vertexCoords and texCoords fields contain data that allows the incoming video stream to be efficiently rendered.

MXRTOOLKIT INSTALLATION INSTRUCTIONS IN MICROSOFT VISUAL C++

Before compiling and running these projects, make sure you have installed 2 following software in your computer:

- DirectX 9.0 SDK. In MS Visual C++, menu Tool->Options, tab Directories, add the paths of the include files and library files of DirectX 9.0 SDK to the path lists (for example "C:\DXSDK\Include" and "C:\DXSDK\Lib", if "C:\DXSDK" is where you install DirectX SDK) and make sure that they are on top of the lists
- PGRFlyCapture: driver for Firewire/Dragonfly camera of Point Grey Research Inc.

1. Unzip "MXRToolkit.zip" to a folder, for example: "C:\"
2. In folder "C:\MXRToolkit", open MXRToolkit workspace: MXRToolkit.dsw
3. There are 5 projects in this workspace:
   - mxrSDKDLL, mxrSDKMediaDLL and simpleTrack. The first two projects are used to build dll files of MXRToolkit (mxrSDK.dll and mxrMedia.dll), and simpleTrack is a small example using these 2 dlls.
   - mxrCalibration: source code of Calibration Tool. This supporting tool of MXRToolkit is used for Camera Calibration (see "Calibration Tool Documentation NUS.doc")
   - winOGL: source code of Alignment Tool. This supporting tool of MXRToolkit is used to set intuitively the position and size of the virtual object so that it can appear rightly on the markers (see "Alignment Tool Documentation NUS.doc")

4. In menu Tools->Options, click to Directories tab. Add "C:\MXRToolkit\Header Files" as the directory for include files. Also add "C:\MXRToolkit\Lib Files" as the directory for library files.
5. In menu Project->Set Active Project, make sure the project "simpleTrack" is set active.
6. Compile mxrSDKDLL, mxrMedia, and simpleTrack projects, and run simpleTrack program.

RUNNING CALIBRATION AND ALIGNMENT TOOL IN MICROSOFT VISUAL C++

1. Open MXRToolkit workspace: MXRToolkit.dsw, as shown above (from step 1 to 4).
2. Make sure two projects mxrSDKDLL and mxrMedia are successfully built.
3. In menu Project->Set Active Project, set the project corresponding with the tool you want to run as active project (mxrCalibration project for the Calibration Tool, and winOGL project for the Alignment Tool).
4. Compile and run the active project.

RUNNING MXRTOOLKIT SAMPLE PROJECTS IN MICROSOFT VISUAL C++

Before running these samples, make sure that MXRToolkit source code has been compiled successfully (following steps in the previous part).

1. Unzip "Example Projects.zip" to a folder, for example: "C:\". Also assume that the folder containing source code of MXRToolkit is "C:\MXRToolkit".

2. After "Example Projects.zip" is unzipped, in folder "C:\Example Projects" there are 7 sub-folders:

- *Track1, Track2, Track Media, Track Win* : 4 example projects.
- *Bin* : contains all executable files. After each project is compiled, its executable file will be copied to this folder
- *Frames* : contains marker patterns
- *Media Objects* : contains media object files

3. After MXRToolkit source code is compiled, copy all dll files in the directory "C:\MXRToolkit\Bin Debug" to "C:\Example Projects\Bin"

4. To compile each example project, in menu Tools->Options of VC++ 6.0 IDE, click to Directories tab. Add "C:\MXRToolkit\Header Files" as the directory for include files. Also add "C:\MXRToolkit\Lib Files" as the directory for library files.

5. To run the project, in menu Project->Settings, click to Debug tab. Specify correctly the executable file for debug session: "..\Bin\xxxx.exe", in which "xxxx.exe" is the project's executable file name.

OTHER MXRTOOLKIT PROJECT SETTINGS

We provide several example programs. If you wish to set up your own project, please make sure that you set the following directories. Here, we assume that directory "C:\MXRToolkit" contains source code of MXRToolkit, and this source code is compiled successfully.

Setting up a new project in Microsoft Visual C++

1. In Tools->Options add a path in the include files tab to the "C:\MXRToolkit \Header Files" directory.
2. In Tools->Options add a path in the library files tab to the "C:\MXRToolkit\Lib Files" directory
3. In the Project/Settings section under Link, in the "General" settings, add mxrSDK.lib, mxrMedia.lib and openGL32.lib under the project/library modules section.
4. At the top of your program make sure to add the declaration "#include <mxrSDK.h>". and "#include <mxrMedia.h>"
5. Make sure to copy all dll files in the directory "C:\MXRToolkit\Bin Debug" (including "mxrSDK.dll", "mxrMedia.dll" and other 3 necessary dlls) to the folder containing your executable files. You can simply copy these 5 dlls to "C:\WINDOWS\system32", in which "C:\WINDOWS" is the folder of your operating system.

Tutorial #1: Tracking Markers Using MXRToolkit

In this chapter we demonstrate an example of camera capture and fiducial marker tracking using the Mixed Reality Software Development Kit (MXRToolkit). We first present the full code and subsequently consider each part in turn. This code is included with the software development kit as the Visual C++ project "Track 1.exe". This code is included in the package "Example Projects.zip".

Before running the software, print out the test pattern "TrackSpider.pdf" from the C:/Example Projects/Frames directory (assuming that "C:/Example Projects/" is where you unzip "Example Projects.zip"), and mount it to a flat, rigid piece of card. In order to run the software, open the Visual C++ project, and select Build->Execute Track1.exe (remember to follow the instructions for project settings in the previous part). If things are running correctly, a teapot should appear when the camera is pointed at the test pattern, and appear to be rigidly attached to the marker pattern. If you have trouble running the program, then please consult the end of this section for help.

The complete program listing is as follows

```
#include <mxrSDK.h>
#include <GL/gl.h>

// GLOBAL DEFINITIONS

#define NEAR_PLANE            100
#define FAR_PLANE             10000
#define WINDOW_X              800
#define WINDOW_Y              600
#define WINDOW_XPOSN          0
#define WINDOW_YPOSN          0
#define FULL_SCREEN           0

// FUNCTION DEFINITIONS

void mxrMain(void);
void mxrKeyboard(unsigned char,int,int);

// GLOBAL DECLARATIONS

mxrCaptureStream      cap;                  // video capture object
mxrImage              videoImage;           // image to receive data from video capture
mxrFrustum            frustum;              // graphical display object
mxrFrame              frame;                // tracking frame (stores marker info)

// PROGRAM START

void main(int argc, char **argv){
        // INITIALISE CAPTURE STREAM
        mxrCaptureInit(&cap, NULL, MXR_WEBCAM, 0,NULL);
        // INITIALIZE DISPLAY FRUSTUM
        mxrFrustumInit(&frustum,&cap,NEAR_PLANE,FAR_PLANE);
        // INITIALIZE OPEN GL AND SET UP WINDOW
        mxrGLSimpleWindow(FULL_SCREEN,WINDOW_X,WINDOW_Y,WINDOW_XPOSN,WINDOW_YPOSN);
        // READ IN INFORMATION ABOUT OBJECT TO BE TRACKED
        mxrFrameRead(&frame,"C:/Example Projects/Frames/TrackSpider.frame");
        // START MAIN RENDERING ROUTINE
        mxrGLStart(mxrMain, mxrKeyboard, mxrGLReshapeDefault);
}
```

```
// MAIN RENDERING ROUTINE

void mxrMain(void){
	// CLEAR SCREEN
	glClear(GL_COLOR_BUFFER_BIT|GL_DEPTH_BUFFER_BIT);
	// RETRIEVE VIDEO IMAGE FROM CAPTURE STREAM
	mxrCaptureImage(&videoImage, &cap);
	// FIND MARKER IN SCENE
	mxrFrameTrack(&frame,&videoImage,1);
	// DRAW VIDEO
	mxrGLDrawVideo(&frustum,&videoImage);
	// IF FOUND DRAW TEAPOT ON TOP OF MARKER
	if (frame.foundFlag){
		mxrGLProjectionLoad(&frustum);
		mxrGLModelViewLoad(&frame.T);
		glBegin(GL_QUADS);
			glVertex3f(-20,-20, 0);glVertex3f(-20, 20,0);
			glVertex3f( 20, 20, 0);glVertex3f(20 ,-20,0);
		glEnd();
		mxrGLTeapot(&frame.T,50);
	}
	// DISPLAY TO SCREEN
	mxrGLSwapBuffers();
}

// KEYBOARD EVENT HANDLER
void mxrKeyboard(unsigned char keyStroke, int X, int Y){
	if (keyStroke==0x1b){                    // if Escape key pressed
		mxrCaptureKill(&cap);                // then quit...
		exit(0);
	}
}
```

Step-by-Step Tracking Walkthrough

We now work our way through each part of this program in turn to describe the basic functions of the MXRToolkit.

```
mxrCaptureStream	cap;			// video capture object
mxrImage		videoImage;		// image to receive data from video capture
mxrFrustum		frustum;		// graphical display object
mxrFrame		frame;			// tracking frame (stores marker info)
```

Declarations: The first part of the program consists of a number of global declarations. These involve four of the structures discussed in the previous chapter. In order, these are:

- cap is a capture stream object – this grabs images from the camera - it holds information about the camera parameters, and the height, width and frequency of the video stream that it will produce
- videoImage is the image that is produced by the capture stream on each frame, and will be drawn to the screen.
- Frustum is the viewing volume that the video and 3d objects will be drawn into – it is generated based on the particular parameters of the input camera stream
- Frame will contain details about the particular pattern that we are tracking – that is the pdf file that you printed out and mounted to card

```
void main(int argc, char **argv){
	// INITIALISE CAPTURE STREAM
	mxrCaptureInit(&cap, NULL, MXR_WEBCAM, 0,NULL);
```

Capture Initialization: The main routine is entered and the first job is to initialize the camera. The routine "mxrCaptureInit" starts the video grabbing and fills out the structure "cap". In this case, the chosen source is "MXR_WEBCAM", which means that we wish to grab from a standard Windows video source. By passing the second parameter as NULL, we opt to let the program intelligently guess the parameters of our camera rather than loading them from disk. On return from this routine, the height, width, frequency and format of the video stream will be set in the mxrCaptureStream structure. If the initialization has failed, the values will be set to some defaults, but the capture type field of the structure will be set to MXR_NULL. See the Capture library reference material for more details.

```
// INITIALIZE DISPLAY FRUSTUM
mxrFrustumInit(&frustum,&cap,NEAR_PLANE,FAR_PLANE);
```

Viewing Volume Initialization: In order to draw the video correctly and superimpose 3D objects, we must set up a viewing volume or frustum for the computer graphics. Since the perspective projection of the graphics must match that of objects in the real world, the properties of this viewing volume depend on the properties of the capture stream. Hence, we pass in the initialized capture object. We also pass in two parameters NEAR_PLANE and FAR_PLANE, which are constants and represent the nearest and farthest objects from the camera that will be drawn in mm. This range can be set to extend from 0 to infinity, but making it smaller increases the precision of rendering. In the example program we have opted to render objects from 10 cm to 10 m away.

```
// INITIALIZE OPEN GL AND SET UP WINDOW
mxrGLSimpleWindow(FULL_SCREEN,WINDOW_X, WINDOW_Y, WINDOW_XPOSN, WINDOW_YPOSN);
```

Opening a Graphics Window: There are two main options for rendering 3D graphics under windows – these are the OpenGL and Direct 3D API's. All of the examples in this document are demonstrated using OpenGL as this is simpler to learn and understand. There is also a supporting library called GLUT (GL UTility library), which provides very easy to understand routines which open and manipulate windows. This example makes use of both of these libraries. The routine mxrGLSimpleWindow opens a window on the screen at a certain position and with a certain size, and passes some flags which are used to initialize OpenGL. After this routine is called OpenGL commands can be used. The full source code for this routine is given with the description in the reference section.

Advanced users may wish to use Windows routines to initialize the display, or even present their mixed reality application in an MFC context. We provide working examples of both of these options in the projects "Track1Windows" and "Track1MFC". It is also possible that users may wish to use the Direct X API instead of OpenGL. Please consust the frequently asked questions list for how to make this switch.

```
// READ IN INFORMATION ABOUT OBJECT TO BE TRACKED
mxrFrameRead(&frame,"C:/Example Projects/Shared Data/MarkerSpider.frame");
```

Reading in Tracking Information: In order to track an object, the computer must know which object to track. This information is handled using the mxrFrame structure. This routine loads in stored information concerning the pattern from a file.

```
        // START MAIN RENDERING ROUTINE
        mxrGLStart(mxrMain, mxrKeyboard, mxrGLReshapeDefault);
```

Start Main Rendering Loop: This call starts the main rendering loop. The names of several routines are passed. The routine "mxrMain" is called every time the display needs to be refreshed (i.e. all the time!) and contains the bulk of our code. The routine mxrKeyboard is called when a key is pressed. The last routine is called when the window is resized. After this call, the program goes into an infinite loop, calling these routines when necessary and never returning.

```
// MAIN RENDERING ROUTINE

void mxrMain(void){
        // CLEAR SCREEN
        glClear(GL_COLOR_BUFFER_BIT|GL_DEPTH_BUFFER_BIT);
```

Start of Main Rendering Routine/ Clear the Screen: This routine is the main loop that is entered every frame. The first thing we do is clear the screen, using the OpenGL command glClear.

```
        // RETRIEVE VIDEO IMAGE FROM CAPTURE STREAM
        mxrCaptureImage(&videoImage, &cap);
```

Capturing the Video Image: Now we grab an image (videoImage) from the capture stream (cap). On return from this routine, the image structure will be filled out with the size and format of the image and the camera parameters associated with the capture stream as well as the image data.

```
        // FIND MARKER IN SCENE
        mxrFrameTrack(&frame,&videoImage,1);
```

Tracking the marker position: The tracking routine takes the image from the capture stream and attempts to find the position of the marker within the image. If the tracking is successful, then the transformation matrix relating the camera and marker position will be placed in the frame structure, and the flag "frame.foundFlag" will be set to true.

```
        // DRAW VIDEO
        mxrGLDrawVideo(&frustum,&videoImage);
```

Drawing the video: We now draw the video into the viewing frustum. It is pasted on the back wall of the viewing frustum so that all virtual objects will appear in front of it. The video is specially warped to compensate for any distortion introduced by the lens, and identified by the calibration procedure.

```
        // DRAW SQUARE ON TOP OF MARKER
        mxrGLProjectionLoad(&frustum);
        mxrGLModelViewLoad(&frame.T);
        glBegin(GL_QUADS);
```

```
            glVertex3f(-20,-20,0); glVertex3f(-20,20,0);
            glVertex3f(20,20,0)   ; glVertex3f(20,-20,0);
        glEnd();
        // DRAW TESPOT ON TOP OF MARKER
        mxrGLTeapot(&frame.T,50);
```

Drawing Simple Objects on the Marker: The routines above set the viewing frustum to our pre-calculated values, and the position to render the object to be the position returned by the tracking routine. We then render a simple square on top of the marker. The final routine renders a three-dimensional teapot onto the marker using OpenGL.

```
        // DISPLAY TO SCREEN
        mxrGLSwapBuffers();
```

Swapping Video Buffers: The final command in the main routine sends what we have drawn to the screen for the user to see.

This example program gives a simple idea of the structure of most mixed reality applications. There is an initialization section in which the graphics and tracking are initialized, followed by a main loop, which is called indefinitely. This main loop performs several tasks, of which the most important are:

- Clear the screen
- Grab video from the video source
- Track the objects
- Draw the video from the video source
- Draw any 3D objects
- Display to the user.

Troubleshooting Track 1.exe

- Video display does not show
    - make sure the call to mxrCaptureInit is appropriate for your camera – MXR_WEBCAM for most Windows Video sources, MXR_FIREFLY/ MXR DRAGONFLY for Point Gray Firewire cameras
    - Ensure that the camera is connected properly and is working with other programs
    - Make sure that the camera settings are appropriate – we recommend 24 bit RGB 640x480 images
- Video display shows, but tracking is intermittent or absent (teapot appears and disappears). The probable reason for this is that either
    - Lighting is peculiar and the image appears as either very dark or very bright
    - The camera needs to be calibrated – if this is the case run the camera calibration utility, save the data to disk and load it in using the mxrCaptureInit command

- Compiler errors concerning GLUT – if you receive compiler errors complaining about the GLUT library this may be because you have a different version of GLUT than the one the library was compiled with. Please ensure that the path to the version provided with the MXRToolkit is at the top of the path list (found in Tools/Options/Directories).

Tutorial #2 – Tracking Multiple Objects

In the second tutorial, we consider tracking several different objects simultaneously and displaying different content on each. In order to track several objects, we simply declare several mxrFrame structures adjacently in memory. The tracking routine then attempts to find a transformation matrix for each. In this case, one of the objects will be the teapot and one will be a solid cube. These are both simple pre-defined test objects. A separate tutorial considers how to load your own models and display them in mixed reality.

To run the program start the Visual C++ workspace "Track2" and compile and execute the program. You will need to print out the patterns "TrackSpider.pdf" and "Track4.pdf" from the C:/Example Projects/Frames directory (again, assuming that "C:/Example Projects/" is where you unzip the file "Example Projects.zip") and mount them to separate rigid flat card bases. When the program is run you should see different virtual object appear on each pattern – the two objects are tracked independently.

```
#include <mxrSDK.h>
#include <GL/gl.h >

// GLOBAL DEFINITIONS

#define NEAR_PLANE            100
#define FAR_PLANE             10000
#define WINDOW_X              800
#define WINDOW_Y              600
#define WINDOW_XPOSN          0
#define WINDOW_YPOSN          0
#define FULL_SCREEN           0

// FUNCTION DEFINITIONS

void mxrMain(void);
void mxrKeyboard(unsigned char,int,int);

// GLOBAL DECLARATIONS

mxrCaptureStream      cap;                  // video capture object
mxrImage              videoImage;           // image to receive data from video capture
mxrFrustum            frustum;              // graphical display object
mxrFrame              frame[2];             // tracking frame (stores marker info)

// PROGRAM START

void main(int argc, char **argv){
        // INITIALISE CAPTURE STREAM
        mxrCaptureInit(&cap, NULL, MXR_WEBCAM, 0,NULL);
        // INITIALIZE DISPLAY FRUSTUM
        mxrFrustumInit(&frustum,&cap,NEAR_PLANE,FAR_PLANE);
        // INITIALIZE OPEN GL AND SET UP WINDOW
        mxrGLSimpleWindow(FULL_SCREEN,WINDOW_X,WINDOW_Y,WINDOW_XPOSN, WINDOW_YPOSN);
        // READ IN INFORMATION ABOUT OBJECT TO BE TRACKED
        mxrFrameRead(&frame[0],
                        "C:/Example Projects/Frames/TrackSpider.frame");
        mxrFrameRead(&frame[1],
                        "C:/Example Projects/Frames/TrackFour.frame");
        // START MAIN RENDERING ROUTINE
        mxrGLStart(mxrMain, mxrKeyboard, mxrGLReshapeDefault);
}

// MAIN RENDERING ROUTINE
```

```
void mxrMain(void){
	// CLEAR SCREEN
	glClear(GL_COLOR_BUFFER_BIT|GL_DEPTH_BUFFER_BIT);
	// RETRIEVE VIDEO IMAGE FROM CAPTURE STREAM
	mxrCaptureImage(&videoImage, &cap);
	// FIND MARKER PATTENRS IN SCENE
	mxrFrameTrack(frame,&videoImage,2);
	// DRAW VIDEO
	mxrGLDrawVideo(&frustum,&videoImage);
	// IF FOUND DRAW SQUARE AND TEAPOT ON TOP OF MARKER

	if (frame[0].foundFlag){
		mxrGLTeapot(&frame[0].T,50);
	}
	if (frame[1].foundFlag){
		mxrGLSolidCube (&frame[1].T,50);
	}

	// DISPLAY TO SCREEN
	mxrGLSwapBuffers();
}

// KEYBOARD EVENT HANDLER
void mxrKeyboard(unsigned char keyStroke, int X, int Y){
	if (keyStroke==0x1b){				// if Escape key pressed
		mxrCaptureKill(&cap);			// then quit...
		exit(0);
	}
}
```

Almost all of the program is exactly the same as in the first example, so we will not consider the details of the video capture etc. There are are several parts which differ.

```
mxrFrame		frame[2];		// tracking frame (stores marker info)
```

Declaration of Frames: We now declare more than one frame to track.

```
	// READ IN INFORMATION ABOUT OBJECT TO BE TRACKED
	mxrFrameRead(&frame[0],"C:/Example Projects/Frames/TrackSpider.frame");
	mxrFrameRead(&frame[1],"C:/Example Projects/Frames/TrackFour.frame");
```

Reading in Tracking Information: Accordingly, we must read in information about the two different sets of markers.

```
	// FIND MARKER PATTENRS IN SCENE
	mxrFrameTrack(frame,&videoImage,2);
```

Tracking several frames: We now inform the tracking routine that we wish to recover more than one frame, and pass a pointer to the first of these frames in memory. This method assumes that the frame structures are stored adjacently.

```
	if (frame[0].foundFlag){
		mxrGLTeapot(&frame[0].T,50);
	}
	if (frame[1].foundFlag){
		mxrGLSolidCube (&frame[1].T,50);
	}
```

Rendering Objects: We now render each object if the respective card was successfully tracked.

Notice that one of the patterns ("track4.pdf") contained several spatially separated patterns. In this mode, all four patterns are tracked and the estimate of the card position is based on an average of all four pattern positions – this lends some robustness to the tracking – if the users hand obscures one of the markers the others may still be used to calculate the card.

Tutorial #3 – Create your own patterns to track

In the previous two tutorials, we have loaded in patterns from files using the routine "mxrFrameRead." What if you wish to make your own pattern or combination of patterns? There are actually several ways to do this, but the simplest way is to design your own bitmap file containing the pattern and print it out. MXR Toolkit can read in the bitmap file and establish the marker information and create a frame structure.

To create your own tracking object, follow these steps:

- Open "TrackBlank.jpg" from the C:/Example Projects/Frames directory in an image editor like Adobe Photoshop (assuming that "C:/ Example Projects/" is where you unzip the file "Example Projects.zip")
- Draw something inside the marker outline.
- Save the pattern as "TrackMine.jpg" or similar.
- Print the patten out – note down the resolution (dots per inch) that you print the pattern out at

Now use the project "Track1", but replace the line:

```
// READ IN INFORMATION ABOUT OBJECT TO BE TRACKED
mxrFrameRead(&frame,"C:/Example Projects/Frames/MarkerSpider.frame");
```

with:

```
mxrFrameCreateFromImage(&frame,"C:/Example Projects/Frames/TrackMine.jpg",dpi);
```

Where dpi is the resolution at which the image was printed out. The routine mxrFrameCreateFromImage will analyse the pattern and fill out a frame structure accordingly.

This routine will work with as many different patterns as you want in any planar configuration – the pattens can be multiple sizes – the only constraints are:

- The patterns must be square
- They must have a unique spatial pattern inside them
- They must all be in the same plane (i.e. printed out on a flat sheet of card).

Tutorial #4 - Media Files and Animation

The following example loads in one VRML media file "snoman.wrl". The important routines used in the example are shown in the boxes below.

```
#include "mxrSDK.h"
#include "mxrMedia.h"
#include "glut.h"

// GLOBAL DEFINITIONS

#define NEAR_PLANE            100
#define FAR_PLANE             10000
#define WINDOW_X              800
#define WINDOW_Y              600
#define WINDOW_XPOSN          0
#define WINDOW_YPOSN          0
#define FULL_SCREEN           0

// FUNCTION DEFINITIONS

void mxrMain(void);
void mxrKeyboard(unsigned char,int,int);

// GLOBAL DECLARATIONS

mxrCaptureStream      cap;                  // video capture object
mxrImage              videoImage;           // image to receive data from video capture
mxrFrustum            frustum;              // graphical display object
mxrFrame              frame;                // tracking frame (stores marker info)
mxrMedia              mediaObj;

// PROGRAM START

void main(int argc, char **argv){
        // INITIALISE CAPTURE STREAM
        mxrCaptureInit(&cap, NULL, MXR_WEBCAM, 0,NULL);
        // INITIALIZE DISPLAY FRUSTUM
        mxrFrustumInit(&frustum,&cap,NEAR_PLANE,FAR_PLANE);
        // INITIALIZE OPEN GL AND SET UP WINDOW
        mxrGLSimpleWindow(FULL_SCREEN,WINDOW_X, WINDOW_Y, WINDOW_XPOSN,WINDOW_YPOSN);
        // READ IN INFORMATION ABOUT OBJECT TO BE TRACKED
        mxrFrameRead(&frame,"C:/Example Projects/Frames/TrackSpider.frame");

        // READ IN A VRML FILE TO BE DISPLAYED
        mxrMediaRead(&mediaObj,"C:/Example Projects/Media Objects/snoman.wrl",NULL);

        // START MAIN RENDERING ROUTINE
        mxrGLStart(mxrMain, mxrKeyboard, mxrGLReshapeDefault);
}

// MAIN RENDERING ROUTINE

void mxrMain(void){
        // CLEAR SCREEN
        glClear(GL_COLOR_BUFFER_BIT|GL_DEPTH_BUFFER_BIT);
        // RETRIEVE VIDEO IMAGE FROM CAPTURE STREAM
        mxrCaptureImage(&videoImage, &cap);
        // FIND MARKER IN SCENE
        mxrFrameTrack(&frame,&videoImage,1);
        // DRAW VIDEO
        mxrGLDrawVideo(&frustum,&videoImage);
        // IF FOUND DRAW MEDIA OBJECT ON TOP OF MARKER
        if (frame.foundFlag){
                mxrGLProjectionLoad(&frustum);
                mxrGLModelViewLoad(&frame.T);
                glBegin(GL_QUADS);
                        glVertex3f(-20,-20, 0);glVertex3f(-20, 20,0);
```

```
                glVertex3f( 20, 20, 0);glVertex3f(20 ,-20,0);
        glEnd();

        // DRAW VRML FILE
        mxrMediaRender(&mediaObj,&frame.T);
    }

    // DISPLAY TO SCREEN
    mxrGLSwapBuffers();
}

// KEYBOARD EVENT HANDLER
void mxrKeyboard(unsigned char keyStroke, int X, int Y){
    if (keyStroke==0x1b){                    // if Escape key pressed
        mxrCaptureKill(&cap);                // then quit...
        exit(0);
    }
}
//end
```

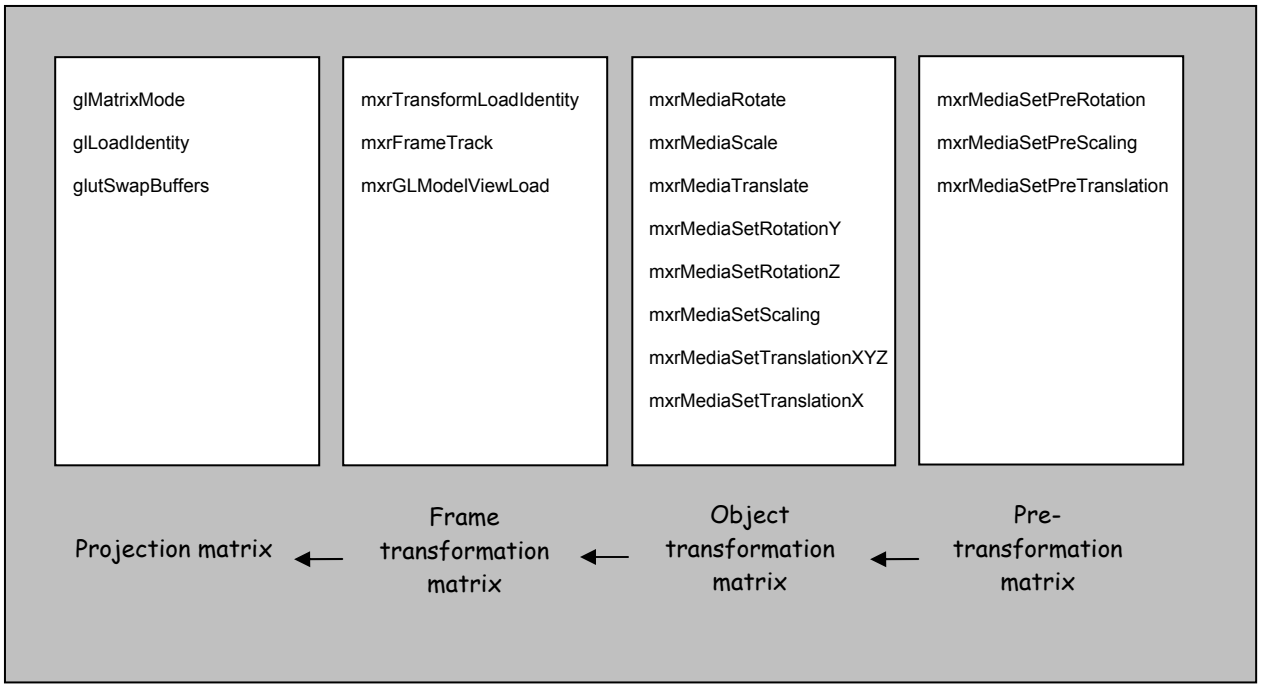
Loading a Media File

```
    // READ IN A VRML FILE TO BE DISPLAYED
(1) mxrMediaRead(&mediaObj,"C:/Example Projects/Media Objects/snoman.wrl",NULL);
```

Routine (1) automatically creates a handle and assigns the VRML file to a media object instance, mediaObj. However, the handle which was created is not visible to the user. The function of the handle will be explained later in "Loading Several Instances of the Same Media File".

```
    // DRAW VRML FILE
(2) mxrMediaRender(&mediaObj,&frame.T);
```

Routine (2) draws the object assigned to mediaObj at a particular point in space as defined by frame transformation matrix frame.T



There is a possibility that the media file that is loaded might not display in the correct orientation. In such cases, we might need to do input some predefined values to set pre-transformation matrix. We can save these predefined values in a "media info" file using the "mxrMediaInfoWrite" routine.

Transforming the Object

We can rotate, translate and change the scale of the object according to our needs by setting the correct values in the object transformation matrix. This can be done by adding in some of the following routines:

```
        // DRAW VRML FILE
(3);    mxrMediaRotate(&mediaObj, 0, 1.57, 0);
(4);    mxrMediaTranslate(&mediaObj, 10, 10, 20);
(5);    mxrMediaScale(&mediaObj, 1.5, 1.5, 1.5);
        mxrMediaRender(&mediaObj,&frame.T);
```

In general, routines (3), (4), (5) are in the form of:

mxrMedia *Transform* (&*media, x, y, z*);

Where *Transform* specifies the type transformation,
media specifies the media object,
x,y,z specify the X-axis, Y-axis and Z-axis

Routine (3) rotates the object by 1.57 radians (90 degrees) with respect to the Y-axis.
Routine (4) translates the object by 10 in the X and Y direction and by 20 in the Z direction
Routine (5) scales the object 1.5 times larger than the current size.

Setting Absolute Translation, Rotation and Scale

We can set the absolute rotational, translational and scaling values but calling the following routines:

```
        // DRAW VRML FILE
(6);    mxrMediaSetRotationZ(&mediaObj, 1.57);
(7);    mxrMediaSetTranslationXYZ(&mediaObj, 10, 0, 0);
(8);    mxrMediaSetTranslationY(&mediaObj,5);
(9);    mxrMediaSetScaling(&mediaObj, 2.0, 1.5, 2.0);
        mxrMediaRender(&mediaObj,&frame.T);
```

The routines (3), (4) and (5) shown above transform the object with respect to the current parameters. Eg. If the object is currently at the position (10, 10, 10) calling "`mxrMediaTranslate(&mediaObj, 10, 0, 0)`" will the new position to (20, 10, 10).

However, rountines (6), (7), (8) and (9) set the absolute value of the transformation and will not take into consideration the current values in the transformation matrix. Eg. If the object is currently at the position (10, 10, 10) calling "`mxrMediaSetTranslationXYZ(&mediaObj, 5, 0, 0)`" will set the new position to (5, 0, 0).

Retrieving Current Transformation

We can retrieve information on rotational, translation and scale from the object transformation matrix by calling routines (10), (11) and (12):

```
        mxrMediaRead(&mediaObj,"snoman.wrl",NULL);
        mxrMediaTranslate(&mediaObj,1.0,1.0,1.0);
        mxrMediaSetRotationZ(&mediaObj, 1.57);
        mxrMediaScale(&mediaObj, 1.5, 1.5, 1.5);
        mxrMediaRender(&mediaObj,&T);

        mxrFloat x, y, z;
(10);   mxrMediaGetTranslation(&x,&y,&z,&mediaObj);
        printf("Translation:\t%f\t%f\t%f\n",x,y,z);

        mxrFloat a, b, c;
(11);   mxrMediaGetRotation(&a,&b,&c,&mediaObj);
        printf("Rotation:\t%f\t%f\t%f\n",a,b,c);

        mxrFloat l, m, n;
(12);   mxrMediaGetScaling(&l,&m,&n,&mediaObj);
        printf("Scale:\t%f\t%f\t%f\n",l,m,n);

        //Output:
        //Translation  1.0     1.0     1.0
        //Rotation     0.0     1.57    0.0
        //Scale        1.5     1.5     1.5
```

In general, routines (3), (4), (5) are in the form of:

`mxrMediaGet` *`TransformationInfo`* `(`*`x, y, z, &media`*`);`

Where *TransformationInfo* specifies the type transformation information retrieved,
x,y,z specify variables which hold parameters of the X-, Y- and Z- components
media specifies the media object,

How is the information stored in the media structure?

Each instance of a media object is represented by a unique media structure. Information on the object's transformation matrix is encapsulated using the following mxrMedia structure:

```
struct mxrMedia {

        .. .. ..

        mxrTransform          tInstance;
        mxrRotVec             r;
        mxrPoint3D            t;
        mxrPoint3D            s;
};
```

`tInstance`, is a 3X4 matrix which holds the current transformation matrix of a media object.
`r`, is a 3-component vector which holds the rotational vector of a media object.
`t` and `s`, are 4-component vectors which hold the translational and scaling vectors of a media object respectively.

Loading several Media Files

In order to load in the more media files, we need to assign the addition media files new media object instances.

```
        mxrMediaRead(&mediaObj1,"C:/Example Projects/Media Objects/snoman.wrl",NULL);
(13);   mxrMediaRead(&mediaObj2,"C:/Example Projects/Media Objects/tris.md2",”rhino.bmp”);
(14);   mxrMediaRead(&mediaObj3,"C:/Example Projects/Media Objects/foot.obj",NULL);
```

Routine (13) and (14) load in a Quake2 (tris.md2) object and a Maya (.obj) object and assign to two different media object instances, mediaObj2 and mediaObj3, respectively. Each media object automatically generates a handle which is not visible to the user. The uses of handles will be further explained in the next section.

Loading Several Instances of the Same Media File

The following example shows you how to display multiple instances of the same object with each object having different orientation in different point in space.

```
(15);   mxrMediaHandle A;

(16);   mxrMediaRead(&A,"C:/Example Projects/Media Objects/snoman.wrl",NULL);
(17);   mxrMediaSet(&mediaObj1, A);
(18);   mxrMediaSet(&mediaObj2, A);
(19);   mxrMediaSet(&mediaObj3, A);
```

In routine (16), instead of passing in a media object, a handle A is passed in. Routine (17), (18) and (19) then assign the handle A to three different instances.

Compare to the previous example of loading in three different files, this example has only one handle, while previously we had three different handles which were automatically generated. Unlike the handles in the former example, this handle has to be declared by the user (shown in routine (15)). This handle is then assigned to as many instances as the user likes. Each instance can be placed in different points in space with different rotational, translational and scale values.

```
        // DRAW THREE VRML INSTANCES
(20);   mxrMediaRotate(&mediaObj1, 0, 1.57, 0);
(21);   mxrMediaScale(&mediaObj1, 1.5, 1.5, 1.5);
        mxrMediaRender(&mediaObj1,&frame1.T);

(22);   mxrMediaTranslate(&mediaObj2, 10, 10, 20);
(23);   mxrMediaScale(&mediaObj2, 0.5, 0.5, 0.5);
        mxrMediaRender(&mediaObj2,&frame2.T);

(24);   mxrMediaRotate(&mediaObj3, 0, 0, 1.57);
(25);   mxrMediaScale(&mediaObj3, 2.0, 2.0, 2.0);
        mxrMediaRender(&mediaObj3,&frame3.T);
```

We can see that frame1.T, frame2.T and frame3.T represent 3 different points in space as each holds a different frame transformation matrix. Routines (20) and (21) set the desired object transformation of mediaObj1 to be displayed in a point in space defined by frame1.T.

Hence, it is easy to see that routines (22) and (23) set the desired object transformation of mediaObj2 to be displayed in a point in space defined by frame2.T while routines (24) and (25) set the desired object transformation of mediaObj3 to be displayed in a point in space defined by frame3.T.

Adding animation to the object

There are many media formats that support animation namely VRML, Quake2, Maya Obj, 3D Studio Max ASE files, etc. Basically, there are several key frames stored in the media file and animation is created by interpolating from one key frame to the next key frame. Fortunately, this is all taken care by the mxrMediaRender routine and the user only need to set the looping parameters.

```
        // READ IN A QUAKE2 FILE THAT SUPPORTS ANIMATION
        mxrMediaHandle A;

        mxrMediaRead(&A,"C:/Example Projects/Media Objects/tris.md2",”rhino.bmp”);

        mxrMediaSet(&mediaObj1,A);
        mxrMediaSet(&mediaObj2,A);
```

In this example, a Quake2 file that supports animation is used.

```
        // DRAW QUAKE2 FILE WITH ANIMATION

(26);   mxrMediaSetLoop(&mediaObj1, MXR_LOOP_FOREVER, 1.1, 9.9);
(27);   mxrMediaSetFrame(&mediaObj1, 5.6);
(28);   mxrMediaSetSpeed(&mediaObj1, 5.0);

(29);   mxrMediaSetLoop(&mediaObj2, MXR_LOOP_ONCE_THROUGH, 20.3, 39.9);
(30);   mxrMediaSetSpeed(&mediaObj2, 1.5);


        mxrMediaRender(&mediaObj1,&frame1.T);
        mxrMediaRender(&mediaObj2,&frame2.T);
```

Function declaration:

```
void mxrMediaSetLoop((mxrMedia *media, mxrLoopStatus loop, int startFrame, int endFrame);
```

The loop status of an animation is encoded using the mxrLoopStatus enumeration:

```
typedef enum mxrLoopStatus{
        MXR_LOOP_FOREVER,
        MXR_LOOP_ONCE_THROUGH,
        MXR_LOOP_ONCE_AND_FREEZE,
        MXR_FINISHED,
};
```

Routine (26) sets the animation for mediaObj1 to be looping forever from frames 1.1 to 9.9. However, routine (27) sets the starting frame of the animation to be frame 5.6. This means that the animation starts running from frame 5.6 to 9.9, then goes back to frame 1.1 and loops forever between frames 1.1 to 9.9. The speed of the animation is set to 5.0 in routine (28).

For the animation in mediaObj2, the object will run from frames 20.3 to 39.9. Since the starting frame is not set, by default, it will starts on the first frame in the loop which is 20.3. This animation is to run only once by passing in MXR_LOOP_ONCE_THROUGH in routine (29). Hence, the object will disappear from the display after it reaches frame 39.9. The speed of the animation is set to 1.5 in routine (30).

The example above shows that it is possible to load in multiple instances for the same object with each instance displaying a different animation in a different point in space.

Checking the Current Loop Status of an Animation

The following example shows a simple method to check the loop status of an animation:

```
        //KEYBOARD EVENT HANDLER
        void mxrKeyboard(unsigned char keyStroke, int X, int Y){
                if (keyStroke==0x1b){
                // if Escape key pressed
                        mxrCaptureKill(&cap);
                        mxrMediaFreeAll();
                // then quit...
                mxrKill();
                }

                switch (keyStroke) {

                case ('c'):

                        mxrLoopStatus l;
(31);                   l=mxrMediaGetLoopStatus(media);

                        switch(l){
                        case MXR_LOOP_FOREVER:
                        printf("Animation is in an indefinte loop\n");
                        break;

                        case MXR_LOOP_ONCE_THROUGH:
                                printf("Animation is playing once only\n");
                                break;

                        case MXR_LOOP_ONCE_AND_FREEZE:
                                printf("Animation is playing once and the last frame will
be held still\n");
                                break;

                        case MXR_LOOP_FINISHED:
                        printf("Animation has finished\n");
                        break;
                        }
                        break;
                }
        }
```

- When 'c' is pressed while running the program, routine (31) is called. This routine checks the current status of the animation and returns an enum type defined in the mxrLoopStatus enumeration. By checking the returned value using a simple switch command, we can print out the current loop status easily.

MXRToolkit FAQ

Q. I don't know OPENGL. Can I use Direct X with the MXRToolkit?

A. Yes you can – the vast majority of the MXRToolkit is ambivalent to your chosen graphics language. However, you will have to implement certain parts yourself. MXRToolkit will calculate all of the parameters needed to set up the viewing frustum. It will help grab the video stream and track the markers. However, the final display must be implemented by the programmer – he must write code to draw the video and any other three-dimensional objects. Essentially, this involves replacing the routines entitled "mxrGL…". To make this job easier, we have provided the code to most of these OpenGL routines so a straight conversion can take place.

Q. I want to present the results in stereo. Can I use two cameras?

A. Yes. Simply declare two capture stream objects, and two frustum objects and capture and render to each in each frame.

Q. How can I change the resolution of the input video?

A. At present the MXRToolkit does not provide the ability to dynamically change the resolution of the video – we recommend using 640x480 video in 24-bit RGB format. The resolution of the video should be changed using the drivers / utilities that were providedwith the camera.

II

REFERENCE

MXRToolkit Libraries

The routines that are provided as part of the software development kit are organized into a number of separate libraries:

- CAPTURE LIBRARY: Deals with camera capture from a variety of different video sources
- DISPLAY LIBRARY: Deals with setting up the viewing frustum for augmented reality display. It also contains OPENGL routines for rendering the video stream and drawing test objects.
- MEDIA LIBRARY: Deals with reading in and drawing model formats in the world. Model formats include a number of 3d mesh formats as well as two-dimensional movies and images.
- TRACKING LIBRARY: Integrates tracking estimates from a number of different sources.
- MATH LIBRARY: contains a number of routines for linear algebra and solution of least squares problems.
- OPTIMIZATION LIBRARY: provides an easy to use Levenberg-Marquardt multidimensional optimization routine, with optional use of finite differences
- GEOMETRY LIBRARY: routines that deal with geometry in three dimensions and transformatoins
- CAMERA LIBRARY: utilities to deal with computer vision camera model and projection from the world into the image
- IMAGE LIBRARY: utilities to deal with loading, and saving image and converting between image formats
- NETWORK LIBRARY: presents a simple interface for client-server networking using WinSock v1.1. Allows networked applications to be built in minutes with no prior expertise.

NAMING CONVENTIONS

- All data types and routines visible to the user start with "mxr".
- All variable names and functions start with a small letter. Second and third words in the names start with a capital letter
    - e.g. `mxrCameraRead`.
- All "`#define`"d constants have capital letters, with words separated by underscores.
    - e.g. `MXR_RGB`
- Function declarations are always such that the destinations come first and the sources come afterwards
    - e.g. `mxrImageRead(mxrImage *im, char *filename)`, reads the information from the second parameter (filename) and into the first (im);
    - but in `mxrImageWrite(char *filename, mxrImage *im)`, the parameter order is reversed as the information flows from the image structure to the filename.

MXRTOOLKIT DATA TYPES:

PROJECTION MATRIX: `mxrProjMat`

The product of an intrinsic camera matrix and the extrinsic matrix is 3x4 projection matrix, mxrProjMat. When you multiply an mxrPoint3D by a mxrProjMat, you get the position in the image of the point, in the absence of radial distortion. i.e. this matrix encompasses the whole linear part of the camera model. The first index of each float is a

```
struct mxrProjMat{
      mxrFloat p11;
      mxrFloat p12;
      mxrFloat p13;
      mxrFloat p14;
      mxrFloat p21;
      mxrFloat p22;
      mxrFloat p23;
      mxrFloat p24;
      mxrFloat p31;
      mxrFloat p32;
      mxrFloat p33;
      mxrFloat p34;
};
```

CAMERA MODEL: `mxrCamera`

The camera structure encompasses all information about the camera model. If you know nothing about computer vision, then all you need to know that is that there are some numbers which describe the properties of the real-world camera, such as focal length and field of view. These must be estimated if we are going to place objects realistically into the world.

For people familiar with computer vision, the first five parameter describe the components of the intrinsic matrix: x focal length, y focal length, x offset, y offset and skew parameters. The next four parameters describe the radial distortion of the camera. They are the second order radial distortion, fourth order radial distortion, and first and second transverse distortion components respectively. The penultimate parameter, T is the transformation matrix from world to camera co-ordinates, or the extrinsic matrix. Since in augmented reality, we do not necessarily know where the users camera is in an absolute world frame, but only relative to some (possibly moving) objects, we tend to use the camera frame as the world frame, and the both the rotation and translation components of the transformation matrix are zero. The final parameter is the projection matrix, which encompasses the whole linear camera model

```
struct mxrCamera{
      mxrFloat fx;
      mxrFloat fy;
      mxrFloat ox;
      mxrFloat oy;
      mxrFloat skew;
```

```
      mxrFloat r2;
      mxrFloat r4;
      mxrFloat xy1;
      mxrFloat xy2;
      mxrTransform T;
      mxrProjMat P;
};
```

TWO DIMENSIONAL IMAGE POINT: `mxrPoint2D`

This structure describes a two dimensional image position. Readers not well versed in computer graphics/ computer vision may ask the obvious question "why does a two dimensional point need three parameters to describe it?" The short answer is that this is a "homogeneous" 2D point – for most cases you will handle, the third value, z will be set to one and the first two values will operate like normal x and y co-ordinates.

```
struct mxrPoint2D{
      mxrFloat x;
      mxrFloat y;
      mxrFloat z;
};
```

TWO-DIMENSIONAL LINE: `mxrLine`

This structure describes a line in two dimensions (e.g. an image!). The parameters describe the line in the form:

$$a\,x + b\,y + c = 0$$

```
struct mxrLine{
      mxrFloat a;
      mxrFloat b;
      mxrFloat c;
};
```

THREE DIMENSIONAL IMAGE POINT: `mxrPoint3D`

This structure contains information about a position in three dimensional space. As with the 2D point structure, the 3D position is represented with one extra number and is termed a "homogeneous" co-ordinate. If you don't know what this means, then expect the last component to be set to one and the first three components to behave like normal X,Y,Z components. Any book on computer vision or computer graphics will fill you in on the subtleties of homogenous co-ordinates.

```
struct mxrPoint3D{
      mxrFloat X;
      mxrFloat Y;
      mxrFloat Z;
      mxrFloat W;
};
```

THREE-DIMENSIONAL PLANE: `mxrPlane3D`

The mxrPlane structure describes the position and orientation of a plane in three-dimensional space and is the 3D analogue of the 2D line. The parameters can be interpreted in the light of the equation defining the plane:

$$A X + B Y + C Z + D = 0$$

```
struct mxrPlane{
      mxrFloat A;
      mxrFloat B;
      mxrFloat C;
      mxrFloat D;
};
```

QUATERNION: `mxrQuat`

The mxrQuat structure contains the four components of a quaternion. If you are not familiar with quaternions, then all you need to know is that this is an alternative way of describing three-dimensional rotations that has some advantages over other methods. It is particularly useful for interpolation between two rotation positions. In fact the parameters describe a four-dimensional complex number where r is the real part, and i, j and k are the components representing the first, second and third complex dimensions respectively.

```
struct mxrQuat{
      mxrFloat r;
      mxrFloat i;
      mxrFloat j;
      mxrFloat k;
};
```

ROTATION VECTOR: `mxrRotVec`

This is a third representation for 3d rotations, in addition to a 3x3 rotation matrix and a quaternion. The parameters taken as a vector represent the direction of the axis of rotation. The magnitude of the vector represents the size of the rotation in radians. From a technical standpoint, this representation has the advantage that it is a minimal representation and is hence suitable for use in optimization routines.

```
struct mxrRotVec{
      mxrFloat rx;
      mxrFloat ry;
      mxrFloat rz;
};
```

TRANSFORMATION MATRIX `mxrTransform`

This represents a Euclidean transformation matrix. This is one of the most important components of the MXRToolkit to understand. In simple terms, the relationship between

any two positions in the real world can be described by a rotation and a translation. The "r" components of the structure define the rotation and the t components represent the translation. The rotation is applied before the translation.
In fact the rotation components refer to the elements of a 3x3 rotation matrix, which is a special matrix with the property the its inverse is its own transpose (i.e. the columns are orthogonal and or unit length). The translation components refer to movement in the x,y, and z directions in mm, respectively.

```
struct mxrTransform{
      mxrFloat r11;
      mxrFloat r12;
      mxrFloat r13;
      mxrFloat tx;
      mxrFloat r21;
      mxrFloat r22;
      mxrFloat r23;
      mxrFloat ty;
      mxrFloat r31;
      mxrFloat r32;
      mxrFloat r33;
      mxrFloat tz;
};
```

Matrix Structure mxrMatrix

The matrix structure is a minimal representation of a matrix for mathematics. It holds a the x and y dimensions of the matrix and a pointer to the data. The data is stored along the rows first, in the order A11, A12, A13… A21., A22, … Ann. In fact this is technically a C++ class with a constructor that sets the image pointer to null, and a destructor that clears the memory associated with the pointer if the pointer is not set to NULL. Hence, the memory associated with the matrix is automatically removed when the variable goes out of scope. A similar system is applied with the mxrImage object.

```
struct mxrMatrix{
      int x;
      int y;
      mxrFloat *data;
};
```

Image Structure mxrImage

The image structure contains details about an image. The first two parameters refer to the x any y size of the image. The next parameter is a pointer to the image data. The fourth parameter gives the storage type of the image. Valid values are MXR_RGB, MXR_BGR, MXR_RGBA, MXR_BGRA, MXR_GRAYSCALE. The final parameter contains details about the camera with which the image was taken, when known.

```
struct mxrImage{
```

```
	int x;
	int y;
	unsigned char *data;
	mxrImFormat format;
	mxrCamera cam;
};

typedef enum mxrImFormat{
	MXR_RGB,
	MXR_RGBA,
	MXR_GRAYSCALE,
	MXR_BGR,
	MXR_BGRA,
};
```

OPTIMIZATION LIBRARY

The optimization library provides a simple to use Levenberg-Marquardt optimization routine. This is a gradient-based algorithm and hence requires that the derivatives of the underlying function are well behaved.

The aim of these optimization routines is to minimize and error function over a number of parameters. To provide a concrete example, imagine fitting a straight line through two dimensional data points. We wish to minimize the mean squared deviation of the points from the line as a function of the two parameters (slope and y-intersect) of the line. We term the number of data points nData, and the number of parameters nParam.

In order to solve this problem, we must provide (i) an initial estimate of the parameters to be estimated, and (ii) an error function that estimates the error for each contributing point given a certain number of parameters.

The optimization library currently consists of only two routines:

- mxrOptimNonLin – general interface for LM algorithm, requiring the user to provide both the error and the derivatives of the error at the current point
- mxrOptimNonLinFD – finite difference interface for LM algorithm – here the user only has to provide the error for a given set of parameters, and the derivatives are calculated automatically using finite differences.

Parameter Passing:

The error functions describe above often require data passed to them. However, the routines only pass one void pointer for the data in general. If the data is not all in one convenient structure/class then we may have to create one. To circumvent this problem, we introduce a simple structure "optimParam":

```
struct optimParam{
        void *param1;
        void *param2;
        void *param3;
        void *param4;
        void *param5;
        void *param6;
};
```

We fill the components of the structure with pointers to the various data we require, and then simply pass a pointer to this structure to the error function.

mxrOptimNonLin *Release Version:* MXRToolkit V1.0

Short Description:

A convenient front end for Levenberg Marquardt optimization. Optimizes an arbitrary error function based on nData points over some set of nParam parameters. User must supply error function and derivatives.

Function declaration:

```
void mxrOptimNonLin(mxrFloat *chiSq,mxrMatrix *paraMat, int nData,
                    void (*errorFn)(mxrMatrix *, mxrMatrix *, mxrMatrix *, void *),
                    void *dataPtr);
```

Parameter Interpretation

- `chiSq` returns the final value of the sum of the square of the errors.
- `nData` passes the number of data points.
- paraMat is a (nParam x 1) matrix which contains the intial estimates of the parameters upon entry, and the final estimates upon leaving.
- `errorFn` is a pointer to a function which calculates the error on each iteration. See below for function declaration.
- `dataPtr` is null pointer which is passed to the error routine on each iteration. It should be used to pass the data to the error function.

The error function should have the following declaration:

```
void errorFn(mxrMatrix *error, mxrMatrix *dErrordParam, mxrMatrix *param, void *dataPtr)
```

The parameters here have the following interpretation:

- `error` is an (nData x 1) matrix into which the error values should be placed
- `dErrordParam` is an (nData x nParam) matrix into which the derivatives of the error values with respect to each parameter should be placed
- `param` is an (nParam x 1) matrix which passes the current values of the parameters
- `dataPtr` is a pointer to the data points used to calculate each error value

Example: See mxrOptimNonLinFD for a similar example

Notes: See note in main optimization description on passing parameters using the `optimParam` structure

See also: mxrOptimNonLinFD

mxrOptimNonLinFD *Release Version:* MXRToolkit V1.0

Short Description:

Easy front end for Levenberg-Marquardt non-linear optimization – calculates derivatives using finite differences. Minimizes and error function based on nData points as a function of nParam parameters.

Function Declaration:

```
void mxrOptimNonLinFD(mxrFloat *chiSq,mxrMatrix *paraMat, int nData, void
(*errorFn)(mxrMatrix *, mxrMatrix *, void *), void *dataPtr,int maxIter, mxrFloat fdInc);
```

Parameter Interpretation

- `chiSq` returns the final value of the sum of the square of the errors.
- paraMat is a (nParam x 1) matrix which contains the initial estimates of the parameters upon entry, and the final estimates upon leaving. The number of parameters is passed implicitly via the y size of this matrix
- `nData` passes the number of data points.
- `errorFn` is a pointer to a function which calculates the error on each iteration. See below for function declaration.
- `dataPtr` is null pointer which is passed to the error routine on each iteration. It should be used to pass the data to the error function.
- maxIter is the maximum number of iterations that performed by the optimization routine.
- fdInc is the step size used in the finite differencing calculation. It should be of the order of the precision of the answer required (i.e. a small number).

The error function should have the following declaration:

```
void errorFn(mxrMatrix *error, mxrMatrix *param, void *dataPtr)
```

The parameters here have the following interpretation:

- `error` is an (nData x 1) matrix into which the error values should be placed
- `param` is an (nParam x 1) matrix which passes the current values of the parameters
- `dataPtr` is a pointer to the data points used to calculate each error value

Example: Optimizing estimates of extrinsic matrix

The routine mxrOptimizeTransform takes some three-dimensional data points (P1), their projected points into the current image (P2), and the current camera matrix, and aims to find the Euclidean transformation T which best accounts for this projection.

The initial estimate of the Euclidean transformation is provided in the mxrTransform structure, T, which consists of the components of a rotation matrix and a translation vector. However, the rotation matrix is somewhat redundant due to the non-linear constraints between the components. We convert this matrix to a minimal parameterization (rotation vector, mxrRotVec) before performing the optimization in order to help ensure a

unique minimum. After performing the optimization, we change the rotation vector back to a rotation matrix.

The 2D and 3D data points must be passed to the error function on each iteration and are wrapped in the first two fields of the structure optimParam.

```
/* ROUTINE TO OPTIMIZE EUCLIDEAN TRANSFORMATION MATRIX */
void    mxrOptimizeTransform (mxrTransform *T,mxrCamera *camera,
                                        mxrPoint3D *P1,mxrPoint2D *P2, int n){

        mxrMatrix                   paraMat;
        mxrFloat                    chiSq;
        mxrRotVec                   r;
        optimParam                  opt;
        mxrMatrixCreate (&paraMat,6,1);


        /* CONVERT TO MINIMAL PARAMETERIZATION */
        mxrTransformToRotVec(&r,T);
        paraMat.data[0] = r.rx;
        paraMat.data[1] = r.ry;
        paraMat.data[2] = r.rz;
        paraMat.data[3] = T->tx;
        paraMat.data[4] = T->ty;
        paraMat.data[5] = T->tz;

        /* PASS DATA POINTS AND PARAMETERS TO OPTIMIZATION ROUTINE */
        opt.param1 = (void *) P1;
        opt.param2 = (void *) P2;
        opt.param3 = (void *) camera;
        mxrOptimNonLinFD(&chiSq,&paraMat,n*2,transformError,(void *)&opt,8,0.00001);

        /* CONVERT PARAMETERS BACK TO TRANSFORMATION MATRIX FORM */
        r.rx = paraMat.data[0];
        r.ry = paraMat.data[1];
        r.rz = paraMat.data[2];
        mxrRotVecToTransMat(T,&r);
        T->tx = paraMat.data[3];
        T->ty = paraMat.data[4];
        T->tz = paraMat.data[5];

        mxrMatrixDelete(&paraMat);
}
```

The error function, transform error has the following structure: firstly we retrieve the data points from the passed structure. We transform the rotation vector into the more convenient transformation matrix. Then we use this transformation matrix and the camera matrix to project the 3D points into the current image. The errors are simply calculated as the differences between the predicted and actual image points. Note that in this case, the number of error estimates is twice the number of points as we return the difference in both the x and y positions.

```
/* ERROR FUNCTION - CALLED BY OPTIMIZATION ROUTINE */
void transformError(mxrMatrix *error, mxrMatrix *paraMat, void *dataPtr){
        int c1;
        mxrRotVec r;
        mxrTransform T;
        mxrPoint3D *P1; mxrPoint2D *P2;
        mxrFloat x,y,z;
        optimParam *opt;
        mxrCamera *camera;
```

```
        /* RETRIEVE PASSED DATA FROM OPTIMPARAM STRUCTURE */
        opt = (optimParam *) dataPtr;
        P1 = (mxrPoint3D *)opt->param1;
        P2 = (mxrPoint2D *)opt->param2;
        camera  = (mxrCamera *)opt->param3;


        /* CONVERT PARAMETERS BACK FROM MINIMAL PARAMETERIZATION */
        r.rx = paraMat->data[0];
        r.ry = paraMat->data[1];
        r.rz = paraMat->data[2];
        mxrRotVecToTransMat(&T,&r);
        T.tx = paraMat->data[3];
        T.ty = paraMat->data[4];
        T.tz = paraMat->data[5];

        /* CALCULATE ERROR FOR EACH POINT */
        for (c1 = 0; c1<yDiff->y/2; c1++){

                /* APPLY TRANSFORMATION MATRIX */
                x = T.r11*P1[c1].X+T.r12*P1[c1].Y+T.r13*P1[c1].Z+T.tx;
                y = T.r21*P1[c1].X+T.r22*P1[c1].Y+T.r23*P1[c1].Z+T.ty;
                z = T.r31*P1[c1].X+T.r32*P1[c1].Y+T.r33*P1[c1].Z+T.tz;

                /* DIVIDE THROUGH HOMOGENOUS CO-ORDINATES */
                x = x/z;
                y = y/z;

                /* APPLY CAMERA MATRIX TO GENERATE PREDICTED CAMERA CO-ORDS */
                x = x*camera->fx-y*camera->skew-camera->ox;
                y = y*camera->fy-camera->oy;

                /* FILL IN ERROR VECTOR WITH DIFF BETWEEN PREDICTED AND MEASURED POINTS
                error->data[c1*2+0] =  (P2[c1].x-x);
                error->data[c1*2+1] =  (P2[c1].y-y);
        }
}
```

Notes: See note in main optimization description on passing parameters using the `optimParam` structure

See also: mxrOptimNonLin

CAMERA LIBRARY

The camera library deals with the forward camera model. This describes how a point in the three-dimensional world is mapped to a certain pixel in the final image. It is essential that this process is modeled accurately in order to recover geometrical aspects of the real world. The camera model can be divided into three components:

- linear camera model (intrinsic matrix) – includes information such as the focal length and field of view of the camera
- non-linear or radial distortion component – a model for distortion induced by non-ideal real world lenses
- the position and orientation of camera in the world (extrinsic matrix)

Often in mixed reality, we have good prior estimates of the first two components, and we aim to solve for the latter component. Once we have retrieved the camera pose relative to real world objects, it is a simple matter to integrate computer graphics into the real world scene. The parameters of all three parts of the model are described in the `mxrCamera` structure:

```
struct mxrCamera{
        mxrFloat fx;
        mxrFloat fy;
        mxrFloat ox;
        mxrFloat oy;
        mxrFloat skew;
        mxrFloat r2;
        mxrFloat r4;
        mxrFloat xy1;
        mxrFloat xy2;
        mxrTransform T;
        mxrProjMat P;
};
```

The first five parameters describe the *intrinsic matrix* or linear camera model. In an ideal camera of unity focal length, placed at the origin with the optical axis lined up along the positive Z axis, a 3D point in the world (X,Y,Z) projects to the points (x_{cam},y_{cam}) by the equation:

$$X_{cam} = X/Z;$$
$$Y_{cam} = Y/Z;$$

However, real world cameras do not have unity focal length. Moreover, the point (0,0) in pixel co-ordinates is not in the centre of the image, but the top-left. The intrinsic matrix converts from ideal camera co-ordinates (x_{cam},y_{cam}) to image co-ordinates (x_{im},y_{im}).

$$x_{im} = f_x * x_{cam} + skew * y_{cam} + o_x$$
$$y_{im} = f_y * y_{cam} + o_y$$

These parameters are given real world interpretations as follows:

- fx is the focal length in the x direction expressed in pixels
- fy is the focal length in the y direction expressed in pixels
- ox is the x offset of the centre of the image from the top-left corner

- oy is the y offset of the image centre from the top-left corner
- skew describes the "skewness" of the CCD array with respect to the camera axis

The parameters, r2, r4, xy1, xy2 describe the non-linear or radial distortion component of the model. In a real camera, the linear model generally doesn't apply, but a simple distortion of the camera co-ordinates before applying the intrinsic matrix corrects most of the problems.

$$x_{dist} = (1+r_2*r_{cam}^2+r_4*r_{cam}^4) *x_{cam} + 2* xy_1*x_{cam}*y_{cam}+xy_2*(r_{cam}+2*x_{cam}^2)$$
$$y_{dist} = (1+r_2*r_{cam}^2+r_4*r_{cam}^4) *x_{cam} + xy_1*(r_{cam}^2+2*y_{cam}^2)+2*xy_2*x_{cam}*y_{cam}$$

where:

$$r_{cam}^2 = x_{cam}^2+y_{cam}^2$$

The final component of the distortion model is the extrinsic matrix. This is a transformation matrix and describes the position and orientation of the camera with respect to the world co-ordinate system. In particular, the matrix T transforms points form the world reference frame to a reference frame centered on the camera. In other words, a world point must be pre-multiplied by the transformation matrix, T before the camera model described above is applied.

In the absence of the radial distortion component, the model is completely linear and under some circumstances, the intrinsic and extrinsic matrices are concatenated to form a single 4 x 3 matrix called the projection matrix. This part of the image structure is not often used or even maintained in the camera structure, but it is included for completeness.

```
struct mxrProjMat{
        mxrFloat p11;
        mxrFloat p12;
        mxrFloat p13;
        mxrFloat p14;
        mxrFloat p21;
        mxrFloat p22;
        mxrFloat p23;
        mxrFloat p24;
        mxrFloat p31;
        mxrFloat p32;
        mxrFloat p33;
        mxrFloat p34;
};
```

Here, the relationship between a world point (X,Y,Z) and an image point (x,y) is given by:

$$x = (p_{11} * X + p_{12} * Y + p_{13} *Z + p_{14}) / (p_{31} * X+ p_{32} * Y + p_{33} *Z + p_{34})$$
$$y = (p_{21} * X + p_{22} * Y + p_{23} *Z + p_{24}) / (p_{31} * X+ p_{32} * Y + p_{33} *Z + p_{34})$$

The camera library currently consists of the following routines:

- mxrRadialDistortion – applies the forward radial distortion to a list of points
- mxrRadialDistortionOne – applies the forward radial distortion to a single x,y point

- mxrRadialDistortionInvert – applies the inverse of the radial distortion function to a list of points
- mxrRadialDistortionInvertOne – applies the inverse of the radial distortion function to a single x,y point
- mxrCameraDefault – fills the camera structure with some sensible parameters given the height and width of the current image
- mxrCameraPrint – prints out the current contents of the camera structure to the screen
- mxrProjMatCreate – takes the intrinsic parameters from a camera matrix and a separate set of extrinsic parameters and concatenates them to form a 3x4 projection matrix
- mxrProjMatUpdate – takes a camera matrix and uses its internal and external parameters to update its projection matrix
- mxrCamToImageCoords – takes a list of two dimensional points in ideal image co-ordinates and converts them to image(pixel) co-ordinates using the intrinsic matrix from the camera structure
- mxrImageToCamCoords – takes a list of image (pixel) points and converts them to camera co-ordinates.
- mxrPoint3DProject – uses the linear camera model to project a 3D point down to a 2D (pixel) point.
- mxrPointXYZProject – uses the linear camera model to project a 3D point that was explicitly specified as X,Y,Z values
- mxrCameraRead – loads a camera structure from a file on the hard disk
- mxrCameraWrite – saves the camera structure to a file on the hard disk
- mxrCameraGetIntrinsic – extracts the intrinsic parameters and puts them into a 3x3 matrix
- mxrCameraGetIntrinsicInverse – extracts the inverse intrinsic matrix parameters and puts them into a 3x3 matrix structure.

mxrCameraDefault *Release Version:* MXRToolkit V1.0

Short Description:

Places some sensible default parameters in a camera structure, given a height and width value. Useful for when you don't know the exact calibration data for your camera, or you are trying to calibrate it and need to some sensible initial settings for the optimization.

Function declaration:

```
void    mxrCameraDefault(mxrCamera *camera,int imX,int imY);
```

Parameter Interpretation

- `camera` is a pointer to the camera structure which is to be filled
- `imX` is the horizontal size of the camera image in pixels
- `imY` is the vertical size of the camera image in pixels

Example:

```
mxrCamera camera;
mxrCameraDefault(&camera,640, 480);                 // default values for cam struc
```

Notes:

The exact values of the put into the camera structure as follows. The focal length parameters of the intrinsic matrix are set to the horizontal width (i.e. the horizontal field of view is 45 degrees). The x and y offset parameters are set to half of the width/ height respectively, so that the optical axis passes through the centre of the image. The skew and the radial distortion parameters are set to zero, and the extrinsic matrix is set so there is no rotation or translation.

See also:

mxrCameraRead, mxrCameraWrite

mxrCameraPrint *Release Version:* MXRToolkit V1.0

Short Description:

Prints out the parameters of the current camera to the screen.

Function declaration:

```
void    mxrCameraPrint(mxrCamera *camera);
```

Parameter Interpretation

- `camera` is a pointer to the camera structure who's parameters we are about to display.

Example:

```
mxrCamera camera;

mxrCameraDefault(&camera,640, 480);                 // default values for cam struc
mxrCameraPrint(&camera);                            // print out camera values
```

Notes:

Currently this is just printed to the standard output stream.

See also:

mxrCameraRead, mxrCameraWrite, mxrCameraDefault.

mxrCameraGetIntrinsicInverse *Release Version:* MXRToolkit V1.0

Short Description:

Copies intrinsic parameters out of the camera structure and places them in a 3x3 matrix and the inverts this matrix.

Function declaration:

```
void    mxrCameraGetIntrinsicInverse(mxrMatrix *matrix, mxrCamera *camera);
```

Parameter Interpretation

- `matrix` is a pointer to an mxrMatrix structure, with dimensions 3x3 which will contain the inverse of the intrinsic matrix on return
- camera is a pointer to an mxrCamera structure

Example:

```
mxrCamera camera;
mxrCameraDefault(&camera,640, 480);                 // default values for cam struc

mxrMatrix M;
mxrMatrixCreate(&M,3,3);                            // allocate matrix memory

mxrCameraGetIntrinsicInverse(&M,&cam);              // extract inverse parameters

mxrMatrixDelete(&M,3,3);                            // delete matrix memory
```

Notes:

For some applications it is useful to treat the intrinsic matrix as a 3x3 matrix, which can be used to pre- or post-multiply other geometric entities. As such this matrix can be inverted and subsequently represents the transformation from image to camera co-ordinates. This routine assumes that the memory for the 3x3 matrix is already declared. The resulting 3x3 matrix is upper triangular.

See also:

mxrCameraGetIntrinsic

mxrCameraGetIntrinsic *Release Version:* MXRToolkit V1.0

Short Description:

Copies intrinsic parameters out of the camera structure and places them in a 3x3 matrix.

Function declaration:

```
void    mxrCameraGetIntrinsic(mxrMatrix *matrix, mxrCamera *camera);
```

Parameter Interpretation

- `matrix` is a pointer to an mxrMatrix structure, with dimensions 3x3 which will contain the intrinsic matrix on return
- camera is a pointer to an mxrCamera structure

Example:

```
mxrCamera camera;
mxrCameraDefault(&camera,640, 480);                 // default values for cam struc

mxrMatrix M;
mxrMatrixCreate(&M,3,3);                            // allocate matrix memory

mxrCameraGetIntrinsic(&M,&cam);                     // extract intrinsic parameters

mxrMatrixDelete(&M,3,3);                            // delete matrix memory
```

Notes:

For some applications it is useful to treat the intrinsic matrix as a 3x3 matrix, which can be used to pre- or post-multiply other geometric entities. This routine assumes that the memory for the 3x3 matrix is already declared. The resulting 3x3 matrix is upper triangular.

See also:

mxrCameraGetIntrinsicInverse

mxrCameraRead *Release Version:* MXRToolkit V1.0

Short Description:

Reads in camera parameters from stored file on the disk to an mxrCamera structure.

Function declaration:

```
void    mxrCameraRead(mxrCamera *camera, char *filename, int width, int height);
```

Parameter Interpretation

- `camera` is a pointer to the camera structure which will be filled with data from the file.
- `filename` is a c-style null-terminated string which stores the name of the file.
- `width` is an integer describing the current width of the video stream in pixels.
- `height` is an integer describing the current height of the video stream in pixels.

Example:

```
mxrCamera camera;

mxrCameraRead(&camera,"myCam.mxrCam",640,480);       // load in new camera values
mxrCameraPrint(&camera);                             // print out camera values
```

Notes:

The standard file extension for a camera used within MXRToolkit is ".mxrCam". The file format is simply a binary copy of the structure prefaced by an integer version number. The height and width of the video stream must be specified as some of the camera parameters need to be scaled to the current resolution.

See also:

mxrCameraPrint, mxrCameraWrite, mxrCameraDefault.

mxrCameraWrite *Release Version:* MXRToolkit V1.0

Short Description:

Writes camera parameters to stored file on the disk from an mxrCamera structure.

Function declaration:

```
void    mxrCameraWrite(char *filename, mxrCamera *camera, int width , int height);
```

Parameter Interpretation

- `filename` is a c-style null-terminated string which stores the name of the file to be written
- `camera` is a pointer to the camera structure whose parameters we are about to save.
- `width` is an integer describing the current width of the video stream in pixels.
- `height` is an integer describing the current height of the video stream in pixels.

Example:

```
mxrCamera camera;

mxrCameraDefault(&camera,640,480);                  // fill structure with some value
mxrCameraWrite ("myCam.mxrCam", &camera,640,480);   // save new camera values
```

Notes:

The standard file extension for a camera used within MXRToolkit is ".mxrCam". The file format is simply a binary copy of the structure. The height and width of the current video stream must be specified as some of the parameters depend on the resolution. For the purposes of storage, these are rescaled so that they are suitable for a VGA (640x480) image.

See also:

mxrCameraRead, mxrCameraRead, mxrCameraDefault.

mxrCamToImageCoords *Release Version:* MXRToolkit V1.0

Short Description:

Converts ideal camera co-ordinates into pixel co-ordinates in an image, by applying the intrinsic parameters in the camera structure.

Function declaration:

```
void    mxrCamToImageCoords(mxrPoint2D *imP, mxrCamera *A, mxrPoint2D *camP, int n);
```

Parameter Interpretation

- `imP` is a pointer to a list of image points in pixels which are filled by the routine.
- `camera` is a pointer to the camera structure whose intrinsic parameters which are used to apply the intrinsic matrix
- `camP` is a pointer to a list of two dimensional points expressed in ideal camera co-ordinates.
- `n` is the number of points in the list

Example:

```
mxrCamera camera;
mxrCameraDefault(&camera,640,480);                  // fill structure with some value

mxrPoint2D P;
P.x = -0.11; P.y = 0.25; P.z = 1;

mxrCamToImageCoords(&P,&camera, &P,1);              // convert point P to image co-ords
```

Notes:

The algorithm is implemented in such a way that it can be carried out in place (i.e. the destination and source pointers can be the same block of memory). Ideal camera co-ordinates are the two-dimensional positions that would be produced by perspective projection into a camera of unity focal length, with the optical axis pointing along the positive Z direction.

See also:

mxrImageToCamCoords

mxrImageToCamCoords *Release Version:* MXRToolkit V1.0

Short Description:

Converts image or pixel co-ordinates into ideal camera co-ordinates, by applying the inverse of the intrinsic parameters from the camera structure. Essentially, this makes the co-ordinates neutral to the particular camera used to capture them, which is important for many computer vision applications.

Function declaration:

```
void    mxrImageToCamCoords(mxrPoint2D *camP, mxrCamera *A, mxrPoint2D *imP, int n);
```

Parameter Interpretation

- `camP` is a pointer to a list of image points, which are filled with camera co-ordinate values by this routine.
- `camera` is a pointer to the camera structure whose intrinsic parameters which are used to apply the inverse intrinsic matrix
- `imP` is a pointer to a list of two dimensional points expressed in pixels
- `n` is the number of points in the list

Example:

```
mxrCamera camera;
mxrCameraDefault(&camera,640,480);                  // fill structure with some value

mxrPoint2D P;
P.x = 345; P.y = 120; P.z = 1;

mxrImageToCameraCoords(&P,&camera, &P,1);           // convert point P to image co-ords
```

Notes:

The algorithm is implemented in such a way that it can be carried out in place (i.e. the destination and source pointers can be the same block of memory). Ideal camera co-ordinates are the two-dimensional positions that would be produced by perspective projection into a camera of unity focal length, with the optical axis pointing along the positive Z direction.

See also:

mxrCamToImageCoords

mxrProjMatCreate *Release Version:* MXRToolkit V1.0

Short Description:

Combines intrinsic and extrinsic parameters to form a 4x3 projection matrix which represents the whole linear part of the model.

Function declaration:

```
void    mxrProjMatCreate(mxrProjMat *P,mxrCamera *camera,mxrTransform *T);
```

Parameter Interpretation

- `P` is a pointer to the projection matrix structure which will be filled.
- `camera` is a pointer to the camera structure whose intrinsic parameters are to be used
- T is a pointer to an mxrTransform structure which contains the desired extrinsic parameters.

Example:

```
mxrCamera camera;
mxrCameraDefault(&camera,640,480);          // fill camera structure with some values

mxrTransform T;
mxrTransformLoadIdentity(&T);
T.tz = 300; T.ty = 23;                      // fill out transformation matrix structure

mxrProjMat P;
mxrProjMatCreate(&P,&camera,&T);            // compute projection matrix
```

Notes:

The projection matrix is useful when a lot of points need to be projected into the image very fast. It is more efficient that applying the intrinsic and extrinsic parameters in turn.

See also:

mxrProjMatUpdate.

mxrProjMatUpdate *Release Version:* MXRToolkit V1.0

Short Description:

Combines intrinsic and extrinsic parameters from the camera matrix to update the 4x3 projection matrix structure. This represents the whole linear part of the model.

Function declaration:

```
void    mxrProjMatUpdate (mxrCamera *camera);
```

Parameter Interpretation

- `camera` is a pointer to the camera structure whose intrinsic and extrinsic parameters are to be used. On return, the projection matrix field of the camera structure is updated based on these parameters

Example:

```
mxrCamera camera;
mxrCameraDefault(&camera,640,480);          // fill camera structure with some values

mxrTransform T;
mxrTransformLoadIdentity(&T);
T.tz = 300; T.ty = 23;                      // fill out transformation matrix structure

mxrProjMatCreate(&camera);                  // compute projection matrix
```

Notes:

The projection matrix is useful when a lot of points need to be projected into the image very fast. It is more efficient that applying the intrinsic and extrinsic parameters in turn. This routine is equivalent to:

mkxrProjMatCreate(&camera.P, &camera, &camera.T);

See also:

mxrProjMatCreate.

mxrRadialDistortion *Release Version:* MXRToolkit V1.0

Short Description:

Implements the non-linear part of the camera model by distorting the image according to the radial distance from the image centre. Takes a list of points in image co-ordinates and returns a second distorted list also in image co-ordinates. Internally it converts first to camera co-ordinates, distorts and then transfers back to image co-ordinates.

Function declaration:

```
void    mxrRadialDistortion(mxrPoint2D *outPts, mxrPoint2D *inPts, mxrCamera *camera,
                                                                              int n);
```

Parameter Interpretation

- `outPts` is a pointer to the destination list of image points.
- `inPts` is a pointer to the source list of image points.
- `Camera` is a pointer to the camera structure which contains the four parameters needed to implement the radial distortion.
- `n` is an integer passing the number of points in the source and destination lists.

Example:

```
mxrPoint2D A[20];
mxrPoint2d B[20];
mxrCamera cam;

/* (FILL A WITH PIXEL POSITIONS AND cam WITH DATA HERE) */

mxrRadialDistortion (B,A,&cam,20);

/* NOW B CONTAINS DISTORTED VERSIONS OF POINTS THAT WERE IN A */
```

Notes:

Note carefully that this routine operates on image points in pixels rather than points expressed in ideal camera co-ordinates, although the actual parameters describe the relationship between camera co-ordinates before and after distortion. Internally, the image points are converted to camera points, distorted and converted back to image points.

See also:

mxrRadialDistortionOne, mxrRadialDistortionInvert, mxrRadialDistortionInvertOne

mxrRadialDistortionOne *Release Version:* MXRToolkit V1.0

Short Description:

Implements the non-linear part of the camera model by distorting the image according to the radial distance from the image centre. However, it operates directly on a single x,y position value rather than a list of points as in the main radial distortion routine.

Function declaration:

```
void   mxrRadialDistortionOne(mxrFloat *xOut, mxrFlout *yOut, mxrFloat xIn, mxrFloat yIn,
mxrCamera *camera);
```

Parameter Interpretation

- `xOut` is a pointer to an double where the distorted x value in pixels will be placed
- `yOut` is a pointer to an double where the distorted y value in pixels will be placed
- `xIn` is the original x co-ordinate in pixels
- `yIn` is the original y co-ordinate in pixels
- `Camera` is a pointer to the camera structure which contains the four parameters needed to implement the radial distortion.

Example:

```
mxrFloat x = 210;
mxrFloat y = 124;
mxrCamera camera;

mxrCameraDefault(&camera,640, 480);                 // default values for cam struc
camera.r2 = 0.119; camera.r4 = -0.0012              // fill in 2 rad. dist components

mxrRadialDistortionOne(&x, &y,x,y,&camera);         // distort point
```

Notes:

Note carefully that this routine operates on image points in pixels rather than points expressed in ideal camera co-ordinates, although the actual parameters describe the relationship between camera co-ordinates before and after distortion. Internally, the image points are converted to camera points, distorted and converted back to image points.

See also:

mxrRadialDistortion, mxrRadialDistortionInvert, mxrRadialDistortionInvertOne

mxrRadialDistortionInvert *Release Version:* MXRToolkit V1.0

Short Description:

Inverts the non-linear radial distortion function. Takes a list of points from a real camera and returns a list of points that have had their position adjusted to account for the estimated radial distortion. If these estimates are correct then the camera model should be linear after calling this function.

Function declaration:

```
void    mxrRadialDistortionInvert(mxrPoint2D* outPts,mxrPoint2D *inPts,mxrCamera *camera,
                                                                        int nPoints);
```

Parameter Interpretation

- `outPts` is a pointer to the list of undistorted 2D points returned by the routine, in pixels
- `inPts` is a pointer to the list of original distorted points, also expressed in pixels
- `camera` is a pointer to a camera structure which contains the parameters needed to invert the radial distortion function
- `nPoints` is the number of points in the list to be converted

Example:

```
mxrPoint2D;
mxrCamera camera;

mxrCameraDefault(&camera,640, 480);                 // default values for cam struc
camera.r2 = 0.119; camera.r4 = -0.0012              // fill in 2 rad. dist components
P.x = 230; P.y = 12; P.z = 1;                       // fill in point positions

mxrRadialDistortionInvert(&P, &P,&camera,1);        // distort point
```

Notes:

Note carefully that this routine operates on image points in pixels rather than points expressed in ideal camera co-ordinates, although the actual parameters describe the relationship between camera co-ordinates before and after distortion. Internally, the image points are converted to camera points, distorted and converted back to image points. There is no explicit formula for finding the inverse of the radial distortion function, so an iterative algorithm is applied, which requires more computation than the forward model. The code is such that the inversion can be done in place by passing the same pointer in the first and second parameter.

See also:

mxrRadialDistortion, mxrRadialDistortionOne, mxrRadialDistortionInvert

mxrRadialDistortionInvertOne *Release Version:* MXRToolkit V1.0

Short Description:

Inverts the non-linear part of the camera model by attempting to compensate for the radial distortion. However, it operates directly on a single x,y position value rather than a list of points as in the main radial distortion inversion routine.

Function declaration:

```
void    mxrRadialDistortionInvertOne(mxrFloat *xOut,mxrFloat *yOut,mxrFloat xIn,
                                                    mxrFloat yIn,mxrCamera *camera);
```

Parameter Interpretation

- `xOut` is a pointer to an double where the undistorted x value in pixels will be placed
- `yOut` is a pointer to an double where the undistorted y value in pixels will be placed
- `xIn` is the distorted (measured) x co-ordinate in pixels
- `yIn` is the distorted(measured) y co-ordinate in pixels
- `camera` is a pointer to the camera structure which contains the four parameters needed to implement the radial distortion.

Example:

```
mxrFloat x = 210;
mxrFloat y = 124;
mxrCamera camera;

mxrCameraDefault(&camera,640, 480);                 // default values for cam struc
camera.r2 = 0.119; camera.r4 = -0.0012              // fill in 2 rad. dist components

mxrRadialDistortionInvertOne(&x,&y,x,y,&camera);    // distort point
```

Notes:

Note carefully that this routine operates on image points in pixels rather than points expressed in ideal camera co-ordinates, although the actual parameters describe the relationship between camera co-ordinates before and after distortion. Internally, the image points are converted to camera points, undistorted and converted back to image points. There is no explicit formula for the inversion of the radial distortion function so the algorithm is iterative and hence is more computationally expensive than the forward radial distortion model.

See also:

mxrRadialDistortion, mxrRadialDistortionOne, mxrRadialDistortionInvertOne

mxrPointXYZProject *Release Version:* MXRToolkit V1.0

Short Description:

Projects a three-dimensional point expressed in X,Y, and Z co-ordinates down into an image, using intrinsic parameters from a camera structure and a separate transformation matrix.

Function declaration:

```
void    mxrPointXYZProject (mxrPoint2D *pOut, mxrCamera *A, mxrTransform *T,
                                                    mxrFloat X,mxrFloat Y,mxrFloat Z);
```

Parameter Interpretation

- `pOut` is a pointer to a two dimensional image point in pixels which is the result of the projection
- `camera` is a pointer to an mxrCamera structure which contains the intrinsic parameters used in the projection.
- `T` contains a pointer to a Euclidean transform structure representing the extrinsic matrix of the camera.
- `X` contains the three-dimensional X co-ordinate of the point
- `Y` contains the three-dimensional Y co-ordinate of the point
- `Z` contains the three-dimensional Z co-ordinate of the point

Example:

```
mxrCamera camera;                                   // declare camera and fill values
mxrCameraDefault(&camera,640, 480);

mxrTransform T;                                     // declare transformation and fill
T.x = 240; T.y = 25; T.z = 25;

mxrPoint2D P;
mxrPointXYZProject(&P,&camera,&T,200,241,12);       // project into point
```

Notes:

The projection implements only the linear part of the model and assumes that the radial distortion component is zero. To implement the full model, the final point must be warped using the radial distortion model.

See also:

mxrPoint3DProject, mxrCameraProject, mxrRadialDistortion

mxrPoint3DProject *Release Version:* MXRToolkit V1.0

Short Description:

Projects a list of three-dimensional point structure down into pixel co-ordinates in an image, using intrinsic parameters from a camera structure and a separate transformation matrix.

Function declaration:

```
void    mxrPoint3DProject(mxrPoint2D *pOut, mxrCamera *A, mxrTransform *T,
                                                    mxrPoint3D *Pin, int nPoints);
```

Parameter Interpretation

- `pOut` is a pointer to a list of two dimensional image point in pixels which are the result of the projection
- `camera` is a pointer to an mxrCamera structure which contains the intrinsic parameters used in the projection.
- `T` contains a pointer to a Euclidean transform structure representing the extrinsic matrix of the camera.
- `P` is a pointer to a list of three dimensional points which must be converted.
- `nPoints` is the number of points in the list.
-

Example:

```
mxrCamera camera;                                   // declare camera and fill values
mxrCameraDefault(&camera,640, 480);

mxrTransform T;                                     // declare transformation and fill
T.x = 240; T.y = 25; T.z = 25;

mxrPoint3D P3D[2];                                          // make list of 2 points
P[0].X = 200; P[0].Y = 234; P[0].Z = 24; P[0].W = 1;
P[1].X = 124; P[1].Y = 183; P[1].Z = 12; P[1].W = 1;

mxrPoint2D P2D[2];
mxrPoint3DProject(P2D,&camera,&T,P3D,2);                    // project list
```

Notes:

The project implements only the linear part of the model and assumes that the radial distortion component is zero. To implement the full model, the final point must be warped using the radial distortion model.

See also:

mxrPointXYZProject, mxrCameraProject, mxrRadialDistortion

mxrCameraProject *Release Version:* MXRToolkit V1.0

Short Description:

Projects a list of three-dimensional point structures down into pixel co-ordinates in an image, using intrinsic and extrinsic parameters from a camera structure.

Function declaration:

```
void mxrCameraProject(mxrPoint2D *pOut, mxrCamera *camera, mxrPoint3D *PIn, int nPoints);
```

Parameter Interpretation

- `pOut` is a pointer to a list of two dimensional image point in pixels which are the result of the projection
- `camera` is a pointer to an mxrCamera structure which contains the intrinsic and extrinsic parameters used in the projection.
- `PIn` is a pointer to a list of three dimensional points which must be converted.
- `nPoints` is the number of points in the list.

Example:

```
mxrCamera camera;                                    // declare camera and fill values
mxrCameraDefault(&camera,640, 480);
camera.T.x = 240; camera.T.y = 25; camera.T.z = 25;

mxrPoint3D P3D[2];                                          // make list of 2 points
P[0].X = 200; P[0].Y = 234; P[0].Z = 24; P[0].W = 1;
P[1].X = 124; P[1].Y = 183; P[1].Z = 12; P[1].W = 1;

mxrPoint2D P2D[2];
mxrPoint3DProject(P2D,&camera,&T,P3D,2);                    // project list
```

Notes:

The project implements only the linear part of the model and assumes that the radial distortion component is zero. To implement the full model, the final point must be warped using the radial distortion model. An alternative is to apply a projection matrix to the three-dimensional points

See also:

mxrPointXYZProject, mxrPoint3DProject, mxrRadialDistortion

void mxrCameraProject(mxrPoint2D *p, mxrCamera *A, mxrPoint3D *P, int nPoints);

NETWORK LIBRARY

The network library provides a simple interface for developing client-server applications. In particular, it provides a front end for Winsock V1.1 blocking sockets using the AF_INET protocol, which allows simple messages to be passed back and forth between a client and server. The main aim of the library is to allow people who do not understand what the previous sentence means to implement networked applications! Users with serious networking experience will probably prefer to implement their own networking code.

The library introduces only one new data type, mxrSocket. This is actually a implemented as a C++ class, but for simplicity, it can be considered as a simple structure with only two fields:

```
struct mxrSocket{
        char *mesg;
        int mesgLength;
}
```

where

- `mesg` is a pointer to a received message
- `mesglength` is the length of the received message in bytes

In order to connect two machines together, the client needs the server IP Address, and they must both agree on a Port Number to communicate on. If you don't know what this means, then stick to the number given in the example below.

The network library currently consists of the following routines:

- `mxrNetworkConnect` attempts to connect to a network server
- `mxrNetworkListen` initializes a network server routine
- `mxrNetworkSendMesg` sends a packaged message between a client and server who are communicating
- `mxrNetworkRecvMesg` receives a packaged message and fills in the mxrSocket structure with the data
- `mxrNetworkSendBytes` sends raw (unpackaged) bytes down an active connection
- `mxrNetworkReceiveBytes` receives raw(unpackaged) bytes from an active connection
- `mxrNetworkDelete` closes a connection with another machine
- `mxrNetworkIsConnected` returns the connection status of the current socket
- `mxrNetworkIsListening` returns the listening status of the current socket

CLIENT SERVER EXAMPLE

The simplest method to introduce the library is to provide a concrete example of a client and server application. We present a server program which receives a connection from the

client, receives a message from the client and returns a second message. We also present the corresponding client program.

SERVER PROGRAM

```
#include "mxrNetwork.h"
#include "stdio.h"

void main(int argc, char **argv){
        mxrSocket s;
        char reply[100];

        sprintf(reply,"This is the reply from the server\n");

        mxrNetworkListen(&s, "9099");                                   // listen on port 9099
        mxrNetworkReceiveMesg(&s);                                      // receive a message
        printf("%s",s.mesg);                                            // print out message
        mxrNetworkSendMesg(&s,strlen(reply)+1,reply);                   // send reply
}
```

CLIENT PROGRAM

```
#include "mxrNetwork.h"
#include "stdio.h"

void main(int argc, char **argv){
        mxrSocket s;
        char text[100];

        sprintf(text,"Sending test message \n");

        mxrNetworkConnect(&s, "127.0.0.1","9099");                      // connect to server
        mxrNetworkSendMesg(&s, strlen(text)+1,text);                    // send messsage
        mxrNetworkReceiveMesg(&s);                                      // receive reply
        printf("%s",s.mesg);\                                           // print reply
}
```

Notes:

As you can see, it is incredibly simple to set up network communication using this system. There are several things worth commenting on. Firstly, we send a message of strlen(text)+1 as the data is a null terminated string and we wish to include the termination character in our message. Secondly, we use the special IP address 127.0.0.1, which is a loopback to the current machine – so in this case both the client and the server run on the same machine. Thirdly, note that we never explicitly close the network connection or release (or declare!) the memory associated with the received memory. This is handled automatically and all the memory is deleted when the mxrSocket variable goes out of scope.

mxrNetworkConnect *Release Version:* MXRToolkit V1.0

Short Description:

This routine sets up a connection from a client to a server at a known IP address and connected to a known port number. The routine returns 1 if the connection is successful and zero if it fails.

Function declaration:

```
bool    mxrNetworkConnect(mxrSocket *s, char *hostIP, char *portNo);
```

Parameter Interpretation

- `s` is a pointer to an unused mxrSocket object
- `hostIP` is a c-style null-terminated string containing the IP address of the server
- `portNo` is a c-style null-terminated string containing the port number over which communication will take place.
- returns 1 if successful and zero if not.

Example:

```
mxrSocket s;

if (mxrNetworkConnect(&s, "192.162.1.2","9099")){
        printf("Connected to server\n");
}
else{
        printf("Connection Failed\n");
}

/* THIS EXAMPLE ATTEMPTS TO CONNECT TO A SERVER AT IP ADDRESS 192.162.1.2 OVER PORT NO
9099 AND PRINTS OUT THE STATUS OF THE CONNECTION */
```

Notes:

The connection uses the AF_INET protocol and sets up a blocking Berkeley socket in the background using WinSock1.1. The connection is automatically closed when the mxrSocket object goes out of scope.

See also:

mxrNetworkListen, mxrNetworkSendMesg, mxrNetworkRecvMesg

mxrNetworkDelete *Release Version:* MXRToolkit V1.0

Short Description:

Closes an active socket connection and frees any memory that is associated with messages already received.

Function declaration:

```
void    mxrNetworkDelete (mxrSocket *s);
```

Parameter Interpretation

- `s` is a pointer to a socket structure that has already been connected

Example:

```
mxrSocket s;
int n;


mxrNetworkListen(&s, "9099");
n = mxrNetworkReceiveMesg(&s);
mxrNetworkDelete(&s);


/* SETS UP A SERVER, WAITS FOR A CONNECTION AND READS MESSAGE.  THEN CLOSES CONNECTION
AND DELETES MESSAGE MEMORY/
```

Notes:

This function deletes the memory associated with the last received message, so this must be copied or processed before calling this function. In many cases it may not be necessary to call this function as these tasks are actually carried out automatically when the mxrSocket structure goes out of scope.

See also:

mxrNetworkListen, mxrNetworkConnect

mxrNetworkIsConnected *Release Version:* MXRToolkit V1.0

Short Description:

Returns the connection status of a socket – i.e. tells you whether is has successfully connected to another machine, or been connected to by a client.

Function declaration:

```
bool    mxrNetworkIsConnected (mxrSocket *s);
```

Parameter Interpretation

- `s` is a pointer to a socket structure
- `returns` 1 if the socket is in use and 0 if it is free

Example:

```
mxrSocket s;
bool connected;

connected =mxrNetworkIsConnected(&s);
mxrNetworkListen(&s, "9099");
connected = mxrNetworkIsConnected(&s);
mxrNetworkDelete(&s);
connected = mxrNetworkIsConnected(&s);

/* The three queries return 0, 1 and 0 respectively /
```

Notes:

A successful reply only reflects the status of this end of the connection – the other machine may leave the connection for any reason and this will not be reflected.

See also:

mxrNetworkConnect, mxrNetworkListen, mxrNetworkDelete

mxrNetworkListen *Release Version:* MXRToolkit V1.0

Short Description:

The routine mxrNetworkListen sets up a server and waits for a connection on a given port. The routine only returns when the connection is made. If the routine returns success, the server is ready to send and receive from the client.

Function declaration:

```
bool    mxrNetworkListen(mxrSocket *s,char *portName);
```

Parameter Interpretation

- `s` is a pointer to an unused mxrSocket structure
- `portName` is a c-style null terminated string containing the port number
- the function returns 1 or 0 depending on whether a client is or isn't successfully connected

Example:

```
mxrSocket s;
if (mxrNetworkListen(&s, "9099")){
        printf("Successful:  Connected to client on Port 9099\n");
}
else{
        printf("Failed to set up server\n");
};

/* THIS SETS UP A SERVER ON PORT 9099 AND WAITS FOR THE FIRST CONNECTION*/
```

Notes:

See the example code in the introduction to the mxrNetwork section for a complete client-server application.

See also:

mxrNetworkConnect, mxrNetworkSendMesg, mxrNetworkRecvMesg

mxrNetworkReceiveBytes *Release Version:* MXRToolkit V1.0

Short Description:

Assuming that an active connection has already been made between client and server, this routine receives a given number of bytes from the connection. Note that in contrast to mxrNetworkReceiveMesg, no information is automatically sent concerning the proper length of the message. See notes below

Function declaration:

```
bool    mxrNetworkReceiveBytes (mxrSocket *s, char *inBuffer, int nBytes);
```

Parameter Interpretation

- `s` is a pointer to a socket structure that has already been connected
- `inBuffer` is a pointer to the first byte of memory what the message will be stored
- `nBytes` is the number of bytes that will be read from the connection
- `returns` 1 if the message was received correctly and 0 if there was a problem.

Example:

```
mxrSocket s;
char buffer[4];


mxrNetworkListen(&s, "9099");
if (mxrNetworkReceiveBytes(&s,4,buffer){
        printf("Received 4 bytes\n");
}
else{
        printf("Failed to Receive Bytes\n");
}

/* SETS UP A SERVER, WAITS FOR A CONNECTION AND READS FOUR BYTES FROM THE CONNECTION/
```

Notes:

To use this function you must develop a proper protocol, since otherwise the program has no way of knowing how many bytes are being sent down the connection. Hence if you ask to read a longer set of bytes than the message is long, the command will wait forever. A simple solution to this, which is implements in mxrNetworkSend/Receive message is to send a fixed size header at the start of each message, which contains information about the number of bytes to follow.

Note also that the memory where the message is stored must be declared by the user. This is not handled automatically as in mxrNetworkReceiveMesg.

See also:

mxrNetworkSendBytes, mxrNetworkReceiveMesg

mxrNetworkReceiveMesg *Release Version:* MXRToolkit V1.0

Short Description:

Given a socket connection between a client and a server, this routine receives a message of from the connection. The code waits until the first message arrives and then grabs it. The message is preceded by a header including the number of bytes in the message so that an appropriate amount of memory can be allocated. The pointer `mesg` in the socket structure points to the first byte of the message.

Function declaration:

```
int     mxrNetworkReceiveMesg (mxrSocket *s);
```

Parameter Interpretation

- `s` is a pointer to a socket structure that has already been connected
- returns the number of bytes in the message (can be 0). The message itself is stored in s->message.

Example:

```
mxrSocket s;
int n;

mxrNetworkListen(&s, "9099");
n = mxrNetworkReceiveMesg(&s);

/* SETS UP A SERVER, WAITS FOR A CONNECTION AND RETRIEVES THE FIRST MESSAGE SENT
n CONTAINS THE NUMBER OF BYTES IN THE MESSAGE, WHICH IS FOUND AT S.mesg */
```

Notes:

See the introduction to the Network library for a complete example of a client and server application. The socket class itself handles the memory management – if there is not already enough memory allocated at `s.mesg` then more will be allotted. The memory is deleted when the socket structure goes out of scope or mxrNetworkDelete is called.

See also:

mxrNetworkSendMesg

mxrNetworkSendBytes *Release Version:* MXRToolkit V1.0

Short Description:

Assuming that an active connection has already been made between client and server, this routine sends a given number of bytes down the connection to the host. Note that in contrast to mxrNetworkSendMesg, no information is sent about the length of the message, so the user must develop their own protocol to ensure that the receiving end knows when the message is complete.

Function declaration:

```
bool    mxrNetworkSendBytes (mxrSocket *s, int nBytes, char *inBuffer);
```

Parameter Interpretation

- `s` is a pointer to a socket structure that has already been connected
- nBytes is the number of bytes to be transmitted
- inBuffer is a pointer to the first byte of the message to be transmitted.
- returns 1 if the message was sent correctly and 0 if there was a problem.

Example:

```
mxrSocket s;
char buffer[4];

buffer[0] = 1; buffer[1] = 2; buffer[2] = 24; buffer[3] = 6;

mxrNetworkListen(&s, "9099");
if (mxrNetworkSendBytes(&s,4,buffer){
        printf("Sent 4 bytes\n");
}
else{
        printf("Failed to Send Bytes\n");
}

/* SETS UP A SERVER, WAITS FOR A CONNECTION AND SENDS FOUR BYTES DOWN THE CONNECTION/
```

Notes:

Note that the successful sending of a message does not imply that it was received correctly.

See also:

mxrNetworkReceiveBytes, mxrNetworkSendMesg

mxrNetworkSendMesg *Release Version:* MXRToolkit V1.0

Short Description:

Given a socket connection between a client and a server, this routine sends a message of a given number of bytes down the connection. The message is preceded by a header including the number of bytes in the message so that the other end knows how many characters to expect in the message.

Function declaration:

```
bool    mxrNetworkSendMesg (mxrSocket *, int nBytes, char *data);
```

Parameter Interpretation

- `s` is a pointer to a socket structure that has already been connected
- `nBytes` is the number of bytes that are about to be sent in message
- `data` is a pointer to the message itself
- returns 1 if the message was sent successfully and 0 if there was a failure

Example:

```
mxrSocket s;
char message[4];

message[0] = 25; message[1] = 2; message[2] = 244; message[3] = 7;

mxrNetworkListen(&s, "9099");
mxrNetworkSendMesg(&s,4,message);

/* SETS UP A SERVER, WAITS FOR A CONNECTION AND SENDS A FOUR BYTE MESSAGE */
```

Notes:

See the introduction to the Network library for a complete example of a client and server application. Note that the sending of a message correctly does not imply that it was received correctly. It is possible to send a NULL message that is zero characters long, to denote a failed request.

See also:

mxrNetworkReceiveMesg

MATH LIBRARY

The math library contains a number of routines for implementing linear algebra and solving least square problems. The routines rely on an extremely simple matrix structure, which contains only three fields:

```
struct mxrMatrix{
      int x;
      int y;
      double *data;
}
```

The x field contains the number of columns in the matrix and the y field the number of rows. The field data contains a pointer to the matrix data. If the pointer is set to null then no data has been allocated yet and the previous two fields may be ignored.

NOTE CAREFULLY: data is stored in row order, so that data[1] is in the first row, but the second column. This differs from some other implementations of matrix structures, but makes the representation consistent with the mxrImage structure.

The memory for the data must be allocated using the mxrCreate command, but is deleted automatically when the structure goes out of scope if not manually deleted. In fact the mxrMatrix is a C++ class and the destructor checks whether the pointer is set to null and de-allocates the memory if it is not.

The math library currently contains the following routines:

- mxrMatrixCreate – allocates memory for data in matrix
- mxrMatrixDelete – frees memory for data part of matrix
- mxrMatrixInvert – inverts square matrix
- mxrMatrixMult – multiplies one matrix by another
- mxrMatrixPrint – prints out contents of matrix
- mxrMatrixTranspose – transposes matrix
- mxrMatrixLSSolve – solves least squares problem Ax = b
- mxrMatrixLSSolveNull – solves least squares problem Ax = 0
- mxrMatrixTranspose – calculates the transpose of a matrix
- mxrMatrixTransposeIP – calculates the transpose of a matrix in place
- mxrMatrixSVD – calculates the singular value decomposition of a matrix
- mxrMatrixCopy - copies the contents of one matrix to another

mxrMatrixCreate *Release Version:* MXRToolkit V1.0

Short Description:

The mxrMatrixCreate routine fills in the x and y size fields of the matrix structure and declares memory for the data.

Function declaration:

```
void    mxrMatrixCreate (mxrMatrix *M, int x, int y);
```

Parameter Interpretation

- `M` is a pointer to a matrix structure
- `x contains the x-size or number of columns in the matrix`
- `y contains the y-size of number of rows in the matrix`

Example:

```
mxrMatrix M;
mxrMatrixCreate(&M, 1, 6);                  // allocate a column vector, length 6

mxrMatrixDelete(&M);                        // delete memory associated with vector
```

Notes:

The memory that is created will be automatically destroyed when the matrix goes out of scope, although it is good practice to delete it by hand. Note that vectors are not handled differently from matrices and are simply 1 x n or n x 1 matrix structures.

See also:

mxrMatrixDelete

mxrMatrixDelete *Release Version:* MXRToolkit V1.0

Short Description:

This routine frees the memory associated with a matrix structure. Although the memory is deleted automatically when the matrix structure goes out of scope, it is good practice to automatically delete the object.

Function declaration:

```
void    mxrDelete (mxrMatrix *M);
```

Parameter Interpretation

- `M` is a pointer to an mxrMatrix structure

Example:

```
mxrMatrix M;
mxrMatrixCreate(&M, 1, 6);                  // allocate a column vector, length 6

mxrMatrixDelete(&M);                        // delete memory associated with vector
```

Notes:

If the pointer in the matrix structure is set to NULL then the routine will do nothing.

See also:

mxrMatrixCreate

mxrMatrixInvert *Release Version:* MXRToolkit V1.0

Short Description:

This routine inverts a square matrix stored in an mxrMatrix structure. The routine will fail and abort if the matrix is singular in the current implementation.

Function declaration:

```
void    mxrMatrixInvert(mxrMatrix *invM, mxrMatrix *M);
```

Parameter Interpretation

- `invM` is a pointer to an mxrMatrix structure which contains the inverted matrix
- `M` is a pointer to the original matrix to be inverted

Example:

```
mxrMatrix M;

mxrMatrixCreate (&M,2,2);                          // create memory for matrix structure
M.data[0] = 2; M.data[1] = 1;
M.data[2] = 0.5; M.data[3] = 1;

mxrMatrixInvert(&M,&M);                            // invert matrix
mxrMatrixPrint(&M);                                // print matrix contents
mxrMatrixDelete(&M);                               // delete matrix contents
```

Notes:

The matrix inversion can be carried out in place.

See also:

mxrMatrixTranspose, mxrMatrixSVD

mxrMatrixMult *Release Version:* MXRToolkit V1.0

Short Description:

The routine multiplies two matrices together. If the matrices are not the correct size then the routine will fail.

Function declaration:

```
void    mxrMatrixMult(mxrMatrix *AB, mxrMatrix *A, mxrMatrix *B);
```

Parameter Interpretation

- `AB` is a pointer to an mxrMatrix structure which will contain the product of the A and B matrices on return from the routine.
- `A` is a pointer to an mxrMatrix structure which contains the first matrix
- `B` is a pointer to an mxrMatrix structure which contains the second matrix

Example:

```
mxrMatrix A,B,AB;

mxrMatrixCreate (&A,3,2);                   // create memory for first matrix structure
A.data[0] = 1;   A.data[1] = 2;             // fill in values
A.data[2] = 1;   A.data[3] = 4;
A.data[4] = 0.4; A.data[5] = 6;

mxrMatrixCreate (&B,2,2);                   // create memory for second matrix structure
A.data[B] = 2; B.data[1] = 0;               // fill in some values
A.data[B] = 0; B.data[3] = 1;


mxrMatrixCreate(&AB,3,2);
mxrMatrixMult(&AB,A,B);                     // AB now contains product of matrices

mxrMatrixDelete(&AB);                       // free all the memory we have allocated
mxrMatrixDelete(&A);
mxrMatrixDelete(&B);
```

Notes:

Note that this implements MATRIX MULTIPLICATION – not point wise multiplication – the matrices must all be the correct size or the routine will cause the program to abort.

See also:

mxrMatrixTranspose

mxrMatrixPrint *Release Version:* MXRToolkit V1.0

Short Description:

This routine prints the contents of a matrix to stdio, formatted appropriately into rows and columns.

Function declaration:

```
void    mxrMatrixPrint(mxrMatrix *A);
```

Parameter Interpretation

- `A` is a pointer to an mxrMatrix structure which will be printed out

Example:

```
mxrMatrix A;

mxrMatrixCreate (&A,3,2);                   // create memory for matrix structure
A.data[0] = 1;   A.data[1] = 2;             // fill in values
A.data[2] = 1;   A.data[3] = 4;
A.data[4] = 0.4; A.data[5] = 6;

mxrMatrixPrint(&A);                         // print out contents of matrix

mxrMatrixDelete(&A);                        // free memory
```

Notes:

It is not recommended to print out the contents of large matrices as the formatting will wrap around and make the output hard to interpret.

See also:

mxrMatrixCreate, mxrMatrixDelete

mxrMatrixLSSolve *Release Version:* MXRToolkit V1.0

Short Description:

Least squares solution of overdetermined linear equations. Solves a system of equations, ***Ax*** = **b** for the vector **x**, using a least squares criterion, where ***A*** is an (m x n) matrix, **x** is an (n x 1) vector and **b** is an (n x 1) vector, and m>=n (i.e. number of equations is greater or equal to number of unknowns).

Function declaration:

```
void    mxrMatrixLSSolve(mxrMatrix *x, mxrMatrix *A, mxrMatrix *b);
```

Parameter Interpretation

- `x` is a pointer to an mxrMatrix structure which will contain the least squares solution in column vector form on return.
- *A* is a pointer to an mxrMatrix structure which contains the matrix ***A***
- b is a pointer to an mxrMatrix structure which contains the column vector **b**

Example:

```
mxrMatrix A,b,x;

mxrMatrixCreate (&A,3,2);                   // create memory for matrix A structure
A.data[0] = 1;   A.data[1] = 2;             // fill in values
A.data[2] = 1;   A.data[3] = 4;
A.data[4] = 0.4; A.data[5] = 6;

mxrMatrixCreate(&B,2,1);                    // create memory for vector b structure
A.data[0] = 2; A.data[1] = 0.1;             // fill in values

mxrMatrixCreate(&x,2,1);                    // create memory for solution vector

mxrMatrixLSSolve(&x,&A,&b);                 // solve system of equations

mxrMatrixDelete(&A);                        // free memory
mxrMatrixDelete(&b);
mxrMatrixDelete(&x);
```

Notes:

This routine simply implements the standard overdetermined solution **x** = $(\boldsymbol{A}'\boldsymbol{A})^{-1}\boldsymbol{A}'\mathbf{b}$. The memory for the solution vector must be declared on entry to the routine. The routine will fail if the solution is not unique (i.e. some of the equations are the same…).

See also:

mxrMatrixInvert, mxrMatrixLSSolveNull

mxrMatrixLSSolveNull *Release Version:* MXRToolkit V1.0

Short Description:

Least squares solution of overdetermined linear equations. Solves a system of equations, ***Ax*** = **0** for the vector **x**, using a least squares criterion, where ***A*** is an (m x n) matrix, **x** is a (n x 1) vector of zeros and **b** is an (n x 1) vector, and m>=n (i.e. number of equations is greater or equal to number of unknowns).

Function declaration:

```
void    mxrMatrixLSSolveNull(mxrMatrix *x, mxrMatrix *A);
```

Parameter Interpretation

- `x` is a pointer to an mxrMatrix structure which will contain the least squares solution in column vector form on return.
- *A* is a pointer to an mxrMatrix structure which contains the matrix ***A***

Example:

```
mxrMatrix A,x;

mxrMatrixCreate (&A,3,2);                   // create memory for matrix A structure
A.data[0] = 1;   A.data[1] = 2;             // fill in values
A.data[2] = 1;   A.data[3] = 4;
A.data[4] = 0.4; A.data[5] = 6;


mxrMatrixCreate(&x,2,1);                    // create memory for solution vector

mxrMatrixLSSolve(&x,&A,&b);                 // solve system of equations

mxrMatrixDelete(&A);                        // free memory
mxrMatrixDelete(&x);
```

Notes:

The solution to this system of equations is calculated via the singular value decomposition and is the eigenvector of the matrix ***A*** corresponding to the smallest eigenvalue. Since the solution is inherently ambiguous up to scale, we choose an arbitrary scaling factor which ensures that the sum of the square of the elements of **x** is unity.

See also:

mxrMatrixSVD, mxrMatrixLSSolve

mxrMatrixTranspose *Release Version:* MXRToolkit V1.0

Short Description:

Takes transpose of matrix and puts it into another matrix structure.

Function declaration:

```
void    mxrMatrixTranspose(mxrMatrix *AT, mxrMatrix *A);
```

Parameter Interpretation

- `AT` is a pointer to an mxrMatrix structure which will contain the transpose of the matrix in question on return.
- *A* is a pointer to an mxrMatrix structure which contains the matrix ***A*** which we want to transpose.

Example:

```
mxrMatrix A,AT;

mxrMatrixCreate (&A,3,2);                   // create memory for matrix A structure
A.data[0] = 1;   A.data[1] = 2;             // fill in values
A.data[2] = 1;   A.data[3] = 4;
A.data[4] = 0.4; A.data[5] = 6;

mxrMatrixCreate (&AT,2,3);                  // create memory for transpose

mxrMatrixTranspose (&AT,&A);                // take transpose

mxrMatrixDelete(&A);                        // free memory
mxrMatrixDelete(&AT);
```

Notes:

This operation assumes that the memory for the destination matrix has already been declared and will fail if the destination is not the appropriate sizes. Transposition cannot ta be done in place even when the matrix is square – see the routine mxrMatrixTransposeIP for this functionality.

See also:

mxrMatrixTransposeIP

mxrMatrixTransposeIP *Release Version:* MXRToolkit V1.0

Short Description:

Takes transpose of square matrix. Operates in place (i.e. source and destination are the same).

Function declaration:

```
void    mxrMatrixTransposeIP(mxrMatrix *A);
```

Parameter Interpretation

- `A` is a pointer to an mxrMatrix structure which contains the matrix upon entering routine and the transposed matrix upon return.

Example:

```
mxrMatrix A;

mxrMatrixCreate (&A,2,2);                   // create memory for matrix A structure
A.data[0] = 1;   A.data[1] = 2;             // fill in values
A.data[2] = 1;   A.data[3] = 4;

mxrMatrixTransposeIP (&A);                  // take transpose

mxrMatrixDelete(&A);                        // free memory
```

Notes:

This routine only functions when the matrix is square. For non-square matrices, the operation cannot currently be performed in place, and must be peformed using mxrMatrixTranspose.

See also:

mxrMatrixTranspose

mxrMatrixTransposeSVD *Release Version:* MXRToolkit V1.0

Short Description:

Calculates the singular value decomposition of an (m x n) matrix. Returns **U**, an (m x m) rotation matrix containing the left eigenvectors, **L** an (m x n) diagonal matrix containing the singular values and **V**, an (n x n) rotation matrix containing the right eigenvectors, such that **A** = **ULV'**.

Function declaration:

```
void    mxrMatrixSVD(mxrMatrix *U, mxrMatrix *L, mxrMatrix *V, mxrMatrix *A);
```

Parameter Interpretation

- `U` is a pointer to an mxrMatrix structure which will contain the U matrix on return
- `L` is a pointer to an mxrMatrix structure which will contiain the L matrix on return
- `V` is a pointer to an mxrMatrix structure which will contain the V matrix on return
- `A` is a pointer to the source matrix from which the SVD is calculated.

Example:

```
mxrMatrix A,U,V,L;

mxrMatrixCreate (&A,3,2);                   // create memory for matrix A structure
A.data[0] = 1;   A.data[1] = 2;             // fill in values
A.data[2] = 1;   A.data[3] = 4;
A.data[3] = 1;   A.data[4] = 1;

mxrMatrixCreate(&U,3,3);                    // allocate memory for SVD components
mxrMatrixCreate(&L,3,2);
mxrMatrixCreate(&V,2,2);

mxrMatrixSVD(&U,&L,&V,&A);                  // calculate singular value decomposition

mxrMatrixDelete(&U);
mxrMatrixDelete(&L);
mxrMatrixDelete(&V);
mxrMatrixDelete(&A);                        // free memory
```

Notes:

Note that the SVD always exists buy may be non-unique if some of the singular values are zero. Note also that the SVD routine is destructive to the original matrix – if you need to maintain the original matrix, take a copy using mxrMatrixCopy before applying this routine.

See also:

mxrMatrixInverse, mxrMatrixLSSolveNull

mxrMatrixCopy *Release Version:* MXRToolkit V1.0

Short Description:

Copies the contents of one matrix in to another matrix of the same size.

Function declaration:

```
void    mxrMatrixCopy(mxrMatrix *copy, mxrMatrix *orig);
```

Parameter Interpretation

- `copy` is a pointer to an mxrMatrix structure which will contain the original matrix on return
- `orig` is a pointer to an mxrMatrix structure

Example:

```
mxrMatrix orig,copy;

mxrMatrixCreate (&orig,3,2);                // create memory for matrix A structure
orig.data[0] = 1;   orig.data[1] = 2;       // fill in values
orig.data[2] = 1;   orig.data[3] = 4;
orig.data[3] = 1;   orig.data[4] = 1;

mxrMatrixCreate(&copy,3,2);                 // declare memory for copy

mxrMatrixCopy (&copy,&orig);                // copy orig into copy

mxrMatrixDelete(&copy);
mxrMatrixDelete(&orig);                             // free memory
```

Notes:

Note that the destination memory must be pre-declared and the x any y sizes of the source and destination matrix must be the same.d

See also:

mxrMatrixInverse, mxrMatrixLSSolveNull
// copies contents of one matrix to another

MEDIA LIBRARY

The media library is concerned with loading, and displaying media clips in the real world. These may be static 2d pictures, 2d movies, 3d models or even sounds. The aim of the media library is to provide a constant and simple interface to displaying these types of models. The interface is remarkably simple – when the object is loaded into the program, a handle to the object is returned. The handle is then assigned to media objects as required. This is to allow the user to display multiple instances of the same object by loading the object only once. When the user wishes to render this object he passes back the media objects together with the respective transformation matrices corresponding to where in the scene the object should be rendered. A single media object stores information like animation speed, loop status, handle, starting frame, current frame, last frame, total number of frames, current time, and last updated time. They are encapsulated using the following mxrMedia structure:

```
struct mxrMedia {
        mxrMediaHandle          handle;                         // handle for graphics object

        int                     numOfFrames;
        float                   speed;
        mxrLoopStatus           loopStatus;
        float                   loopStartFrame;
        float                   loopEndFrame;
        float                   currentFrame;


        mxrTransform            tInstance;
        mxrRotVec               r;
        mxrPoint3D              t;
        mxrPoint3D              s;

        …
        float                   currentTime;
        float                   lastTime;

};
```

Objects which consist of animation like the Quake MD2 and MD3 objects can be displayed in a scene. The animation can be displayed in 3 modes:

- infinite loop
- play through once
- play through once and hold the last frame

This looping format is encoded using the mxrLoopStatus enumeration:

```
typedef enum mxrLoopStatus{
        MXR_LOOP_FOREVER,
        MXR_LOOP_ONCE_THROUGH,
        MXR_LOOP_ONCE_AND_FREEZE,
        MXR_FINISHED,
};
```

MXR_FINISHED indicates that the animation has reached the end.

Current media types which are handled by the library are VRML 2.0 files, JPG files, BMP files, Targa files, WAV audio files, MIDI audio files, Lightwave 3D files, Quake MD2 files,

Maya OBJ files, 3D Studio Max ascii scene export files, 3d Studio Max .3ds files and MilkShape 3D files. The media format is encoded using the mediaType enumeration:

```
typedef enum mxrMediaType{
	MXR_MEDIA_VRML,
	MXR_MEDIA_JPG,
	MXR_MEDIA_BMP,
	MXR_MEDIA_TGA,
	MXR_MEDIA_WAV,
	MXR_MEDIA_MIDI,
	MXR_MEDIA_LWO,
	MXR_MEDIA_MD2,
	MXR_MEDIA_OBJ,
	MXR_MEDIA_ASE,
	MXR_MEDIA_MS3D,
	MXR_MEDIA_3DS,
};
```

There is a slight complication to displaying media in augmented reality – a given three-dimensional model may have been designed in any arbitrary system of units, and may be positioned anywhere in 3d space, and may be oriented at any angle. In order to display the object on the surface, we must pre-transform the objects to avoid these problems. Since this is generally a one-time problem for each object, and we address it by saving a file to disk which is associated with each media file. This has the file extensions .medInfo. It will be created automatically the first time the media file is loaded. The scaling and alignment of the object can be set using the accompanying alignment utility.

The routines in the media library are as follows:

- mxrMediaRead – reads in a media file and supplies a unique handle for manipulating the media item.
- mxrMediaSet – assigns a handle to a media object
- mxrMediaSetLoop – sets the loop parameters of a media object
- mxrMediaSetSpeed – sets the speed of the animation of a media object
- mxrMediaSetFrame – sets the current frame of the animation of a media object
- mxrMediaGetLoopStatus – returns the current loop status
- mxrMediaPrintLoopStatus – prints the current loop status
- mxrMediaRotate – rotates a media object
- mxrMediaScale – scales a media object
- mxrMediaTranslate – translates a media object
- mxrMediaSetTranslationXYZ – sets the absolute translational value in the 3 axis
- mxrMediaSetTranslationX – sets the absolute translational value in the X-axis
- mxrMediaSetTranslationY – sets the absolute translational value in the Y-axis
- mxrMediaSetRotationY – sets the absolute rotational value about Y-axis
- mxrMediaSetRotationZ – sets the absolute rotational value about Z-axis
- mxrMediaSetScaling – sets the absolute scaling value of a media object
- mxrMediaGetRotation – returns current prior rotation settings of a media object
- mxrMediaGetScaling – – returns current prior scaling settings of a media object
- mxrMediaGetTranslation – returns current prior translation settings of a media object
- mxrMediaRender – renders a media file at a particular point in space
- mxrMediaFree – frees memory associated with media file

- mxrMediaFreeAll – frees all memory associated with media files
- mxrMediaSetPreRotation – sets the initial rotational values for a handle
- mxrMediaSetPreScaling – sets the initial scaling values for a handle
- mxrMediaSetPreTranslation – sets the initial translational values for a handle
- mxrMediaGetPreRotation – returns the initial rotational value
- mxrMediaGetPreScaling – returns the initial scaling value
- mxrMediaGetPreTranslation – returns the initial translational value

-

mxrMediaRead *Release Version:* MXRToolkit V1.0

Short Description:

Reads in a media file and returns a unique handle with which the media can be referred to in the future.

Function declaration:

```
int mxrMediaRead(mxrMediaHandle *handle, char *filename, char *texName);
```

Parameter Interpretation

- `handle` is a pointer to an mxrMediaHandle, which is a unique integer that is with this media file.
- `filename` is a pointer to a c-style null terminated character array containing name of the media file to be loaded in - the type of media file is determined by the file extension.
- `texName is` a pointer to a c-style null terminated character array containing the name of the texture file to be loaded in
- returns an integer where 1 denotes a successful read and zero denotes a fail.

Example:

```
mxrMediaHandle H;

if (mxrMediaRead(&H,"snoman.wrl",NULL)){
        printf("Successfully loaded media\n");
}
else{
        printf("Media failed to load\n");
}
mxrMediaFree(H);
```

Notes:

The texture name field is for three-dimensional formats such as .md2 where the texture file name is not explicitly mentioned in the geometry file.

See also:

mxrMediaFree, mxrMediaRender

mxrMediaSetPreRotation *Release Version:* MXRToolkit V1.0

Short Description:

Sets pre-rotation for a media file. This rotation is applied prior to any user transforms each time an instance of the media file is displayed.

Function declaration:

```
void mxrMediaSetPreRotation(mxrMediaHandle handle, mxrFloat x, mxrFloat y, mxrFloat z);
```

Parameter Interpretation

- `handle` is a handle to an mxrMedia object that has already been loaded in
- `x` is the first component of the rotation vector
- `y` is the second component of the rotation vector
- `z` is the third component of the rotation vector

Example:

```
mxrMediaHandle H;                           // declare media handle
mxrMediaRead(&H,"snoman.wrl",NULL);         // read media object
mxrMediaSetPreRotation (H,1.57,0.0,0.0);    // change rotation to 90 deg about x axis
mxrMediaInfoWrite(H);
mxrMediaFreeAll();
```

Notes:

This prior rotation is part of the "media info" file.

See also:

mxrMediaSetPreScaling, mxrMediaSetPreTranslation, mxrMediaGetPreRotation

mxrMediaSetPreScaling *Release Version:* MXRToolkit V1.0

Short Description:

Sets pre-scaling for a media file. This scaling is applied prior to any user transforms each time an instance of the media file is displayed.

Function declaration:

```
void mxrMediaSetPreScaling (mxrMediaHandle handle, mxrFloat x, mxrFloat y, mxrFloat z);
```

Parameter Interpretation

- `handle` is a handle to an mxrMedia object that has already been loaded in
- `x` is the scaling factor in the x direction
- `y` is the scaling factor in the y direction
- `z` is the scaling factor in the z direction

Example:

```
mxrMediaHandle H;                              // declare media handle
mxrMediaRead(&H,"snoman.wrl",NULL);            // read media object
mxrMediaSetPreScaling (H, 10.0 ,10.0 ,10.0); // change scaling to uniform  x 10
mxrMediaInfoWrite(H);
mxrMediaFreeAll();
```

Notes:

This prior scaling is part of the "media info" file.

See also:

mxrMediaSetPreRotation, mxrMediaSetPreTranslation, mxrMediaGetPreScaling

mxrMediaSetPreTranslation *Release Version:* MXRToolkit V1.0

Short Description:

Sets pre-translation for a media file. This translation is applied prior to any user transforms each time an instance of the media file is displayed.

Function declaration:

```
void mxrMediaSetPreTranslation(mxrMediaHandle handle, mxrFloat x, mxrFloat y, mxrFloat z);
```

Parameter Interpretation

- `handle` is a handle to an mxrMedia object that has already been loaded in
- `x` is the first component of the translation vector
- `y` is the second component of the translation vector
- `z` is the third component of the translation vector

Example:

```
mxrMediaHandle H;                              // declare media handle
mxrMediaRead(&H,"snoman.wrl",NULL);            // read media object
mxrMediaSetPreTranslation (H,40.0,0.0,0.0); // change rotation to 90 deg about x axis
mxrMediaInfoWrite(H);
mxrMediaFreeAll();
```

Notes:

This prior translation is part of the "media info" file.

See also:

mxrMediaSetPreScaling, mxrMediaSetPreRotation, mxrMediaGetPreTranslation

mxrMediaGetPreRotation *Release Version:* MXRToolkit V1.0

Short Description:

Gets pre-rotation associated with media file. This rotation is applied prior to any user transforms each time an instance of the media file is displayed.

Function declaration:

```
void mxrMediaGetPreTranslation(mxrFloat *x,mxrFloat *y,mxrFloat *z,mxrMediaHandle handle);
```

Parameter Interpretation

- `x` is a pointer to a double, where the first component of the rotation vector will be stored
- `y` is a pointer to a double, where the second component of the rotation vector will be stored
- `z` is a pointer to a double, where the third component of the rotation vector will be stored
- `handle` is a handle to an mxrMedia object that has already been loaded in

Example:

```
mxrMediaHandle H;                          // declare media handle
mxrMediaRead(&H,"snoman.wrl",NULL);        // read media object

mxrFloat X,Y,Z;
mxrMediaGetPreRotation (H,&X,&Y,&Z);  // retrieve rotation vector
mxrMediaInfoWrite(H);
mxrMediaFreeAll();
```

Notes:

This prior rotation is part of the "media info" file.

See also:

mxrMediaGetPreScaling, mxrMediaGetPreTranslation, mxrMediaSetPreRotation

mxrMediaGetPreScaling *Release Version:* MXRToolkit V1.0

Short Description:

Gets pre-translation associated with media file. This translation is applied prior to any user transforms each time an instance of the media file is displayed.

Function declaration:

```
void mxrMediaGetPreScaling(mxrFloat *x,mxrFloat *y,mxrFloat *z,mxrMediaHandle handle);
```

Parameter Interpretation

- `x` is a pointer to a double, where the first component of the scaling vector will be stored
- `y` is a pointer to a double, where the second component of the scaling vector will be stored
- `z` is a pointer to a double, where the third component of the scaling vector will be stored
- `handle` is a handle to an mxrMedia object that has already been loaded in

Example:

```
mxrMediaHandle H;                           // declare media handle
mxrMediaRead(&H,"snoman.wrl",NULL);         // read media object

mxrFloat X,Y,Z;
mxrMediaGetPreScaling (H,&X,&Y,&Z);   // retrieve scaling vector
mxrMediaInfoWrite(H);
mxrMediaFreeAll();
```

Notes:

This prior scaling is part of the "media info" file.

See also:

mxrMediaGetPreTranslation, mxrMediaGetPreRotation, mxrMediaSetPreScaling

mxrMediaGetPreTranslation *Release Version:* MXRToolkit V1.0

Short Description:

Gets pre-translation associated with media file. This translation is applied prior to any user transforms each time an instance of the media file is displayed.

Function declaration:

```
void mxrMediaGetPreTranslation(mxrFloat *x,mxrFloat *y,mxrFloat *z,mxrMediaHandle handle);
```

Parameter Interpretation

- `x` is a pointer to a double, where the first component of the translation vector will be stored
- `y` is a pointer to a double, where the second component of the translation vector will be stored
- `z` is a pointer to a double, where the third component of the translation vector will be stored
- `handle` is a handle to an mxrMedia object that has already been loaded in

Example:

```
mxrMediaHandle H;                          // declare media handle
mxrMediaRead(&H,"snoman.wrl",NULL);        // read media object

mxrFloat X,Y,Z;
mxrMediaGetPreTranslation (H,&X,&Y,&Z);          // retrieve translation vector
mxrMediaInfoWrite(H);
mxrMediaFreeAll();
```

Notes:

This prior translation is part of the "media info" file.

See also:

mxrMediaGetPreScaling, mxrMediaGetPreRotation, mxrMediaSetPreTranslation

mxrMediaSet *Release Version:* MXRToolkit V1.0

Short Description:

This assigns a handle to a media object which can be referred to in the future.

Function declaration:

```
void mxrMediaSet(mxrMedia *media, mxrMediaHandle handle);
```

Parameter Interpretation

- `media` is a pointer to an mxrMedia, which is a structure that stores essential information of a media object like animation speed, loop status, handle, starting frame, current frame, next frame, last frame, total number of frames, current time, elapsed time and last stored time.
- `handle` is a pointer to an mxrMediaHandle, which is a unique integer that is with this media file.

Example:

```
mxrMediaHandle H;
mxrMedia       mediaObj1;
mxrMedia       mediaObj2;

mxrMediaRead(&H,"snoman.wrl",NULL);
mxrMediaSet(&mediaObj1,H);
mxrMediaSet(&mediaObj2,H);
```

Notes:

One handle can be assigned to as many media as desired. Each media can hold different information of how the object can be displayed. Hence, it is possible to display multiple instances of the same object doing different animation.

See also:

mxrMediaSetLoop, mxrMediaSetSpeed, mxrMediaSetFrame

mxrMediaSetLoop *Release Version:* MXRToolkit V1.0

Short Description:

This sets the loop status, starting frame and last frame for an animation of a media.

Function declaration:

```
void mxrMediaSetLoop((mxrMedia *media, mxrLoopStatus loop, int startFrame, int endFrame);
```

Parameter Interpretation

- `media` is a pointer to an mxrMedia, which is a structure that stores essential information of a media object like animation speed, loop status, handle, starting frame, current frame, last frame, total number of frames, current time and last stored time.
- `loop` is an enumeration consist of MXR_LOOP_FOREVER, MXR_LOOP_ONCE_THROUGH, MXR_LOOP_ONCE_AND_FREEZE
- `startFrame` is a floating point which holds the value of the first frame of the animation to be displayed.
- `endFrame` is a floating point which holds the value of the last frame of the animation to be displayed.

Example:

```
mxrMediaHandle H;
mxrMedia       mediaObj;

mxrMediaRead(&H,"snoman.wrl",NULL);
mxrMediaSet(&mediaObj,H);
mxrMediaSetLoop(&mediaObj,MXR_LOOP_FOREVER,0,99);
```

Notes:

MXR_LOOP_FOREVER - the animation will be looped infinitely
MXR_LOOP_ONCE_THROUGH – the animation will be played only once
MXR_LOOP_ONCE_AND_FREEZE – the animation will be played once and the last frame will be held.

See also:

mxrMediaSet, mxrMediaSetSpeed, mxrMediaSetFrame

mxrMediaSetSpeed *Release Version:* MXRToolkit V1.0

Short Description:

This sets the animation speed of a media.

Function declaration:

```
void mxrMediaSetSpeed(mxrMedia *media, float speed);
```

Parameter Interpretation

- `media` is a pointer to an mxrMedia, which is a structure that stores essential information of a media object like animation speed, loop status, handle, starting frame, current frame, last frame, total number of frames, current time and last stored time.
- `speed` is a float point which holds the value of the speed defined by the user.

Example:

```
mxrMediaHandle H;
mxrMedia       mediaObj;

mxrMediaRead(&H,"snoman.wrl",NULL);
mxrMediaSet(&mediaObj,H);
mxrMediaSetLoop(&mediaObj,MXR_LOOP_FOREVER,0,99);
mxrMediaSetSpeed(&mediaObj, 5.25);
```

Notes:

If the speed is set to 0.0, the animation stops moving.

See also:

mxrMediaSet, mxrMediaSetLoop, mxrMediaSetFrame

mxrMediaSetFrame *Release Version:* MXRToolkit V1.0

Short Description:

This sets the current frame of the animation to be displayed.

Function declaration:

```
void mxrMediaSetFrame(mxrMedia *media, int frame);
```

Parameter Interpretation

- `media` is a pointer to an mxrMedia, which is a structure that stores essential information of a media object like animation speed, loop status, handle, starting frame, current frame, last frame, total number of frames, current time and last stored time.
- `frame` is a floating point which holds the value of the current frame defined by the user.

Example:

```
mxrMediaHandle H;
mxrMedia       mediaObj;

mxrMediaRead(&H,"snoman.wrl",NULL);
mxrMediaSet(&mediaObj,H);
mxrMediaSetLoop(&mediaObj,MXR_LOOP_FOREVER,45.5,89.9);
mxrMediaSetSpeed(&mediaObj, 5.25);
mxrMediaSetFrame(&mediaObj, 10.0);
```

Notes:

- In the example above, the animation will start playing from frame 10.0 to frame 89.9. Then it starts again from frame 45.5 to frame 89.9.
- `frame` must be a value ranging from `0` to the `endFrame`.

See also:

mxrMediaSet, mxrMediaSetLoop, mxrMediaSetSpeed

mxrMediaGetLoopStatus *Release Version:* MXRToolkit V1.0

Short Description:

This returns the loop status of an animation of a media.

Function declaration:

```
mxrLoopStatus mxrMediaGetLoopStatus(mxrMedia *media);
```

Parameter Interpretation

- `media` is a structure that stores essential information of a media object like animation speed, loop status, handle, starting frame, current frame, last frame, total number of frames, current time and last stored time.

Example:

```
mxrMediaHandle H;
mxrMedia       mediaObj;

mxrTransform T;                                    // declare transformation matrix
mxrTransformLoadIdentity(&T);                      // fill in values
T.tz = 600;

mxrMediaRead(&H,"snoman.wrl",NULL);
mxrMediaSet(&mediaObj,H);
mxrMediaSetLoop(&mediaObj,MXR_LOOP_FOREVER,45,90);
mxrMediaSetSpeed(&mediaObj, 5.25);
mxrMediaSetFrame(&mediaObj, 10);
mxrMediaRender(&mediaObj,&T);

mxrLoopStatus S = mxrMediaGetLoopStatus(&mediaObj);
printf("%s\n",S);

//Output:
//MXR_LOOP_FOREVER
```

Notes:

There are 3 possible outputs corresponding to the 3 different modes of looping animation.

- Animation is in an indefinte loop
- Animation is playing once only
- Animation is playing once and the last frame will be held still

See also:

mxrMediaSetLoop, mxrMediaPrintLoopStatus

mxrMediaPrintLoopStatus *Release Version:* MXRToolkit V1.0

Short Description:

This prints the loop status of an animation of a media in the output.

Function declaration:

```
void mxrMediaPrintLoopStatus(mxrMedia *media);
```

Parameter Interpretation

- `media` is a structure that stores essential information of a media object like animation speed, loop status, handle, starting frame, current frame, last frame, total number of frames, current time and last stored time.

Example:

```
mxrMediaHandle H;
mxrMedia       mediaObj;

mxrTransform T;                                    // declare transformation matrix
mxrTransformLoadIdentity(&T);                      // fill in values
T.tz = 600;

mxrMediaRead(&H,"snoman.wrl",NULL);
mxrMediaSet(&mediaObj,H);
mxrMediaSetLoop(&mediaObj,MXR_LOOP_FOREVER,45,90);
mxrMediaSetSpeed(&mediaObj, 5.25);
mxrMediaSetFrame(&mediaObj, 10);
mxrMediaRender(&mediaObj,&T);

mxrMediaPrintLoopStatus(&mediaObj);

//Output:
//Animation is in an indefinte loop
```

Notes:

There are 3 possible outputs corresponding to the 3 different modes of looping animation.

- Animation is in an indefinte loop
- Animation is playing once only
- Animation is playing once and the last frame will be held still

See also:

mxrMediaSetLoop, mxrMediaGetLoopStatus

mxrMediaRotate *Release Version:* MXRToolkit V1.0

Short Description:

This sets the rotational value for an instance of a media file.

Function declaration:

```
void mxrMediaRotate(mxrMedia *media, mxrFloat x, mxrFloat y, mxrFloat z);
```

Parameter Interpretation

- `handle` is a handle to an mxrMedia object that has already been loaded in
- `x` is the first component of the rotation vector
- `y` is the second component of the rotation vector
- `z` is the third component of the rotation vector

Example:

```
mxrMediaHandle H;
mxrMedia       mediaObj;

mxrMediaRead(&H,"snoman.wrl",NULL);
mxrMediaSet(&mediaObj,H);
mxrMediaSetLoop(&mediaObj,MXR_LOOP_FOREVER,45,90);
mxrMediaSetSpeed(&mediaObj, 5.25);
mxrMediaSetFrame(&mediaObj, 10);

mxrMediaRotate(&mediaObj,1.57,0,0);
mxrMediaRender(&mediaObj,&T);
```

Notes:

This prior rotation is NOT part of the "media info" file. Different instances of the same media file can have different rotational values.

See also:

mxrMediaScale, mxrMediaTranslate, mxrMediaGetRotation

mxrMediaScale *Release Version:* MXRToolkit V1.0

Short Description:

This sets the scaling value for an instance of a media file.

Function declaration:

```
void mxrMediaScale(mxrMedia *media, mxrFloat x,mxrFloat y,mxrFloat z);
```

Parameter Interpretation

- `handle` is a handle to an mxrMedia object that has already been loaded in
- `x` is the scaling factor in the x direction
- `y` is the scaling factor in the y direction
- `z` is the scaling factor in the z direction

Example:

```
mxrMediaHandle H;
mxrMedia       mediaObj;

mxrMediaRead(&H,"snoman.wrl",NULL);
mxrMediaSet(&mediaObj,H);
mxrMediaSetLoop(&mediaObj,MXR_LOOP_FOREVER,45,90);
mxrMediaSetSpeed(&mediaObj, 5.25);
mxrMediaSetFrame(&mediaObj, 10);

mxrMediaScale(&mediaObj,10,10,10);
mxrMediaRender(&mediaObj,&T);
```

Notes:

This prior rotation is NOT part of the "media info" file. Different instances of the same media file can have different scaling values.

See also:

mxrMediaRotate, mxrMediaTranslate, mxrMediaGetScaling

mxrMediaTranslate *Release Version:* MXRToolkit V1.0

Short Description:

This sets the translational value for an instance of a media file.

Function declaration:

```
void mxrMediaTranslate(mxrMedia *media, mxrFloat x, mxrFloat y, mxrFloat z);
```

Parameter Interpretation

- `handle` is a handle to an mxrMedia object that has already been loaded in
- `x` is the first component of the translation vector
- `y` is the second component of the translation vector
- `z` is the third component of the translation vector

Example:

```
mxrMediaHandle H;
mxrMedia       mediaObj;

mxrMediaRead(&H,"snoman.wrl",NULL);
mxrMediaSet(&mediaObj,H);
mxrMediaSetLoop(&mediaObj,MXR_LOOP_FOREVER,45,90);
mxrMediaSetSpeed(&mediaObj, 5.25);
mxrMediaSetFrame(&mediaObj, 10);

mxrMediaTranslate(&mediaObj,100,-50,20);
mxrMediaRender(&mediaObj,&T);
```

Notes:

This prior rotation is NOT part of the "media info" file. Different instances of the same media file can have different translational values.

See also:

mxrMediaScale, mxrMediaTranslate, mxrMediaGetTranslation

mxrMediaSetTranslationXYZ *Release Version:* MXRToolkit V1.0

Short Description:

This sets the absolute translational value for an instance of a media file in the X, Y and Z axis.

Function declaration:

```
void mxrMediaSetTranslationXYZ(mxrMedia *media, mxrFloat x, mxrFloat y, mxrFloat z);
```

Parameter Interpretation

- `handle` is a handle to an mxrMedia object that has already been loaded in
- `x` is the first component of the translation vector
- `y` is the second component of the translation vector
- `z` is the third component of the translation vector

Example:

```
mxrMediaHandle H;
mxrMedia       mediaObj;

mxrMediaRead(&H,"snoman.wrl",NULL);
mxrMediaSet(&mediaObj,H);
mxrMediaSetLoop(&mediaObj,MXR_LOOP_FOREVER,45,90);
mxrMediaSetSpeed(&mediaObj, 5.25);
mxrMediaSetFrame(&mediaObj, 10);

mxrMediaSetTranslationXYZ(&mediaObj,60.0,60.0,0.0);
mxrMediaRender(&mediaObj,&T);
```

Notes:

This prior rotation is NOT part of the "media info" file. Different instances of the same media file can have different absolute translational values.

See also:

mxrMediaSetTranslationX, mxrMediaSetTranslationY, mxrMediaTranslate

mxrMediaSetTranslationX *Release Version:* MXRToolkit V1.0

Short Description:

This sets the absolute translational value for an instance of a media file in the X axis.

Function declaration:

```
void mxrMediaSetTranslationXYZ(mxrMedia *media, mxrFloat x);
```

Parameter Interpretation

- `handle` is a handle to an mxrMedia object that has already been loaded in
- `x` is the first component of the translation vector

Example:

```
mxrMediaHandle H;
mxrMedia       mediaObj;

mxrMediaRead(&H,"snoman.wrl",NULL);
mxrMediaSet(&mediaObj,H);
mxrMediaSetLoop(&mediaObj,MXR_LOOP_FOREVER,45,90);
mxrMediaSetSpeed(&mediaObj, 5.25);
mxrMediaSetFrame(&mediaObj, 10);

mxrMediaSetTranslationX(&mediaObj,60.0);
mxrMediaRender(&mediaObj,&T);
```

Notes:

This prior rotation is NOT part of the "media info" file. Different instances of the same media file can have different absolute translational value.

See also:

mxrMediaSetTranslationXYZ, mxrMediaSetTranslationY, mxrMediaTranslate

mxrMediaSetTranslationY *Release Version:* MXRToolkit V1.0

Short Description:

This sets the absolute translational value for an instance of a media file in the Y axis.

Function declaration:

```
void mxrMediaSetTranslationXYZ(mxrMedia *media, mxrFloat x, mxrFloat y, mxrFloat z);
```

Parameter Interpretation

- `handle` is a handle to an mxrMedia object that has already been loaded in
- `y` is the second component of the translation vector

Example:

```
mxrMediaHandle H;
mxrMedia       mediaObj;

mxrMediaRead(&H,"snoman.wrl",NULL);
mxrMediaSet(&mediaObj,H);
mxrMediaSetLoop(&mediaObj,MXR_LOOP_FOREVER,45,90);
mxrMediaSetSpeed(&mediaObj, 5.25);
mxrMediaSetFrame(&mediaObj, 10);

mxrMediaSetTranslationY(&mediaObj,50.0);
mxrMediaRender(&mediaObj,&T);
```

Notes:

This prior rotation is NOT part of the "media info" file. Different instances of the same media file can have different absolute translational value.

See also:

mxrMediaSetTranslationX, mxrMediaSetTranslationXYZ, mxrMediaTranslate

mxrMediaSetRotationY *Release Version:* MXRToolkit V1.0

Short Description:

This sets the absolute rotational value for an instance of a media file in the Y axis.

Function declaration:

```
void mxrMediaSetRotationY(mxrMedia *media, mxrFloat ry);
```

Parameter Interpretation

- `handle` is a handle to an mxrMedia object that has already been loaded in
- `ry` is the second component of the rotation vector

Example:

```
mxrMediaHandle H;
mxrMedia       mediaObj;

mxrMediaRead(&H,"snoman.wrl",NULL);
mxrMediaSet(&mediaObj,H);
mxrMediaSetLoop(&mediaObj,MXR_LOOP_FOREVER,45,90);
mxrMediaSetSpeed(&mediaObj, 5.25);
mxrMediaSetFrame(&mediaObj, 10);

mxrMediaSetRotateY(&mediaObj,1.57);
mxrMediaRender(&mediaObj,&T);
```

Notes:

This prior rotation is NOT part of the "media info" file. Different instances of the same media file can have different rotational value.

See also:

mxrMediaSetRotationZ, mxrMediaRotate

mxrMediaSetRotationZ *Release Version:* MXRToolkit V1.0

Short Description:

This sets the absolute rotational value for an instance of a media file in the Z axis.

Function declaration:

```
void mxrMediaSetRotationY(mxrMedia *media, mxrFloat ry);
```

Parameter Interpretation

- `handle` is a handle to an mxrMedia object that has already been loaded in
- `rz` is the third component of the rotation vector

Example:

```
mxrMediaHandle H;
mxrMedia       mediaObj;

mxrMediaRead(&H,"snoman.wrl",NULL);
mxrMediaSet(&mediaObj,H);
mxrMediaSetLoop(&mediaObj,MXR_LOOP_FOREVER,45,90);
mxrMediaSetSpeed(&mediaObj, 5.25);
mxrMediaSetFrame(&mediaObj, 10);

mxrMediaSetRotateZ(&mediaObj,3.142);
mxrMediaRender(&mediaObj,&T);
```

Notes:

This prior rotation is NOT part of the "media info" file. Different instances of the same media file can have different rotational value.

See also:

mxrMediaSetRotationY, mxrMediaRotate

mxrMediaSetScaling *Release Version:* MXRToolkit V1.0

Short Description:

This sets the absolute scaling value for an instance of a media file.

Function declaration:

```
void mxrMediaSetScaling(mxrMedia *media, mxrFloat x, mxrFloat y, mxrFloat z);
```

Parameter Interpretation

- `handle` is a handle to an mxrMedia object that has already been loaded in
- `x` is the scaling factor in the x direction
- `y` is the scaling factor in the y direction
- `z` is the scaling factor in the z direction

Example:

```
mxrMediaHandle H;
mxrMedia       mediaObj;

mxrMediaRead(&H,"snoman.wrl",NULL);
mxrMediaSet(&mediaObj,H);
mxrMediaSetLoop(&mediaObj,MXR_LOOP_FOREVER,45,90);
mxrMediaSetSpeed(&mediaObj, 5.25);
mxrMediaSetFrame(&mediaObj, 10);

mxrMediaSetScaling(&mediaObj,2.5,2.5,2.5);
mxrMediaRender(&mediaObj,&T);
```

Notes:

This prior rotation is NOT part of the "media info" file. Different instances of the same media file can have different scaling values.

See also:

mxrMediaScale

mxrMediaGetRotation *Release Version:* MXRToolkit V1.0

Short Description:

Returns current prior rotation settings of a media object

Function declaration:

```
void mxrMediaGetRotation( mxrFloat *x, mxrFloat *y, mxrFloat *z,mxrMedia *media);
```

Parameter Interpretation

- `x` is a pointer to a double, where the first component of the rotation vector will be stored
- `y` is a pointer to a double, where the second component of the rotation vector will be stored
- `z` is a pointer to a double, where the third component of the rotation vector will be stored
- `handle` is a handle to an mxrMedia object that has already been loaded in

Example:

```
mxrMediaHandle H;
mxrMedia       mediaObj;

mxrMediaRead(&H,"snoman.wrl",NULL);
mxrMediaSetPreRotation(&H,1.57,0,0);
mxrMediaSet(&mediaObj,H);
mxrMediaSetLoop(&mediaObj,MXR_LOOP_FOREVER,45,90);
mxrMediaSetSpeed(&mediaObj, 5.25);
mxrMediaSetFrame(&mediaObj, 10);

mxrMediaSetRotationY(&mediaObj,1.57);
mxrMediaRender(&mediaObj,&T);

mxrFloat x, y, z;
mxrMediaGetRotation(&x,&y,&z,&mediaObj);
printf("%f\t%f\t%f\n",x,y,z);

//Output:
//1.57  1.57    0
```

Notes:

This rotational value is the absolute rotational value of an instance of a media object

See also:

mxrMediaGetScaling, mxrMediaGetTranslation, mxrMediaSetPreRotation, mxrMediaSetRotationY, mxrMediaSetRotationZ

mxrMediaGetScaling *Release Version:* MXRToolkit V1.0

Short Description:

Returns current prior scaling settings of a media object

Function declaration:

```
void mxrMediaGetScaling( mxrFloat *x,mxrFloat *y,mxrFloat *z,mxrMedia *media);
```

Parameter Interpretation

- `x` is a pointer to a double, where the first component of the scaling vector will be stored
- `y` is a pointer to a double, where the second component of the scaling vector will be stored
- `z` is a pointer to a double, where the third component of the scaling vector will be stored
- `handle` is a handle to an mxrMedia object that has already been loaded in

Example:

```
mxrMediaHandle H;
mxrMedia       mediaObj;

mxrMediaRead(&H,"snoman.wrl",NULL);
mxrMediaSetPreScaling(&H,2,2,2);
mxrMediaSet(&mediaObj,H);
mxrMediaSetLoop(&mediaObj,MXR_LOOP_FOREVER,45,90);
mxrMediaSetSpeed(&mediaObj, 5.25);
mxrMediaSetFrame(&mediaObj, 10);

mxrMediaScale(&mediaObj,0.5,0.5,0.5);
mxrMediaRender(&mediaObj,&T);

mxrFloat x, y, z;
mxrMediaGetScaling(&x,&y,&z,&mediaObj);
printf("%f\t%f\t%f\n",x,y,z);

//Output:
//1     1       1
```

Notes:

This scaling value is the absolute scaling value of an instance of a media object

See also:

mxrMediaGetRotation, mxrMediaGetTranslation, mxrMediaSetPreScaling, mxrMediaScale

mxrMediaGetTranslation *Release Version:* MXRToolkit V1.0

Short Description:

Returns current prior translational settings of a media object

Function declaration:

```
void mxrMediaGetTranslation( mxrFloat *x, mxrFloat *y, mxrFloat *z,mxrMedia *media);
```

Parameter Interpretation

- `x` is a pointer to a double, where the first component of the translation vector will be stored
- `y` is a pointer to a double, where the second component of the translation vector will be stored
- `z` is a pointer to a double, where the third component of the translation vector will be stored
- `handle` is a handle to an mxrMedia object that has already been loaded in

Example:

```
mxrMediaHandle H;
mxrMedia       mediaObj;

mxrMediaRead(&H,"snoman.wrl",NULL);
mxrMediaSetPreTranslation(&H,20,5,10);
mxrMediaSet(&mediaObj,H);
mxrMediaSetLoop(&mediaObj,MXR_LOOP_FOREVER,45,90);
mxrMediaSetSpeed(&mediaObj, 5.25);
mxrMediaSetFrame(&mediaObj, 10);

mxrMediaTranslate(&mediaObj,1.0,1.0,1.0);
mxrMediaRender(&mediaObj,&T);

mxrFloat x, y, z;
mxrMediaGetTranslation(&x,&y,&z,&mediaObj);
printf("%f\t%f\t%f\n",x,y,z);

//Output:
//21    6       11
```

Notes:

This translational value is the absolute translational value of an instance of a media object

See also:

mxrMediaGetScaling, mxrMediaGetRotation, mxrMediaPreSetTranslation, mxrMediaTranslate, mxrMediaSetTranslationXYZ, mxrMediaSetTranslationX, mxrMediaSetTranslationY

mxrMediaRender *Release Version:* MXRToolkit V1.0

Short Description:

Renders a media file into the scene at a given translation and rotation. This can be a three-dimensional model, sound, an image file or movie.

Function declaration:

```
void mxrMediaRender(mxrMedia *media,mxrTransform *T);
```

Parameter Interpretation

- `media` is a pointer to an mxrMedia, which is a structure that stores essential information of a media object like animation speed, loop status, handle, starting frame, current frame, next frame, last frame, total number of frames, current time, elapsed time and last stored time.
- `T` is a pointer to an mxrTransform structure which contains the Euclidean Rotation and Translation between the camera and the desired object position.

Example:

```
mxrTransform T;                                    // declare transformation matrix
mxrTransformLoadIdentity(&T);                      // fill in values
T.tz = 600;

mxrMediaHandle H;
mxrMedia       media;

mxrMediaRead(&H,"snoman.wrl",NULL);
mxrMediaSet(&media,H);
mxrMediaSetAnim(&media,MXR_LOOP_FOREVER,45,90);
mxrMediaSetSpeed(&media, 5.25);
mxrMediaSetFrame(&media, 10);

mxrMediaRender(&media,&T);
```

Notes:

For sound objects, the rotation part of the transformation matrix is not used, and only the translation values are used.

See also:

mxrMediaRead, mxrMediaSet, mxrMediaSetAnim, mxrMediaSetSpeed

mxrMediaFree *Release Version:* MXRToolkit V1.0

Short Description:

Frees memory associated with a media file and frees space for another media object.

Function declaration:

```
void mxrMediaFree(mxrMediaHandle handle);
```

Parameter Interpretation

- `handle` is a handle to an mxrMedia object that has already been loaded in

Example:

```
mxrMediaHandle H;                                   // declare media handle
if (mxrMediaRead(&H,"snoman.wrl",NULL)){            // read media object
        mxrMediaRender(H,&T);                       // render media object
        mxrMediaFree(H);                            // free media object
}
```

Notes:

When a media handle is loaded in, it occupies a certain amount of memory. In particular, textures associated with it may be loaded into the graphic card memory. When the media object is freed, this memory is returned to common use.

See also:

mxrMediaRender, mxrMediaRead

mxrMediaFreeAll *Release Version:* MXRToolkit V1.0

Short Description:

Frees memory associated with all media file that have been allocated.

Function declaration:

```
void mxrMediaFreeAll(void);
```

Parameter Interpretation

Example:

```
mxrMediaHandle H,I;                                  // declare media handle
mxrMediaRead(&H,"snoman.wrl",NULL);                  // read media object
mxrMediaRead(H, "tris.md2", "rhino.jpg");            // read second object
mxrMediaFreeAll();
```

Notes:

This is a convenient way to free all the objects used in a program.

See also:

mxrMediaFree, mxrMediaRead

mxrMediaInfoRead *Release Version:* MXRToolkit V1.0

Short Description:

Reads information file associated with a media file. This contains the scaling, rotation and translation that allows the model to be realistically displayed in the environment.

Function declaration:

```
void mxrMediaInfoRead(mxrMediaHandle handle, char *filename, mxrMediaType type);
```

Parameter Interpretation

- `handle` is a handle to an mxrMedia object that has already been loaded in
- `filename` is a c-style null terminated string containing the name of the info file.
- `type` contains the type of the media file //TODO – remove this parameter.

Example:

```
mxrMediaHandle H;                                   // declare media handle
mxrMediaRead(&H,"snoman.wrl",NULL);                 // read media object
mxrMediaInfoRead(H, "snoman2.medInfo");             // read in different info file
mxrMediaFreeAll();
```

Notes:

This could be used if the media file is used in two different applications and needs two different scalings/ translations/ rotations. This is convenient way to store this information on disk.

See also:

mxrMediaInfoWrite

mxrMediaInfoWrite *Release Version:* MXRToolkit V1.0

Short Description:

Writes information file containing current scaling, translation and rotation of media file. Can be used to adjust the parameters stored in the media info file.

Function declaration:

```
void mxrMediaInfoWrite(mxrMediaHandle handle);
```

Parameter Interpretation

- `handle` is a handle to an mxrMedia object that has already been loaded in

Example:

```
mxrMediaHandle H;                                   // declare media handle
mxrMediaRead(&H,"snoman.wrl",NULL);                 // read media object
mxrMediaSetScaling(H,2.0,2.0,2.0);                  // set prior scaling
mxrMediaInfoWrite(H);                               // write different info file
mxrMediaFreeAll();
```

Notes:

Can be used to change parameters of media file.

See also:

mxrMediaInfoRead

Three-Dimensional Geometry Library

The three dimensional geometry library deals with the manipulation of objects in three-dimensional space. It is closely related to the two dimensional geometry library which deals with transformations in the image, and the camera library which concerns the projection from three-dimensions to two dimensions. The principle entities in three-dimensions are the point and the plane – there is no general convenient method for describing three-dimensional lines. The three-dimensional plane and point are described by the structures mxrPlane and mxrPoint3D respectively.

The most common three dimensional transformations are rotation, the Euclidean transformation (which represents a rigid rotation and translation of an object in three-dimensional space), and the affine transformation.

A three-dimensional rotation can be represented by the Euclidean matrix structure mxrTransform where the translation vector is set to zero, a quaternion structure, mxrQuat or a rotational vector, mxrRotVec. A three-dimensional Euclidean transformation matrix is described by the mxrTransform structure, and the more general affine transformation can be represented by the same entity.

Current routines in the three-dimensional geometry library consist of:

Rotations

- mxrRotVecToQuat – converts rotation vector into quaternion form
- mxrRotMatToQuat – converts rotation matrix into quaternion form
- mxrRotVecToRotMat – converts rotation vector in to rotation matrix form
- mxrQuatToRotMat – converts quaternion into rotation matrix form
- mxrRotMatToRotVec – converts rotation matrix into rotation vector form
- mxrQuatToRotVec – converts quaternion into rotation vector form
- mxrQuatMult – multiplies two quaternions together
- mxrRotVecMult – composes two rotation vectors
- mxrRotMatMult – composes (multiplies) two rotation matrices

Points and Planes

- mxrPointsToPlane – finds equation of plane through three points
- mxrPlanesToPoint – returns equation of point where three planes intersect

Transformations

- mxrTransformMult – composes two Euclidean transformations
- mxrTransformInvert – inverts Euclidean transformation
- mxrTransformInvertAffine – inverts affine transformation

- mxrTransformLoadIdentity – loads identity matrix into mxrTransform structure
- mxrTransformPoint – transforms position of three-dimensional point in space
- mxrTransformPlane – transforms equation of three-dimensional plane

mxrRotVecToQuat *Release Version:* MXRToolkit V1.0

Short Description:

Converts a rotation vector to a quaternion representing the same rotation.

Function declaration:

```
void    mxrRotVecToQuat(mxrQuat *Q,mxrRotVec *V);
```

Parameter Interpretation

- `Q` is a pointer to a quaternion that will contain the rotation on return
- `V` is a pointer to an mxrRotVec structure containing the rotation vector to be converted

Example:

```
mxrQuat p;
mxrRotVec r;

r.rx = 2.0; r.ry = -1.0; r.rz = 0.3;                // fill rotation vectors

mxrRotVecToQuat(&p,&r);                             // convert to quaternions
```

Notes:

Rotation vectors have the advantage that they are a minimal representation of a three-dimensional rotation and do not need additional constraints. Quaternions are useful for composing rotations and are used for interpolation between rotations.

See also:

mxrRotMatToRotVec, mxrQuatToRotVec, mxrRotVecToRotVec

mxrRotVecToRotMat *Release Version:* MXRToolkit V1.0

Short Description:

Converts a rotation vector into a 3x3 rotation matrix representing the same rotation.

Function declaration:

```
void    mxrRotVecToRotMat(mxrTransform *mat, mxrRotVec *vec);
```

Parameter Interpretation

- `mat` is a pointer to a transformation matrix that will contain the rotation on return
- `vec` is a pointer to an mxrRotVec structure containing the rotation vector to be converted

Example:

```
mxrTransform mat;
mxrRotVec vec;

vec.rx = 2.0; vec.ry = -1.0; vec.rz = 0.3;                  // fill rotation vectors

mxrRotVecToRotMat(&mat,&vec);                               // convert to rotation matrix
```

Notes:

Rotation vectors have the advantage that they are a minimal representation of a three-dimensional rotation and do not need additional constraints. Rotation matrices are useful as they are easily applied to data points by simple matrix multiplication.

See also:

mxrRotMatToRotVec, mxrQuatToRotVec, mxrRotVecToQuat

mxrQuatToRotMat *Release Version:* MXRToolkit V1.0

Short Description:

Converts a quaternion into a 3x3 rotation matrix representing the same rotation.

Function declaration:

```
void    mxrQuatToRotMat(mxrTransform *mat, mxrQuat *quat);
```

Parameter Interpretation

- `mat` is a pointer to a transformation matrix that will contain the rotation on return
- `quat` is a pointer to an mxrQuat structure that contains the rotation vector to be converted

Example:

```
mxrTransform mat;
mxrQuat quat;

quat.r = 1.0; quat.i = 0.0; quat.j = 0.0; quat.k = 0;       // fill quaternion

mxrQuatToRotMat(&mat,&quat);                                 // convert to rotation matrix
```

Notes:

Quaternions are a useful representation for composing rotations or performing spherical linear interpolation. Rotation matrices are useful as they are easily applied to data points by simple matrix multiplication.

See also:

mxrRotMatToRotVec, mxrQuatToRotVec, mxrRotVecToRotMat

mxrRotMatToRotVec *Release Version:* MXRToolkit V1.0

Short Description:

Converts a 3x3 rotation matrix into a 3x1 vector representing the same rotation.

Function declaration:

```
void    mxrRotMatToRotVec(mxrRotVec *v,mxrTransform *mat);
```

Parameter Interpretation

- `v` is a pointer to an mxrRotVec structure that will contain the rotation vector upon return
- `mat` is a pointer to a 3 x 3 rotation matrix stored in an mxrTransform structure which will contain the rotation matrix upon return.

Example:

```
mxrTransform mat;
mxrRotVec v;

mxrTransformLoadIdentity(&mat);                    // load identity into matrix

mxrRotMatToRotVec (&v,&mat);                       // convert to rotation vector
```

Notes:

Rotation vectors are a useful representation as they represent a minimal parameterization of a rotation and are hence suitable for use in optimization techniques. Rotation matrices are useful as they are easily applied to data points by simple matrix multiplication.

See also:

mxrRotMatToQuat, mxrQuatToRotVec, mxrRotVecToRotMat

mxrQuatToRotVec *Release Version:* MXRToolkit V1.0

Short Description:

Converts a quaternion into a 3x1 vector representing the same rotation.

Function declaration:

```
void    mxrQuatToRotVec(mxrRotVec *v, mxrQuat *q);
```

Parameter Interpretation

- `v` is a pointer to an mxrRotVec structure that will contain the rotation vector upon return
- `q` is a pointer to a mxrQuat structure containing the rotation in quaternion form

Example:

```
mxrRotVec       v;
mxrQuat         q;

q.r = 1.0; q.i = 0.0; q.j = 0.0; q.k = 0.0;                 // fill quaternion structure

mxrQuatToRotVec (&v,&q);                                    // convert to rotation vector
```

Notes:

Rotation vectors are a useful representation as they represent a minimal parameterization of a rotation and are hence suitable for use in optimization techniques. Quaternions are a useful representation for concatenating rotations or performing rotational interpolation.

See also:

mxrRotMatToQuat, mxrQuatToRotMat, mxrRotVecToRotMat

mxrQuatToRotVec *Release Version:* MXRToolkit V1.0

Short Description:

Converts a quaternion into a 3x1 vector representing the same rotation.

Function declaration:

```
void    mxrQuatToRotVec(mxrRotVec *v, mxrQuat *q);
```

Parameter Interpretation

- `v` is a pointer to an mxrRotVec structure that will contain the rotation vector upon return
- `q` is a pointer to a mxrQuat structure containing the rotation in quaternion form

Example:

```
mxrRotVec       v;
mxrQuat         q;

q.r = 1.0; q.i = 0.0; q.j = 0.0; q.k = 0.0;                 // fill quaternion structure

mxrQuatToRotVec (&v,&q);                                    // convert to rotation vector
```

Notes:

Rotation vectors are a useful representation as they represent a minimal parameterization of a rotation and are hence suitable for use in optimization techniques. Rotation matrices are useful as they are easily applied to data points by simple matrix multiplication.

See also:

mxrRotMatToQuat, mxrQuatToRotVec, mxrRotVecToRotMat

mxrRotMatToQuat *Release Version:* MXRToolkit V1.0

Short Description:

Converts a rotation matrix into a quaternion representing the same rotation

Function declaration:

```
void    mxrRotMatToQuat (mxrQuat *q, mxrTransform *mat);
```

Parameter Interpretation

- `q` is a pointer to an mxrQuat structure that will contain the quaternion upon return
- `mat` is a pointer to a mxrTransform structure containing the original rotation in matrix form

Example:

```
mxrTransform    M;
mxrQuat         q;

mxrTransformLoadIdentity(&M);                          // load identity into matrix

mxrRotMatToQuat (&q,&mat);                             // convert to rotation matrix
```

Notes:

Quaternions are a useful representation for composing several rotations or performing spherical linear interpolation. Rotation matrices are useful as they are easily applied to data points by simple matrix multiplication.

See also:

mxrQuatToRotMat, mxrQuatToRotVec, mxrRotVecToRotMat

mxrQuatMult *Release Version:* MXRToolkit V1.0

Short Description:

Multiplies together two quaternions using standard quaternion multiplication. When the quaternion is treated as the representation of a rotation, this is equivalent to composing two rotations.

Function declaration:

```
void    mxrQuatMult (mxrQuat *pq,mxrQuat *p, mxrQuat *pq);
```

Parameter Interpretation

- `pq` is a pointer to a quaternion that will contain the product of the two input quaternions
- `p` is a pointer to an mxrQuat structure containing the first of the quaternions to be multiplied
- `q` is a pointer to an mxrQuat structure containing the second of the quaternions to be multiplied

Example:

```
mxrQuat p,q;
mxrRotVec r,s;

r.rx = 2.0; r.ry = -1.0; r.rz = 0.3;                // declare rotation vectors
s.rx = 2.2; s.ry = 0.2; s.rz = 0.1;

mxrRotVecToQuat(&p,&r);                             // convert to quaternions
mxrRotVecToQuat(&q,&s);

mxrQuatMult(&pq,&p,&q);                             // quaternion multiplication
```

Notes:

Quaternion multiplication is a useful way of composing rotations because it does not suffer from the numerical problems of multiplying rotation matrices together. If we take the latter approach repeatedly then the resulting matrices gradually cease to have orthonormal columns and hence cease to be rotation matrices. Quaternion multiplication allows repeated composition of rotations without these problems.

See also:

mxrRotVecToQuat, mxrQuatToRotMat, mxrRotVecMult

mxrRotVecMult *Release Version:* MXRToolkit V1.0

Short Description:

Composes two rotation vectors to form a third composite vector

Function declaration:

```
void    mxrRotVecMult (mxrRotVec *pq,mxrRotVec *p, mxrRotVec *pq);
```

Parameter Interpretation

- `pq` is a pointer to a rotation vector that will contain the product of the two input quaternions
- `p` is a pointer to an mxrRotVec structure containing the first of the rotation vectors to be composed
- `q` is a pointer to an mxrRotVec structure containing the second of the quaternions to be composed

Example:

```
mxrRotVec p,q,pq;

p.rx = 2.0; p.ry = -1.0; p.rz = 0.3;                // declare rotation vectors
q.rx = 2.2; q.ry = 0.2; q.rz = 0.1;

mxrRotVecMult(&pq,&p,&q);                           // compose rotation vectors
```

Notes:

The composition is internally carried out by converting to quaternion form, composing and returning to vector form. Hence, wrapping around the +/- 2π boundary is automatically taken care of. The composition can be carried out in place.

See also:

mxrRotVecToQuat, mxrQuatMult

mxrRotMatMult *Release Version:* MXRToolkit V1.0

Short Description:

Composes two rotation matrices to form a third rotation matrix, such that the orthogonality constraints of the rotation matrix are never compromised.

Function declaration:

```
void    mxrQuatMult (mxrTransform *pq,mxrTransform *p, mxrTransform *pq);
```

Parameter Interpretation

- `pq` is a pointer to a rotation vector that will contain the product of the two input rotation matrices
- `p` is a pointer to an mxrRotVec structure containing the first of the rotation matrices to be composed
- `q` is a pointer to an mxrRotVec structure containing the second of the rotation matrices to be composed

Example:

```
mxrTransform   p,q,pq;
mxrRotVec      a,b;

a.rx = 2.0; a.ry = -1.0; a.rz = 0.3;                // declare rotation vectors
b.rx = 2.2; b.ry = 0.2; b.rz = 0.1;

mxrRotVecToRotMat(&p,&a);                           // convert to rotation matrices
mxrRotVecToRotMat(&q,&b);

mxrRotMatMult(&p,&q);                               // compose rotation matrices
```

Notes:

Although composition of rotation matrices could be carried out by multiplying transformations using mxrTransformMult, repeated application of this technique results in a gradual loss of the orthogonal constraints in the rotation matrix due to rouding errors. Here, the composition is internally carried out by converting to quaternion form, composing and returning to matrix form. The composition can be carried out in place.

See also:

mxrRotVecMult, mxrQuatMult

mxrPointsToPlane *Release Version:* MXRToolkit V1.0

Short Description:

Finds the equation of the plane passing through three three-dimensional points.

Function declaration:

```
void    mxrPointsToPlane (mxrPlane *Q, mxrPoint *p1, mxrPoint *p2, mxrPoint *p3);
```

Parameter Interpretation

- `Q` is a pointer to an mxrPlane structure that will hold the equation of the plane upon exit from the routine
- `P1` is a pointer to an mxrPoint3D structure containing the first of the three-dimensional points defining the plane.
- `P2` is a pointer to an mxrPoint3D structure containing the second of the three-dimensional points defining the plane.
- `P3` is a pointer to an mxrPoint3D structure containing the third of the three dimensional points defining the plane.

Example:

```
mxrPoint3D     p1,p2,p3                              // define points and plane structures
mxrPlane       Q;

p1.X = 1; p1.Y = 0; p1.Z =0; p1.W = 1;
p2.X = 0; p2.Y = 1; p2.Z =0; p2.W = 1;               // fill in values for three points
p3.X = 0; p3.Y = 0; p3.Z =1; p3.W = 1;

mxrPointsToPlane(&Q, &P1, &P2, &P3);                 // calculate equation of the plane
```

Notes:

Note that the points are treated as homogeneous entities, and as such points at infinity can be included. The user must ensure that the "W" component of the three-dimensional point structure is set to one if the points are to be treated as standard three-dimensional entities.

See also:

mxrPlanesToPoint

mxrPlanesToPoint *Release Version:* MXRToolkit V1.0

Short Description:

Finds the equation of the point defined by the mutual intersection of three planes.

Function declaration:

```
void    mxrPlanesToPoint (mxrPoint *p, mxrPlane *Q1, mxrPlane *Q2, mxrPlane *Q3);
```

Parameter Interpretation

- `p` is a pointer to an mxrPoint3D structure that will hold the equation of the point upon exit from the routine
- `Q1` is a pointer to an mxrPlane structure containing the first of the three-dimensional planes defining the point.
- `Q2` is a pointer to an mxrPlane structure containing the second of the three-dimensional planes defining the point.
- `Q3` is a pointer to an mxrPlane structure containing the third of the three dimensional planes defining the point.

Example:

```
mxrPoint3D     p1,p2,p3                          // define planes and point structures
mxrPlane       Q;

Q1.A = 1; Q1.B = 0; Q1.C =0; Q1.D = -1;
Q2.A = 0; Q2.B = 1; Q2.C =0; Q2.D = -1;          // fill in values for three planes
Q3.A = 0; Q3.B = 0; Q3.C =1; Q3.D = -3;

mxrPlanesToPoint(&p, &Q1, &Q2, &Q3);             // calculate position of point
```

Notes:

Note that the resulting point must be treated as a homogeneous entity – the routine does not automatically normalize the point structure. In order to interpret the first three components of the point structure in a conventional Euclidean sense, one must divide each by the fourth component. If the fourth component is zero this implies that the planes are parallel and only intersect at infinity.

See also:

mxrPointsToPlane

mxrTransformMult *Release Version:* MXRToolkit V1.0

Short Description:

Composes together two transformation matrices to form a new composite matrix

Function declaration:

```
void    mxrTransformMult (mxrTransform *pq, mxrTransform *p, mxrTransform *q);
```

Parameter Interpretation

- `pq` is a pointer to an mxrTransform structure that will hold the equation of composite transformation upon exit
- `p` is a pointer to an mxrTransform structure containing the first of the three-dimensional transformations to be composed
- `q` is a pointer to an mxrTransform structure containing the second of the three-dimensional transformations to be composed.

Example:

```
mxrTransform   p,q,pq;                      // declare transformations

mxrTransformLoadIdentity(&p);               // load identity matrices
mxrTransformLoadIdentity(&q);

mxrTransformMult (&pq, &p, &q);             // compose (pq is now identity)
```

Notes:

If both transformations consist of pure Euclidean rotations, it is more appropriate to call mxrRotMatMult, which will ensure that the resulting matrix is still a rotation matrix.

See also:

mxrRotMatMult, mxrTransformLoadIdentity.

mxrTransformLoadIdentity *Release Version:* MXRToolkit V1.0

Short Description:

Loads identity matrix into the rotation components of an mxrTransform structure and zeros into the translation component.

Function declaration:

```
void    mxrTransformLoadIdentity (mxrTransform *T);
```

Parameter Interpretation

- `T` is a pointer to an mxrTransform structure that will represent zero rotation and zero translation on return from the routine.

Example:

```
mxrTransform   T;                          // declare transformation
mxrTransformLoadIdentity(&T);              // load identity matrix
```

Notes:

When the identity transform is applied to any three-dimensional point, the point will be unchanged by the transformation.

See also:

mxrTransformMult, mxrTransformInvert.

mxrTransformInvert *Release Version:* MXRToolkit V1.0

Short Description:

Finds inverse of a given *Euclidean* transformation – i.e. one which will replace transformed points by original ones.

Function declaration:

```
void    mxrTransformInvert (mxrTransform *Tinv, mxrTransform T);
```

Parameter Interpretation

- Tinv is a pointer to an mxrTransform structure which will contain the inverse of the given transformation
- `T` is a pointer to an mxrTransform structure that contains the original transformation matrix

Example:

```
mxrTransform   T,Tinv;                      // declare transformation

mxrRotVec r;
r.rx = 2.0; r.ry= -0.1; r.rz = 1.0;

mxrRotVecToRotMat(&T,&r);                   // fill in rotation part of transformation
T.tx = 100; t.ty = 100;                     // fill in translation part

mxrTransformInvert (&Tinv, &T);             // invert transform
```

Notes:

This routine assumes that the mxrTransform structure holds a EUCLIDEAN transformation matrix (i.e. a pure rotation and translation). If it does not, then the user should call mxrTransformInvertAffine. Can be peformed in place.

See also:

mxrTransformInvertAffine

mxrTransformInvertAffine *Release Version:* MXRToolkit V1.0

Short Description:

Finds inverse of a given transformation – i.e. one which will replace transformed points by original ones.

Function declaration:

```
void    mxrTransformInvertAffine (mxrTransform *Tinv, mxrTransform T);
```

Parameter Interpretation

- Tinv is a pointer to an mxrTransform structure which will contain the inverse of the given transformation
- `T` is a pointer to an mxrTransform structure that contains the original transformation matrix

Example:

```
mxrTransform   T,Tinv;                      // declare transformation

mxrTransformLoadIdentity(&T);
T.r22 = 200; T.r32 = -20.3; T.tx = 204;     // fill in values for transformation

mxrTransformInvertAffine (&Tinv, &T); // invert transform
```

Notes:

If the transformation in the mxrTransform structure is known a priori to be a Euclidean transform (i.e. a pure rotation and translation), then the routine mxrTransformInvert performs this task more efficiently. Can be performed in place.

See also:

mxrTransformInvert

mxrTransformPoint *Release Version:* MXRToolkit V1.0

Short Description:

Applies three-dimensional transformation to a three-dimensional point.

Function declaration:

```
void    mxrTransformPoint (mxrPoint3D *POut, mxrTransform *T, mxrPoint3D *Pin, int n);
```

Parameter Interpretation

- `Pout` is a pointer to the transformed point(s)
- `T` is a pointer to an mxrTransform structure that contains the three-dimensional transformation matrix
- `Pin` is a pointer to the original point(s)
- `n` is the number of points to be transformed.

Example:

```
mxrPoint3D P[2];                                        // declare points
mxrTransform T;                                         // declare transformation

P[0].X = 10; P[0].y = 0 ; P[0].Z = 0; P[0].W = 1;
P[1].X = 40; P[1].y = 20; P[1].Z = 0; P[1].W = 1;       // fill in point positions


P[0].X = 10; P[0].y = 0; P[0].Z = 0; P[0].W = 1;
T.r22 = 200; T.r32 = -20.3; T.tx = 204;                 // fill in transformation

mxrTransformPoint(P,&T,P,2);                            // apply transform
```

Notes:

The user must ensure that the "W" component of the transformation structure is filled in or the results will be incorrect.

See also:

mxrTransformLoadIdentity, mxrTransformInvert, mxrTransformPlane.

mxrTransformPlane *Release Version:* MXRToolkit V1.0

Short Description:

Applies three-dimensional transformation to a three-dimensional plane.

Function declaration:

```
void    mxrTransformPlane (mxrPlane *POut, mxrTransform *T, mxrPlane *Pin, int n);
```

Parameter Interpretation

- `Pout` is a pointer to the transformed plane (s)
- `T` is a pointer to an mxrTransform structure that contains the three-dimensional transformation matrix
- `Pin` is a pointer to the original plane(s)
- `n` is the number of planes to be transformed.

Example:

```
mxrPlane P[2];                                          // declare planes
mxrTransform T;                                         // declare transformation

P[0].X = 10; P[0].y = 0 ; P[0].Z = 0; P[0].W = 1;
P[1].X = 40; P[1].y = 20; P[1].Z = 0; P[1].W = 1;       // fill in plane values

P[0].X = 10; P[0].y = 0; P[0].Z = 0; P[0].W = 1;
T.r22 = 200; T.r32 = -20.3; T.tx = 204;                 // fill in transformation

mxrTransformPlane (P,&T,P,2);                           // apply transform
```

Notes:

n/a

See also:

mxrTransformLoadIdentity, mxrTransformInvert, mxrTransformPoint.

Image Library

The image library contains a number of routines for loading, saving, re-sizing and re-formatting bitmaps. The library is based around the simple mxrImage structure, which has the definition:

```
struct mxrImage{
        int x;                                          // x size of image
        int y;                                          // y size of image
        unsigned char  *ubData;                         // pointer to byte data
        unsigned char  *wData;                          // pointer to word data
        mxrCamera      cam;                             // camera information
        mxrImFormat    format;                          // image format
};
```

The first two parameters refer to the x any y size of the image. The next parameter is a pointer to the image data in terms of bytes. The parameter wData is a pointer to the image data in terms of four-byte words – this is useful when addressing RGBA or BGRA images. The penultimate parameter contains details about the camera with which the image was taken, if this is known. The fourth parameter gives the storage type of the image. The enumeration mxrImFormat has the definition:

```
typedef enum mxrImFormat{
        MXR_RGB,
        MXR_RGBA,
        MXR_GRAYSCALE,
        MXR_BGR,
        MXR_BGRA,
        MXR_BAYER,
};
```

Data storage depends on the format selected. In the RGB and BGR formats, a single pixel is represented by 24bits and accessed via the "ubData" pointer. In the grayscale and Bayer images, a single pixel is represnted by 8 bits and is also accessed via the "ubData" pointer. For the RGBA and BGRA images, a single pixel is represented by 32 bits and may be accessed using either the "ubData" pointer, or accessed one pixel at a time, using the wData pointer.

Memory management of the mxrImage structure is as follows. Upon initialization, the data pointers are set to NULL. When an image is created, the pointers are set to the start of the imgae memory. When an image is deleted they are returned to the value NULL. If the mxrImage object goes out of scope, the memory area pointed to by the "ubData" pointer is automatically freed to prevent a memory leak. The flipside of this is that if you delete the memory at ubData yourself by calling free or delete, you must set the pointer to null, or the system will try and delete the memory again when the image structure goes out of scope.

When passing image structures to sub-routines, it is extremely advisable to pass only a pointer to the structure, rather than the structure itself. If the structure itself is passed, a complete copy of the whole image will be taken, creating a (usually) unnecessary burden on processor speed and memory management.

Current routines in the image library are:

- mxrImageCreate – allocates memory for storage of image
- mxrImageDelete – releases memory that was used for image storage
- mxrImageRead – reads in image from file on disk. Acceptable formats include .JPG and .BMP
- mxrImageWrite – writes an image to disk
- mxrImageCopy – copies image data from one structure to another
- mxrImageResize – resizes an image
- mxrImageChangeFormat – changes the data storage format of an image
- mxrImageConvert – simultaneously changes the size and data storage format of the image

mxrImageDelete *Release Version:* MXRToolkit V1.0

Short Description:

Frees memory associated with an mxrImage structure and sets the data pointers inside to null.

Function declaration:

```
void    mxrImageDelete(mxrImage *im);
```

Parameter Interpretation

- `im` is a pointer to the mxrImage structure which we wish to free

Example:

```
mxrImage im;                                        // define image structure
mxrImageCreate(&im,640,480, MXR_RGB);               // create memory for image
mxrImageDelete(&im);                                // delete image
```

Notes:

This function is called automatically when the image structure goes out of scope. If the user wishes to test whether the image is currently allocated, then they should examine whether the byte data pointer "ubData" in the image structure is equal to NULL.

See also:

mxrImageCreate

mxrImageCreate *Release Version:* MXRToolkit V1.0

Short Description:

Allocates memory for image (bitmap) and initializes an image structure.

Function declaration:

```
bool    mxrImageCreate (mxrImage *im, int x, int y, mxrImFormat format);
```

Parameter Interpretation

- `im` is a pointer to the mxrImage structure which we wish to initialize
- `x` is the width of the desired image in pixels
- `y` is the height of the desired image in pixels
- `format` is the desired format of the image (see notes)
- returns 1 if the memory was successfully allocated or 0 upon fail.

Example:

```
mxrImage im;                                          // define image structure

mxrImageCreate(&im,640,480, MXR_RGB);                 // create memory for image

mxrImageDelete(&im);                                  // delete image
```

Notes:

Upon successful return, the size of the image and the format are updated in in the mxrImage structure, and the "ubData" field points to the start of the image memory. The enumeration mxrImFormat contains the following values.

```
typedef enum mxrImFormat{
       MXR_RGB,                                       // 24 bit RGB images
       MXR_RGBA,                                      // 32 bit RGBA images
       MXR_GRAYSCALE,                                 // 8 bit grayscale images
       MXR_BGR,                                       // 24 bit BGR images
       MXR_BGRA,                                      // 32 bit BGRA images
       MXR_BAYER,                                     // 8 bit Bayer images
};
```

The data should be accessed via the "ubData" pointer in the image structure, except for 32 bit images, where it may alternatively be accessed four bytes at a time, using the "wData" pointer. Image memory is automatically deleted when the image structure goes out of scope.

See also:

mxrImageDelete

mxrImageRead *Release Version:* MXRToolkit V1.0

Short Description:

Reads image from file into image structure in requested format. Allocates or resizes memory if necessary.

Function declaration:

```
bool mxrImageRead(mxrImage *im, unsigned char *filename, mxrImFormat format);
```

Parameter Interpretation

- `im` is a pointer to the mxrImage structure which we wish to initialize
- `filename` is the name of the image file to be read in
- `format` is the desired format of the image
- returns 1 if the image data was successfully read or 0 upon fail.

Example:

```
mxrImage im;                                          // define image structure
mxrImageRead(&im,"test.jpg",MXR_RGBA);                // load in image
mxrImageDelete(&im);                                  // delete image
```

Notes:

Upon successful return, the size of the image and the format are updated in in the mxrImage structure, and the "ubData" field points to the start of the image memory. Current file formats accepted are jpeg files (.jpg), uncompressed windows bitmaps (.bmp), targa image files (.tga) and portable pixel format files (.ppm) The type of image file is decided by the extension of the filename. Where the native format of the file is different from the requested format, an internal format conversion is peformed.

If the memory was previously allocated (i.e. the ubData pointer was not set to NULL), this image memory will be freed or resized to make way for the new image.

See also:

mxrImageDelete, mxrImageCreate

mxrImageWrite *Release Version:* MXRToolkit V1.0

Short Description:

Writes image from image structure into file in requested format.

Function declaration:

```
bool    mxrImageWrite(unsigned char *filename, mxrImage *im);
```

Parameter Interpretation

- `filename` is the name of the image file to be written toin
- `im` is a pointer to the image structure containing the image data that we wish to write to file
- returns 1 if the image data was successfully written or 0 upon fail.

Example:

```
mxrImage im;                                          // define image structure

mxrImageRead(&im,"test.jpg",MXR_RGBA);                // load in jpg image
mxrImageResize(&im,100,100,MXR_LINEAR,&im);           // resize image
mxrImageWrite(&im,"test.bmp");                        // write image to .bmp file

mxrImageDelete(&im);                                  // delete image
```

Notes:

Upon successful return, the image has been written to disk. Current formats supported are uncompressed color jpeg (.bmp), portable pixel format files (.ppm), or windows bitmap (.bmp). The file format required is determined by the extension of the filename.

If the image is in an inappropriate format an internal conversion will take place before writing.

See also:

mxrImageDelete, mxrImageCreate, mxrImageRead

mxrImageResize *Release Version:* MXRToolkit V1.0

Short Description:

Resizes image to an arbitrary new size with same format, allocating memory appropriately,

Function declaration:

```
bool    mxrImageResize(mxrImage *imOut, int x,int y,mxrInterpMethod inter,mxrImage *imIn);
```

Parameter Interpretation

- `imOut` is a pointer to the image structure which will contain the resized image on return
- `x` is the desired width of the resized image in pixels.
- `y` is the desired height of the resize image in pixels
- `inter` is the method for resizing the image – see Notes below
- `imIn` is a pointer to the image which is going to be resized
- returns 1 if the image data was successfully written or 0 upon fail.

Example:

```
mxrImage im;                                                    // define image structure

mxrImageRead(&im,"test.jpg",MXR_RGBA);                          // load in jpg image
mxrImageResize(&im,100,100,MXR_LINEAR,&im);                     // resize image
mxrImageWrite(&im,"test.bmp");                                  // write image to .bmp file

mxrImageDelete(&im);                                            // delete image
```

Notes:

Upon successful return, the image has been resized. Any previous image in the structure "imOut" will be removed. If imOut is not the appropriate size, it will automatically be resized. There are two possible methods for resizing the image, which correspond to nearest-neighbour sampling and bi-linear interpolation respectively.

```
typedef enum mxrInterpMethod{
       MXR_NEAREST_NEIGHBOUR,                       // nearest neighbour interpolation
       MXR_LINEAR,                                  // bi-linear interpolation
};
```

Note that the image can be resized in place if imIn and imOut point to the same structure. However, this is less efficient as it requires internal memory allocation and a copying operation.

See also:

mxrImageFormatChange, mxrImageConvert

mxrImageConvert *Release Version:* MXRToolkit V1.0

Short Description:

Resizes image and changes format simultaneously.

Function declaration:

```
bool    mxrImageConvert(mxrImage *imOut, int x, int y, mxrImFormat format,
                                          mxrInterpMethod interp, mxrImage *imIn);
```

Parameter Interpretation

- `imOut` is a pointer to the image structure which will contain the resized and reformatted image on return
- `x` is the desired width of the resized image in pixels.
- `y` is the desired height of the resize image in pixels
- `format` is the desired format of the new image in pixels
- `inter` is the method for resizing the image – see Notes below
- `imIn` is a pointer to the image which is going to be resized and reformatted
- returns 1 if the image data was successfully written or 0 upon fail.

Example:

```
mxrImage im;                                                    // define image structure

mxrImageRead(&im,"test.jpg",MXR_RGBA);                          // load in jpg image
mxrImageConvert(&im,100,100,MXR_BGR, MXR_LINEAR,&im);           // resize and reformat image

mxrImageDelete(&im);                                            // delete image
```

Notes:

Any previous image in the structure "imOut" will be overwritten. If imOut is not allocated or is an inappropriate size or format then the memory will be re-allocated appopritately. There are two possible methods for resizing the image, which correspond to nearest-neighbour sampling and bi-linear interpolation respectively. See the main image library for a description of valid image formats.

```
typedef enum mxrInterpMethod{
       MXR_NEAREST_NEIGHBOUR,                       // nearest neighbour interpolation
       MXR_LINEAR,                                  // bi-linear interpolation
};
```

Note that the image can be converted in place if imIn and imOut point to the same structure. However, this is less efficient as it requires internal memory allocation and a copying operation and should hence be used with care.

See also:

mxrImageFormatChange, mxrImageResize

mxrImageFormatChange *Release Version:* MXRToolkit V1.0

Short Description:

Converts image data to a different pixel format.

Function declaration:

```
bool    mxrImageFormatChange(mxrImage *imOut, mxrImFormat format, mxrImage *imIn);
```

Parameter Interpretation

- `imOut` is a pointer to the image structure which will contain the reformatted image on return
- `format` is the desired format of the new image in pixels
- `imIn` is a pointer to the image which is going to be reformatted
- returns 1 if the image data was successfully written or 0 upon fail.

Example:

```
mxrImage im,imGray;                                          // define image structures

mxrImageRead(&im,"test.jpg",MXR_RGBA);                       // load in jpg image
mxrImageFormatChange(&imGray, MXR_GRAYSCALE,&im);            // reformat image

mxrImageDelete(&im);                                         // delete images
mxrImageDelete(&imGray);
```

Notes:

Any previous image in the structure "imOut" will be overwritten. If imOut is not allocated or is an inappropriate size or format then the memory will be re-allocated appopritately. Current valid file formats are described by the enumberation mxrImFormat:

```
typedef enum mxrImFormat{
        MXR_RGB,                                             // 24 bit RGB images
        MXR_RGBA,                                            // 32 bit RGBA images
        MXR_GRAYSCALE,                                       // 8 bit grayscale images
        MXR_BGR,                                             // 24 bit BGR images
        MXR_BGRA,                                            // 32 bit BGRA images
        MXR_BAYER,                                           // 8 bit Bayer images
};
```

Some file conversions are not permitted. In these cases the routine will return 0. The file format conversion can be done in place, although this is less efficient than moving the data into a new file.

See also:

mxrImageConvert, mxrImageResize

mxrImageCopy *Release Version:* MXRToolkit V1.0

Short Description:

Copies image from one image structure to another.

Function declaration:

```
bool    mxrImageCopy(mxrImage *imOut, mxrImage *imIn);
```

Parameter Interpretation

- `imOut` is a pointer to the image structure which will contain the resized image on return
- `imIn` is a pointer to the image which is going to be resized
- returns 1 if the image data was successfully written or 0 upon fail.

Example:

```
mxrImage im,imCopy;                                     // define image structures

mxrImageRead(&im,"test.jpg",MXR_RGBA);                  // load in jpg image
mxrImageCopy(&imCopy, &im);                             // copy image

mxrImageDelete(&im);                                    // delete images
mxrImageDelete(&imCopy);
```

Notes:

Any previous image in the structure "imOut" will be overwritten. If imOut is not allocated or is an inappropriate size or format then the memory will be re-allocated appropriately. The image size and format are copied to the new image structure.

See also:

mxrImageConvert, mxrImageResize

Capture Library

The capture library deals with capturing images from cameras or grabbing images from files. It uses multi-threaded code so that capturing occurs in the background while the rest of your program runs. It can capture from more than one source simultaneously, and deals with several different camera types with a constant interface. The principle structure in the capture library is "mxrCaptureStream", which is defined as follows:

```
struct mxrCaptureStream{
        mxrCamera       cam;                    // camera structure containing parameters
                                                // parameters scaled for this viewing size
        int             x;                      // width of stream
        int             y;                      // height of stream
        float           frameRate;              // nominal frame rate
        mxrImFormat     format;                 // pixel format of stream
        mxrCaptureType  type;                   // type of capture object
        mxrCaptureObj*  stream;                 // handle to internal capture object
};
```

The fields have the following interpretations:

- `cam` holds the camera calibration parameters associated with the camera – each image that the capture stream produces is "stamped" with a copy of this structure
- `x` is the width of the images produced by the stream in pixels
- `y` is the height of the images produced by the stream in pixels
- `frameRate` is the nominal frame rate of the camera – this may not be reached in practice
- `type` is the type of capture object – see below
- `stream` is an internal pointer to the streaming object which need not concern the user

Valid capture stream types are defined by the enumeration mxrCaptureType, which has the following definition:

```
typedef enum mxrCaptureType{
        MXR_WEBCAM,                             // general windows video stream
        MXR_DRAGONFLY,                          // point grey dragonfly camera
        MXR_FIREFLY,                            // point grey firefly camera
        MXR_MOVIE,                              // mpg / avi file
        MXR_IMAGE,                              // static bitmap (e.g jpg file)
        MXR_NULL,                               // null renderer (see text).
};
```

The final three stream types do not correspond to cameras, but may be used in testing and debugging code. The MXR_MOVIE type loads in frames from an mpeg or avi file – this allows the user to repeatedly use the same video sequence to test a piece of code. The MXR_IMAGE type repeatedly loads in an image file (see image library) in order to test the behaviouir of a single frame. The final member of the enumeration, MXR_NULL is termed the NULL capture stream, and is defaulted to when there is a failed attempt to open another capture stream. This simply produces "TV static" noise to indicate that the capture stream has not initialized correctly.

The capture library also allows the user to measure the current capture frequency and to set and retrieve camera parameters when supported by hardware. The camera properties currently supported are described by the enumeration mxrCameraProperty, which has the following definition :

```
typedef enum mxrCameraProperty{
        AUTO_EXPOSURE_STATUS,               // 1 or 0 for on or off
        AUTO_GAIN_STATUS,                   // 1 or 0 for on or off
        AUTO_GAIN_HIGH,                     // highest point of range for auto gain
        AUTO_GAIN_LOW,                      // lowest point of range for auto gain
        AUTO_SHUTTER_STATUS,                // 1 or 0 for on or off
        AUTO_SHUTTER_HIGH,                  // highest point of range for auto shutter
        AUTO_SHUTTER_LOW,                   // lowest point of range for auto shutter
        BRIGHTNESS,
        EXPOSURE,
        GAIN,
        GAMMA,
        HUE,
        SATURATION,
        SHARPNESS,
        SHUTTER,
        WHITE_BALANCE_RED,
        WHITE_BALANCE_BLUE,
};
```

The capture library contains the following functions:

- mxrCaptureInit- initializes a camera and retrieves information about image size etc. Allows the user to request different camera modes
- mxrCaptureImage- retrieves an image from the capture stream
- mxrCaptureImageNo - retrieves a particular frame from a movie file
- mxrCaptureKill – closes down a capture stream
- mxrCapturePrintProperties – prints current camera properties to command line
- mxrCaptureGetProperty – retrieves numerical value of specified camera property
- mxrCaptureSetProperty – sets numerical value of specified camera property
- mxrCaptureGetFreq – returns grabbing frequency averaged over the previous 25 frames

Capture Library Example:

The following code gives a working example of capturing simultaneously from three Point-Gray dragonfly cameras, and displaying the output to the screen.

```
#include "mxrSDK.h"

// GLOBAL DEFINITIONS
#define NEAR_PLANE            100
#define FAR_PLANE             10000
#define WINDOW_X              800
#define WINDOW_Y              600
#define WINDOW_XPOSN          0
#define WINDOW_YPOSN          0
#define FULL_SCREEN           0
#define NO_FRAMES             500


// FUNCTION DEFINITIONS
void mxrMain(void);
void mxrKeyboard(unsigned char,int, int);


// GLOBAL DECLARATIONS
mxrCaptureStream      cap1,cap2,cap3;              //  video capture object
mxrImage              im1,im2,im3;                 //  images to receive data
int                   displayCam=0;                //  which camera output is displayed


//
// PROGRAM START
//

int main(int argc, char **argv){

        // ALWAYS CALL BEFORE INITIALIZATION
        mxrInitialize();

        // INITIALISE CAPTURE STREAMS BY CAMERA SERIAL NUMBER
        mxrCaptureInit(&cap1, NULL, MXR_DRAGONFLY, 3030104,NULL);
        mxrCaptureInit(&cap2, NULL, MXR_DRAGONFLY, 3030105,NULL);
        mxrCaptureInit(&cap3, NULL, MXR_DRAGONFLY, 3030106,NULL);

        // INTIALIZE OPEN GL AND SET UP WINDOW
        mxrGLInitOrtho(WINDOW_X, WINDOW_Y);

        // START MAIN RENDERING ROUTINE
        mxrGLStartOrtho(mxrMain, mxrKeyboard);
        return(0);
}


//
// MAIN RENDERING ROUTINE
//

void mxrMain(void){

        // CLEAR SCREEN
        glClear(GL_COLOR_BUFFER_BIT);

        // RETRIEVE VIDEO IMAGE FROM CAPTURE STREAM
        mxrCaptureImage(&im1, &cap1);
        mxrCaptureImage(&im2, &cap2);
        mxrCaptureImage(&im3, &cap3);
```

```
        // DEPENDING ON CONTENTS OF "DISPLAY CAM" DRAW IMAGE FROM CAMERA X TO SCREEEN
        switch(displayCam){
        case 1:
                glDrawPixels(im1.x,im1.y, GL_RGB, GL_UNSIGNED_BYTE,im1.ubData);
                break;
        case 2:
                glDrawPixels(im2.x,im2.y, GL_RGB, GL_UNSIGNED_BYTE,im2.ubData);
                break;
        case 3:
                glDrawPixels(im3.x,im3.y, GL_RGB, GL_UNSIGNED_BYTE,im3.ubData);
                break;
        default:
                break;
        }



        // DRAW FRAME RATE IN TOP CORNER OF SCREEN
        mxrGLDisplayFrameRate(&cap1);
        mxrGLSwapBuffers();
}


// KEYBOARD EVENT HANDLER
void mxrKeyboard(unsigned char keyStroke, int X, int Y){

        if (keyStroke==0x1b){                           // quit if Escape key pressed
                mxrCaptureKill(&cap1);
                mxrCaptureKill(&cap2);
                mxrCaptureKill(&cap3);
                mxrKill();
        }
        if (keyStroke=='c'){                            // change cameras if 'c' button
                displayCam = (displayCam++)%4;
        }
}
```

mxrCaptureInit *Release Version:* MXRToolkit V1.0

Short Description:

Initializes a streaming object (usually a camera) for image capture – sets parameters for camera. Allows the user to request image format and size.

Function declaration:

```
bool mxrCaptureInit(mxrCaptureStream *cap,char *filename, mxrCaptureType type,
                                                  unsigned long idNo, char *movieName);
```

Parameter Interpretation

- `cap` is a pointer to a capture stream object which is to be initialized. The values in the structure upon entering the machine are considered to be a request for a certain size, format and frame-rate
- `filename` is a c-style null-terminated string, containing the name of the mxrCam file which contains details about the parameters describing the camera. If this field is set to NULL, then default parameters will be assigned
- `type` is the type of capture object which is to be initialized. This includes webcams, video files, and Point Gray Firewire cameras (see notes)
- `idNo` contains information about which camera is to be initialized when more than one camera is attached to the bus – this is typically set to 1 for the first camera, 2 for the second etc. Point Gray cameras can be initialized by their serial number, which can alternatively be placed here.
- `filename` is a c-style null-terminated string containing the name of the movie file or image file to be loaded in if the capture type is MXR_MOVIE or MXR_IMAGE.
- returns 1 if the camera was initialized successfully or 0 if it was not initialized.

Example:

```
mxrCapStream cap1,cap2,cap3;                                  // declare capture objects

                                                              // initialize webcam
mxrCaptureInit(&cap1, "myCam.mxrCam", MXR_WEBCAM, 1 ,NULL);
                                                              // initialize movie object
mxrCaptureInit(&cap2, NULL, MXR_MOVIE, 0 , "photo.avi");

cap3.x = 320; cap3.y = 240;                                   // request image size 320x240
mxrCaptureInit(&cap3, NULL, MXR_IMAGE,0 , "simon.png");       // initialize stream to load
                                                              // constant image repeatedly.

mxrCaptureKill(&cap1);                                        // kill capture streams
mxrCaptureKill(&cap2);
mxrCaptureKill(&cap3);
```

Notes:

A particular capture format can be requested by changing the default values of the mxrCaptureStream structure. When the mxrCaptureInit is called, it will search for the closest video mode to that requested. Parameters that can be changed are the image format, image size and frame rate. Upon return from the routine, these parameters will

have changed to the closest values. The philosophy of the capture library is to only supply formats that are directly supported by the hardware itself, and no internal conversions will be carried out internally. Consult your hardware manual rather than experimenting with thesevalues.

The types of object that can be initialized are described by the enumeration:

```
typedef enum mxrCaptureType{
        MXR_WEBCAM,                                     // general windows video stream
        MXR_DRAGONFLY,                                  // point grey dragonfly camera
        MXR_FIREFLY,                                    // point grey firefly camera
        MXR_MOVIE,                                      // mpg / avi file
        MXR_IMAGE,                                      // static bitmap
        MXR_NULL,                                       // null capture stream
};
```

The MXR_MOVIE type will read all standard Windows movie file formats, including .MPG and .AVI. The MXR_IMAGE type will read in a number of common image formats including .JPG and .BMP files – see mxrImageRead for more information.

When more than one camera is attached to the computer, the user specifies a parameter which selects either the n'th camera on the bus, or directly specifies the serial number of the camera.

If camera initialization fails the NULL capture stream will be defaulted to. This simply produces a "TV static" noisy image to indicate that the capture initialization has failed.

See also:

mxrCaptureKill, mxrCaptureImage

mxrCaptureImage *Release Version:* MXRToolkit V1.0

Short Description:

The function mxrCaptureImage grabs a single image from an initialized video stream. Upon exit, the data pointer in the passed image structure will point to the image data.

Function declaration:

```
bool mxrCaptureImage(mxrImage *im, mxrCaptureStream *cap);
```

Parameter Interpretation

- `im` is a pointer to an uninitialized image structure, which will contain the image on return
- `cap` is a pointer to an initialized capture stream from which we intent to grab
- returns 1 if the image is successfullyl grabbed and 0 if not.

Example:

```
mxrCaptureStream cap;                                       // declare capture object
mxrCaptureInit(&cap, "myCam.mxrCam", MXR_WEBCAM,1,NULL);    // initialize capture object

mxrImage im;                                                // declare image structure
mxrCaptureImage(&im, &cap);                                 // capture one image

…                                                           // process/ use image here…

mxrCaptureDone(&im, &cap);                                  // inform capture stream we
                                                            //   are done with the image
mxrCaptureKill(&cap);                                       // kill the capture stream
```

Notes:

The image structure is only initialized between the mxrCaptureImage and the mxrCaptureDone declarations. The latter informs the capture stream that it can start writing over it with the next image from the stream. The user should be sure not to delete or resize the image memory or the capture stream will have no-where to copy the next image. In addition to the image data itself, the image structure is filled with the details of the size and format of the image, and the parameters of the camera that was used to take the image. For MXR_MOVIE objects, this routine just captures the next image from the video file.

See also:

mxrCaptureInit, mxrCaptureKill, mxrCaptureImageNo

mxrCaptureImageNo *Release Version:* MXRToolkit V1.0

Short Description:

The function mxrCaptureImageNo grabs a single requested image from a movie file that has been opened with an mxrCapture object. Upon exit, the data pointer in the passed image structure will point to the image data.

Function declaration:

```
bool mxrCaptureImageNo(mxrImage *im, mxrCaptureStream *cap, int no);
```

Parameter Interpretation

- `im` is a pointer to an uninitialized image structure, which will contain the image on return
- `cap` is a pointer to an initialized capture stream, which has been set to grab from a movie file
- `no` is an integer which selects the frame number
- returns 1 if the image is successfullyl grabbed and 0 if not.

Example:

```
mxrCaptureStream cap;                                    // declare capture object
mxrCaptureInit(&cap, NULL , MXR_MOVIE,1, "3dLive.mpg");  // initialize capture object

mxrImage im;                                             // declare image structure
mxrCaptureImageNo(&im, &cap,31);                         // capture frame 31

…                                                        // process/ use image here…

mxrCaptureDone(&im, &cap);                               // inform capture stream we
                                                         //   are done with the image
mxrCaptureKill(&cap);                                    // kill the capture stream
```

Notes:

The image structure is only initialized between the mxrCaptureImage and the mxrCaptureDone declarations. The latter informs the capture stream that it can start writing over it with the next image from the stream. The user should be sure not to delete or resize the image memory or the capture stream will have no-where to copy the next image. In addition to the image data itself, the image structure is filled with the details of the size and format of the movie, and (when known) the parameters of the camera that was used to take the image. For MXR_MOVIE objects, frames can be called sequentially by using mxrCaptureIMage.

See also:

mxrCaptureInit, mxrCaptureImage, mxrCaptureKill

mxrCaptureKill *Release Version:* MXRToolkit V1.0

Short Description:

Closes a capture stream that has been initialized with mxrCaptureInit. Allows other programs to access the device or file.

Function declaration:

```
void mxrCaptureKill(mxrCaptureStream *cap);
```

Parameter Interpretation

- `cap` is a capture stream which has previously been initialized successfully with mxrCaptureInit

Example:

```
mxrCaptureStream cap;                                    // declare capture object
mxrCaptureInit(&cap, "myCam.mxrCam", MXR_WEBCAM,1,NULL);  // initialize capture object

mxrImage im;                                             // declare image structure
mxrCaptureImage(&im, &cap);                              // capture one image

…                                                        // process/ use image here…

mxrCaptureDone(&im, &cap);                               // inform capture stream we
                                                         //   are done with the image
mxrCaptureKill(&cap);                                    // close the capture stream
```

Notes:

If the capture stream is not properly closed, it is possible that the program will crash upon exit.

See also:

mxrCaptureInit, mxrCaptureImage, mxrCaptureDone

mxrCapturePrintProperties *Release Version:* MXRToolkit V1.0

Short Description:

Prints current properties of the capture stream to the console. Allows user to get an overview of which parameters are supported by the given camera and what their current settings are.

Function declaration:

```
void mxrCapturePrintProperties(mxrCaptureStream *cap);
```

Parameter Interpretation

- `cap` is a pointer to an mxrCaptureStream object that has previously been initialized with mxrCaptureInit.

Example:

```
mxrCaptureStream cap;                                       // declare capture stream
mxrCaptureInit(&cap, "cam.mxrCam", MXR_FIREFLY, 1 , NULL);  // initialize camera
mxrCapturePrintProperties(&cam);                            // print camera properties
mxrCaptureKill(&cap);                                       // kill capture stream
```

Notes:

If the camera has no properties which can be manipulated by software, or they are not supported by the MXR_SDK API, this routine does nothing. The numerical values of properties can be retreived and set using the functions mxrCaptureGetProperty and mxrCameraSetProperty respectively.

See also:

mxrCaptureGetProperty, mxrCaptureSetProperty

mxrCaptureGetProperty *Release Version:* MXRToolkit V1.0

Short Description:

Retrieves numerical value of specified capture device parameter.

Function declaration:

```
bool mxrCaptureGetProperty(double *X, mxrCameraProperty prop, mxrCaptureStream *cap);
```

Parameter Interpretation

- `X` is a pointer to a double precision floating point number which will contain the numerical property of the value on successful return.
- `prop` passes the name of the property that we are querying.
- `cap` is a pointer to the mxrCaptureStream object (usually a camera) that we are querying
- returns 1 if the value was successfully returned or 0 upon failure

Example:

```
double gain;
mxrCaptureStream cap;                                          // declare capture stream
mxrCaptureInit(&cap, "cam.mxrCam", MXR_FIREFLY, 1 , NULL);    // initialize camera

mxrCaptureGetProperty(&gain,MXR_GAIN, &cap);                   // retrieve gain parameter
mxrCaptureKill(&cap);                                          // kill capture stream
```

Notes:

If the camera property is not supported by the hardware then the routine will fail to return a value. Binary ON/OFF values are returned as 1.0 and 0.0 for ON and OFF respectively. Camera properties are described by the enumeration mxrCameraProperty, which has the definition:

```
typedef enum mxrCameraProperty{
        MXR_AUTO_EXPOSURE_STATUS,           // 1 or 0 for on or off
        MXR_AUTO_GAIN_STATUS,               // 1 or 0 for on or off
        MXR_AUTO_GAIN_HIGH,                 // highest point of range for auto gain
        MXR_AUTO_GAIN_LOW,                  // lowest point of range for auto gain
        MXR_AUTO_SHUTTER_STATUS,            // 1 or 0 for on or off
        MXR_AUTO_SHUTTER_HIGH,              // highest point of range for auto shutter
        MXR_AUTO_SHUTTER_LOW,               // lowest point of range for auto shutter
        MXR_BRIGHTNESS,
        MXR_EXPOSURE,
        MXR_GAIN,
        MXR_GAMMA,
        MXR_HUE,
        MXR_SATURATION,
        MXR_SHARPNESS,
        MXR_SHUTTER,
        MXR_WHITE_BALANCE_RED,
        MXR_WHITE_BALANCE_BLUE,
};
```

See also:

mxrCaptureSetProperty, mxrCapturePrintProperties

mxrCaptureSetProperty *Release Version:* MXRToolkit V1.0

Short Description:

Sets numerical value of capture device property

Function declaration:

```
bool mxrCaptureSetProperty(mxrCameraProperty prop,mxrFloat X, mxrCaptureStream *cap);
```

Parameter Interpretation

- `prop` passes the name of the property that we are setting.
- X is a double precision floating point number which will contains the desired numerical value of the property
- `cap` is a pointer to the mxrCaptureStream object (usually a camera) that we are querying
- returns 1 if the value was successfully returned or 0 upon failure

Example:

```
double gain;
mxrCaptureStream cap;                                          // declare capture stream
mxrCaptureInit(&cap, "cam.mxrCam", MXR_FIREFLY, 1 , NULL);    // initialize camera

mxrCaptureGetProperty(MXR_EXPOSURE, 3.44, &cap);              // set exposure parameter
mxrCaptureKill(&cap);                                         // kill capture stream
```

Notes:

If the camera property is not supported by the hardware then the routine will fail to return a value. Binary ON/OFF values should be requested as 1.0 and 0.0 for ON and OFF respectively. Camera parameters supported are described bythe mxrCameraProperty enumeration which has the definition.

```
typedef enum mxrCameraProperty{
        MXR_AUTO_EXPOSURE_STATUS,           // 1 or 0 for on or off
        MXR_AUTO_GAIN_STATUS,               // 1 or 0 for on or off
        MXR_AUTO_GAIN_HIGH,                 // highest point of range for auto gain
        MXR_AUTO_GAIN_LOW,                  // lowest point of range for auto gain
        MXR_AUTO_SHUTTER_STATUS,            // 1 or 0 for on or off
        MXR_AUTO_SHUTTER_HIGH,              // highest point of range for auto shutter
        MXR_AUTO_SHUTTER_LOW,               // lowest point of range for auto shutter
        MXR_BRIGHTNESS,
        MXR_EXPOSURE,
        MXR_GAIN,
        MXR_GAMMA,
        MXR_HUE,
        MXR_SATURATION,
        MXR_SHARPNESS,
        MXR_SHUTTER,
        MXR_WHITE_BALANCE_RED,
        MXR_WHITE_BALANCE_BLUE,
};
```

See also:

mxrCaptureGetProperty, mxrCapturePrintProperties

mxrCaptureGetFreq *Release Version:* MXRToolkit V1.0

Short Description:

Returns the device capture frequency in Hz averaged over the last 25 frames.

Function declaration:

```
float mxrCaptureGetFreq(mxrCaptureStream *cap);
```

Parameter Interpretation

- `cap` is a pointer to an mxrCaptureStream device that has been successfully initialized using mxrCaptureInit and is grabbing frames
- `returns capture` frequency in Hz averaged over the last 25 frames.

Example:

```
mxrCaptureStream cap;                                          // declare capture object
mxrCaptureInit(&cap, "myCam.mxrCam", MXR_WEBCAM,1,NULL);       // initialize capture object

mxrImage im;                                                   // declare image structure

for (int c1 = 0; c1<50 ;c1++){
        mxrCaptureImage(&im, &cap);                            // capture one image
        mxrCaptureDone(&im, &cap);                             // inform capture stream we
}                                                              //   are done with the image

printf("Frequency = %f\n",mxrCaptureGetFreq(&cap));            // print frequency to screen
mxrCaptureKill(&cap);                                          // close the capture stream
```

Notes:

Since the timer averages the frequency over the previous 25 frames, the value returned during the first 25 frames will be incorrect.

See also:

mxrCaptureInit, mxrCaptureImage

Display Library

The display library is responsible for setting up the viewing frustum and provides a number of utility routines for drawing the video, and a number of test objects into the scene. Since graphics operations are inherently specific to a particular graphics language, most routines assume that the user will be programming in OpenGL. Each of these routines is prefaced "mxrGL…" so that the Direct X user will instantly know which parts he must implement himself.

However, there are certain calculations which are independent of the graphics standard used:

- the frustum/ projection matrix must be calculated such that it corresponds exactly to the parameters of the real camera
- the incoming video is divided into 20x20 sub-regions and these are drawn at the back of the viewing frustum. The exact positions are warped to compensate for radial distortion in the camera lens. The 3d positions of these quads must be calculated.

The results of both of these calculations are stored in the mxrFrustum structure, which has the following definition.

```
struct mxrFrustum{
        mxrMatrix       projection;                     // projection matrix
        mxrMatrix       invProjection;                  // inverse of projection matrix
        mxrPoint3D      vertexCoords[MXR_TEX_RES+1][MXR_TEX_RES+1];
                                                        // texture co-ordinates
        mxrPoint2D      texCoords[MXR_TEX_RES+1][MXR_TEX_RES+1];
                                                        // vertex co-ordinates
        int             x;                              // x size of video stream
        int             y;                              // y size of video stream
}
```

The projection matrix and inverse projection matrix are stored in 4x4 mxrMatrix structures. The height and width of the video stream are stored. The fields vertexCoords and textureCoords contain the 3D vertex positions and the 2D texture co-ordinates of the sub-divided image. The simplest way to draw the video image is using quads. An example of this in OpenGL is provided overleaf. The texture co-ordinates are generated so that they are correct if the size of the texture being displayed is the same as the video stream. Since most current hardware insists on textures that have dimensions of integer powers of two, the texture co-ordinates should be re-scaled to to fit the current texture size. If no texture can be allocated that is larger than the video, it must be displayed using two separate textures.

The routine mxrFrustumInit initializes the display structure based on the current parameters of the capture stream. It also allows the user to choose the near and far planes of the viewing frustum in units of mm – this defines what range of depths graphical object will be drawn over.

A number of utility routines are provided which will help the user initialize OpenGL and perform common graphics tasks. The OpenGL specific routines included in the display library are:

- mxrGLSimpleWindow – initializes a simple openGL window using the glut library – see below for code
- mxrGLStart – starts main OpenGL loop – see below for code
- mxrGLDrawVideo – draws video image suitable for AR into viewing frustum – see below for code
- mxrGLSwapBuffers – swaps buffers and displays drawn image
- mxrGLKeyboardDefault – default keyboard handling routine if user does not wish to specify
- mxrGLReshapeDefault – default reshaping routine if user does not wish to specify
- mxrGLDisplayImage – draws a bimap to the screen at a given position
- mxrGLProjMatrixLoad – loads projection matrix from mxrDisplayStructure
- mxrGLModelViewLoad – loads ModelView matrix from mxrTransform structure
- mxrGLWireCube – draws wire cube test object
- mxrGLSolidCube – draws solid cube test object
- mxrGLTeapot – draws teapot test object
- mxrGLText – displays text on screen
- mxrGLDisplayFrameRate – displays current camera sampling rate on screen
- mxrGLInitOrtho – initializes openGL with orthographic projection
- mxrGLStartOrtho – calls main loop with orthographic projection
- mxrGLReshapeOrtho – default reshape function for orthographic projection.

Internals of Selected OpenGL based Display Routines

Since there are many possible choices for how to initialize a window and graphics programming language, we provide full source code for a number of the routines concerning the initialization of the display. This will allow the user to quickly modify the code to suit his own needs. The code is presented with the full description of the routines.

mxrFrustumInit *Release Version:* MXRToolkit V1.0

Short Description:

This routine fills out the mxrFrustum structure (see below), based on the properties of the incoming video stream. There are two major jobs that are performed. First, the appropriate projection matrix is chosen so that the perspective projection used for drawing the 3D models corresponds to that in the real physical camera. Secondly, the routine provides 3D vertex co-ordinates and texture co-ordinates that correctly remove any radial lens distortion from the incoming video.

Function declaration:

```
void mxrFrustumInit(mxrFrustum *frustum, mxrCaptureStream *cap, double near, double far);
```

Parameter Interpretation

- `frustum` is a pointer to the mxrFrustum structure that will be filled by the routine
- `cap` is a point to an initialized capture stream object that we wish to display in the frustum
- `near` is the distance of the near plane in mm – i.e. the closest point to the camera at which objects will be drawn
- `far` is the distance of the far plane in mm – i.e. the furthest distance from the camera at which objects will be drawn.

Example:

```
mxrCaptureStream cap;                                          // declare capture object
mxrCaptureInit(&cap, "myCam.mxrCam", MXR_WEBCAM,1,NULL);       // initialize capture object

mxrFrustum frustum;                                            // declare frustum object
mxrFrustumInit(&frustum, &cap, 100,10000);                     // initialize Frustum object
```

Notes:

The mxrFrustum structure has the following interpretation. The first two fields are 4x4 mxrMatrix structures which contain the projection and inverse projection matrices respectively. The next two fields contain the vertex and texture co-ordinates to display the incoming video. These are in real-world co-ordinates, and the projection matrix must first be loaded to use them properly.

```
struct mxrFrustum{
        mxrMatrix       projection;                     // projection matrix
        mxrMatrix       invProjection;                  // inverse of projection matrix
        mxrPoint3D      vertexCoords[MXR_TEX_RES+1][MXR_TEX_RES+1];
                                                        // texture co-ordinates
        mxrPoint2D      texCoords[MXR_TEX_RES+1][MXR_TEX_RES+1];
                                                        // vertex co-ordinates
        int             x;                              // x size of video stream
        int             y;                              // y size of video stream
}
```

The question might be asked – why do we need to generate special vertex and texture co-ordinates. The reason is that the projection in the real camera is non-linear – it contains radial distortion. In order to match the properties of the computer graphics and the real camera we are faced with a choice – either we must warp the incoming video to compensate for this distortion or warp the graphics that we draw to mach the video. It is both theoretically better and easier to do the former. The vertex co-ordinates in the frustum structure reflect this warping. They place the image so that the warped edges are clipped by the viewing frustum. A side-effect of this is that as the radial distortion effect gets larger the actual area of video viewed diminishes.

See also:

mxrGLDrawVideo

mxrGLSimpleWindow *Release Version:* MXRToolkit V1.0

Short Description:

Opens a simple OpenGL window using the GLUT libraries.

Function declaration:

```
void mxrGLSimpleWindow(bool full, int xWinSize,int yWinSize, int xposn, int yposn);
```

Parameter Interpretation

- `full` is a Boolean variable indicating whether the desired window will be full-screen
- `xWinSize` is an integer containing the desired width of the window in pixels
- `yWinSize` is an integer containing the desired heightof the window in pixels
- `xposn` is an integer containing the top-left position of the window on the screen
- `yposn` is an integer containing the top-right position of the window on the screen

Example:

```
mxrCaptureStream      cap;
mxrFrustum            frustum;

mxrCaptureInit(&cap, NULL, MXR_FIREFLY, 0,NULL);
mxrFrustumInit(&frustum,&cap,NEAR_PLANE,FAR_PLANE);
mxrGLSimpleWindow(1,640,480, 0,0);
```

Notes:

The mxrGLSimpleWindow routines presents a simple way to produce an OpenGL window based on the GLUT library. Since the user may require more flexibility in this area, we provide the actual code which can be modified as required:

```
// initializes simple openGL window suitable for AR
void mxrGLSimpleWindow(bool  full, int xWinSize,int yWinSize, int xposn, int yposn,){
        glutInitDisplayMode(GLUT_DOUBLE|GLUT_RGB);
        glutInitWindowSize(xWinSize,yWinSize);
        glutInitWindowPosition(xposn,yposn);
        glutCreateWindow("MXRToolkit");
        if (full)
                glutFullScreen();
        mxrGLTextureAlloc();   // internal routine allocate texture memory for video
}
```

See also:

mxrGLDrawVideo, mxrGlStart

mxrGLStart *Release Version:* MXRToolkit V1.0

Short Description:

Starts the main OpenGL rendering loop.

Function declaration:

```
void mxrGLStart(void (*mxrMain)(void),void (*mxrKeyboard)(unsigned char, int , int),
                                                    void(*mxrReshape)(int, int));
```

Parameter Interpretation

- `mxrMain` is a pointer to the main rendering routine which will be called every time the screen is updated
- `mxrKeyboard` is a pointer to a routine that will be entered when the keyboard is pressed
- `mxrReshape` is a pointer to a routine that will be called when the window is reshaped

Example:

```
mxrCaptureStream      cap;
mxrFrustum            frustum;

mxrCaptureInit(&cap, NULL, MXR_FIREFLY, 0,NULL);
mxrFrustumInit(&frustum,&cap,NEAR_PLANE,FAR_PLANE);
mxrGLSimpleWindow(1,640,480, 0,0);
mxrGLStart(mxrMain, mxrKeyboardDefault, mxrReshapeDefault);
```

Notes:

This routine can be passed the default arguments "mxrKeyboardDefault" and "mxrReshapeDefault" if the user does not wish to specify routines to handle the keyboard and reshape functions. Once more, we present the complete code for this short routine.

```
// start main open GL loop
void mxrGLStart(void (*mxrMain)(void), void (*mxrKeyboard)(unsigned char, int , int)
                                                    void(*mxrReshape)(int, int)){
        glutReshapeFunc(mxrReshape);
        glutDisplayFunc(mxrMain);
        glutKeyboardFunc(mxrKeyboard);
        glutIdleFunc(mxrMain);
        glutMainLoop();
}
```

See also:

mxrGLDrawVideo, mxrGlSimpleWindow

mxrGLDrawVideo *Release Version:* MXRToolkit V1.0

Short Description:

Uses the information from the mxrFrustum structure to draw the current image to the screen.

Function declaration:

```
void mxrGLDrawVideo(mxrFrustum *frustum,mxrImage *image);
```

Parameter Interpretation

- `frustum` is a pointer to an initialized mxrFrustum structure
- `image` is a pointer to the image from the video stream that we wish to display

Example:

```
…

void mxrMain(void){
        glClear(GL_COLOR_BUFFER_BIT);
        mxrCaptureImage(&image,&cap);
        mxrGLDrawVideo(&frustrum,&cap);
        mxrGLSwapBuffers();
}
```

Notes:

This routine simply uses the pre-calculated vertex and texture co-ordinates to draw the video stream. The complete code is:

```
void mxrGLDrawVideo(mxrFrustum *frustum,mxrImage *image){

    // load in projection matrix
    mxrGLProjMatrixLoad(frustum);

    // select texture and substitute image
    glEnable(GL_TEXTURE_2D);
    glTexEnvf(GL_TEXTURE_ENV,GL_TEXTURE_ENV_MODE, GL_REPLACE);
    glBindTexture(GL_TEXTURE_2D,textureName);
    glTexSubImage2D(GL_TEXTURE_2D,0,0,0,image->x,image->y, GL_RGB ,
                                                    GL_UNSIGNED_BYTE,image->ubData);

    // calculate scaling factors - see main text
    double xScalingFac = (double) frustum->x/(double) TEXTURE_WIDTH;
    double yScalingFac = (double) frustum->y/(double) TEXTURE_HEIGHT;

    // draw quads for video
    glBegin(GL_QUADS);
    for (int cY =0 ;cY<MXR_TEX_RES; cY++){
        for (int cX = 0; cX<MXR_TEX_RES; cX++){
            glTexCoord2f(  frustum->texCoords[cX][cY].x*xScalingFac,
                           frustum->texCoords[cX][cY].y*yScalingFac);
            glVertex3f(    frustum->vertexCoords[cX][cY].X,
                           frustum->vertexCoords[cX][cY].Y,
                           frustum->vertexCoords[cX][cY].Z);
            glTexCoord2f(  frustum->texCoords[cX+1][cY].x*xScalingFac,
                           frustum->texCoords[cX+1][cY].y*yScalingFac);
```

```
            glVertex3f(    frustum->vertexCoords[cX+1][cY].X,
                           frustum->vertexCoords[cX+1][cY].Y,
                           frustum->vertexCoords[cX+1][cY].Z);
            glTexCoord2f(  frustum->texCoords[cX+1][cY+1].x*xScalingFac,
                           frustum->texCoords[cX+1][cY+1].y*yScalingFac);
            glVertex3f(    frustum->vertexCoords[cX+1][cY+1].X,
                           frustum->vertexCoords[cX+1][cY+1].Y,
                           frustum->vertexCoords[cX+1][cY+1].Z);
            glTexCoord2f(  frustum->texCoords[cX][cY+1].x*xScalingFac,
                           frustum->texCoords[cX][cY+1].y*yScalingFac);
            glVertex3f(    frustum->vertexCoords[cX][cY+1].X,
                           frustum->vertexCoords[cX][cY+1].Y,
                           frustum->vertexCoords[cX][cY+1].Z);
        }
    }
    glEnd();
    glDisable(GL_TEXTURE_2D);
}
```

See also:

mxrGLStart, mxrGLSimpleWindow, mxrGLSwapBuffers

mxrGLSwapBuffers *Release Version:* MXRToolkit V1.0

Short Description:

Swaps the front and back OpenGL buffers to the latest update to the screen

Function declaration:

```
void mxrGLSwapBuffers(void);
```

Parameter Interpretation:

n/a

Example:

…

```
void mxrMain(void){
        glClear(GL_COLOR_BUFFER_BIT);
        mxrCaptureImage(&image,&cap);
        mxrGLDrawVideo(&frustrum,&cap);
        mxrGLSwapBuffers();
}
```

Notes:

This routine simply loads the identity matrix into the modelview matrix and calls the glutSwapBuffers command. The complete code is:

```
// swap buffers
 void mxrGLSwapBuffers(void){
        glMatrixMode(GL_MODELVIEW);
        glLoadIdentity();
        glutSwapBuffers();
}
```

See also:

mxrGLStart, mxrGLSimpleWindow, mxrGLSwapBuffers, mxrGLDrawVideo

mxrGLProjectionLoad *Release Version:* MXRToolkit V1.0

Short Description:

Copies information from the mxrFrustum structure to the OpenGl projection matrix.

Function declaration:

```
void mxrGLProjectionLoad(mxrFrustum *frustum);
```

Parameter Interpretation:

- `frustum` is a pointer to the mxrFrustum structure which contains the projection matrix

Example:

```
…

mxrGLProjectionLoad(&frustum);
mxrGLModelViewLoad(&T);
glBegin(GL_QUADS);
        glVertex3f(-100,100,0);
        glVertex3f(100,100,0);
        glVertex3f(100,-100,0);
        glVertex3f(-100,-100,0);
glEnd();

…
```

Notes:

Note that if you intend to use the projection information in the mxrFrustum matrix without calling this routine, that the components are stored in memory differently from the OpenGL matrix, and the matrix must be transposed before a straight transfer can occur

See also:

mxrGLModelViewLoad

mxrGLModelViewLoad *Release Version:* MXRToolkit V1.0

Short Description:

Copies information from the mxrTransform structure into the OpenGL Modelview matrix.

Function declaration:

```
void mxrGLModelViewLoad(mxrTransform *T);
```

Parameter Interpretation:

- `T` is a pointer to the mxrTransform structure which contains the modelview matrix.

Example:

…

```
void mxrMain(void){
        glClear(GL_COLOR_BUFFER_BIT|GL_DEPTH_BUFFER_BIT);    // CLEAR SCREEN
        mxrCaptureImage(&videoImage, &cap);                  // RETRIEVE VIDEO IMAGE
        mxrFrameTrack(&frame,&videoImage);                   // FIND MARKER IN SCENE
        mxrGLDrawVideo(&frustum,&videoImage);                // DRAW VIDEO
        mxrGLProjectionLoad(&frustum);
        mxrGLModelViewLoad(&frame.T);
        glBegin(GL_QUADS);                                   // DRAW SQUARE ON MARKER
                glVertex3f(-20,-20,0); glVertex3f(-20,20,0);
                glVertex3f(20,20,0);   glVertex3f(20,-20,0);
        glEnd();
        mxrGLSwapBuffers();                                  // DISPLAY TO SCREEN
}
```

…

Notes:

The most typical use for this routine is to load the transformation matrix generated by the marker tracking into the model view matrix and then to draw an object on top of the marker.

See also:

mxrGLProjectionLoad

mxrGLDisplayImage *Release Version:* MXRToolkit V1.0

Short Description:

Draws a bitmap to the OpenGL window. This routine is a convenient front end for glDrawPixels that takes image in the form of an mxrImage structure.

Function declaration:

```
void mxrGLDisplayImage(int xWin, int yWin,  mxrImage *im, int xPos, int yPos);
```

Parameter Interpretation:

- `xWin` is an integer providing the horizontal size of the current window
- `yWin` is an integer providing the vertical size of the current window
- `im` is a pointer to an mxrImage structure
- xPos is an integer describing the horizontal position of the origin of the bitmap
- yPos is an integer describing the vertical position of the origin of the bitmap

Example:

```
mxrImage im;
mxrImageRead("temp.png",MXR_RGBA);

…

void mxrMain(void){
        glClear(GL_COLOR_BUFFER_BIT|GL_DEPTH_BUFFER_BIT);    // CLEAR SCREEN
        mxrCaptureImage(&videoImage, &cap);                  // RETRIEVE VIDEO IMAGE
        mxrGLDrawVideo(&frustum,&videoImage);                // DRAW VIDEO
        mxrGLDisplayImage(640,480,im,0,0);                   // SUPERIMPOSE IMAGE
        mxrGLSwapBuffers();                                  // DISPLAY TO SCREEN
}

…
```

Notes:

If the image is passed in MXR_RGBA or MXR_BGRA format then the fourth component is interpreted as transparency during rendering.

See also:

mxrGLStart, mxrGLSwapBuffers

mxrGLWireCube *Release Version:* MXRToolkit V1.0

Short Description:

Draws a wire cube test object given a particular model matrix. A simple way to test the effectiveness of tracking routines

Function declaration:

```
void mxrGLWireCube(mxrTransform* transMat, mxrFloat size);
```

Parameter Interpretation:

- `transMat` is a pointer to a transformation matrix containing the desired modelview matrix – this is usually yielded from a tracking routine
- `size` represents the size of the cube in mm.

Example:

```
void mxrMain(void){
        glClear(GL_COLOR_BUFFER_BIT|GL_DEPTH_BUFFER_BIT);    // CLEAR SCREEN
        mxrCaptureImage(&videoImage, &cap);                  // RETRIEVE VIDEO IMAGE
        mxrFrameTrack(&frame,&videoImage);                   // FIND MARKER IN SCENE
        mxrGLDrawVideo(&frustum,&videoImage);                // DRAW VIDEO
        mxrGLProjectionLoad(&frustum);
        mxrGLWireCube(&frame.T,40);                          // DRAW WIRE CUBE
        mxrGLSwapBuffers();                                  // DISPLAY TO SCREEN
}
```

Notes:

This routine is useful to test whether a given tracking routine is working reliably. This routine is nothing more than a convenient front end for the similarly named routine in the GLUT library.

See also:

mxrGLSolidCube, mxrGLTeapot

mxrGLSolidCube *Release Version:* MXRToolkit V1.0

Short Description:

Draws a solid lit cube test object given a particular model matrix. A simple way to test the effectiveness of tracking routines.

Function declaration:

```
void mxrGLSolidCube(mxrTransform* transMat, mxrFloat size);
```

Parameter Interpretation:

- `transMat` is a pointer to a transformation matrix containing the desired modelview matrix – this is usually yielded from a tracking routine
- `size` represents the size of the cube in mm.

Example:

```
void mxrMain(void){
        glClear(GL_COLOR_BUFFER_BIT|GL_DEPTH_BUFFER_BIT);    // CLEAR SCREEN
        mxrCaptureImage(&videoImage, &cap);                  // RETRIEVE VIDEO IMAGE
        mxrFrameTrack(&frame,&videoImage);                   // FIND MARKER IN SCENE
        mxrGLDrawVideo(&frustum,&videoImage);                // DRAW VIDEO
        mxrGLProjectionLoad(&frustum);
        mxrGLSolidCube(&frame.T,40);                         // DRAW SOLID CUBE
        mxrGLSwapBuffers();                                  // DISPLAY TO SCREEN
}
```

Notes:

This routine is useful to test whether a given tracking routine is working reliably. This routine is nothing more than a convenient front end for the similarly named routine in the GLUT library.

See also:

mxrGLWireCube, mxrGLTeapot

mxrGLTeapot *Release Version:* MXRToolkit V1.0

Short Description:

Draws a Utah teapot test object given a particular model matrix. A simple way to test the effectiveness of tracking routines.

Function declaration:

```
void mxrGLTeapot (mxrTransform* transMat, mxrFloat size);
```

Parameter Interpretation:

- `transMat` is a pointer to a transformation matrix containing the desired modelview matrix – this is usually yielded from a tracking routine
- `size` represents the size of the teapot in mm.

Example:

```
void mxrMain(void){
        glClear(GL_COLOR_BUFFER_BIT|GL_DEPTH_BUFFER_BIT);    // CLEAR SCREEN
        mxrCaptureImage(&videoImage, &cap);                  // RETRIEVE VIDEO IMAGE
        mxrFrameTrack(&frame,&videoImage);                   // FIND MARKER IN SCENE
        mxrGLDrawVideo(&frustum,&videoImage);                // DRAW VIDEO
        mxrGLProjectionLoad(&frustum);
        mxrGLTeapot(&frame.T,40);                            // DRAW TEAPOT
        mxrGLSwapBuffers();                                  // DISPLAY TO SCREEN
}
```

Notes:

This routine is useful to test whether a given tracking routine is working reliably. This routine is nothing more than a convenient front end for the similarly named routine in the GLUT library.

See also:

mxrGLSolidCube, mxrGLWireCube

mxrGLText *Release Version:* MXRToolkit V1.0

Short Description:

Renders text string on top of the video stream.

Function declaration:

```
void mxrGLText(float x, float y, char *msg);
```

Parameter Interpretation:

- `x` is the desired horizontal position on the screen of the text where 0 represents the extreme left of the viewport and 1 the extreme right
- `y` is the desired vertical position on the screen of the text where 0 represents the top of the viewport and 1 represents the bottom.
- `msg` is a pointer to a c-style null-terminated string containing the message to be rendered to the screen.

Example:

```
void mxrMain(void){
       glClear(GL_COLOR_BUFFER_BIT|GL_DEPTH_BUFFER_BIT);    // CLEAR SCREEN
       mxrCaptureImage(&videoImage, &cap);                  // RETRIEVE VIDEO IMAGE
       mxrGLDrawVideo(&frustum,&videoImage);                // DRAW VIDEO
       mxrGLText(0.5,0.5,"Enjoying the show?");             // DRAW TEXT
       mxrGLSwapBuffers();                                  // DISPLAY TO SCREEN
}
```

Notes:

This routine is a convenient way to display textual enhancements to the video such as the frame-rate. Is is actually a simple front end to the text handling routines in the GLUT library, which allow much finer control including the selection of fonts and text size.

See also:

mxrGLSwapBuffers

Tracking Library

The tracking library is concerned with identifying the position of known patterns in the world with respect to the camera. This information is returned in the form of a transformation matrix which relates the position of the camera and the item which is being tracked. The principle structure in the tracking library is termed mxrFrame. It contains information about known patterns that are in a particular fixed reference frame in the world.

For example, if we attach fiducial patterns to the walls of a room to be tracked, all of these patterns are in the same reference frame – since the walls are rigid. If we know the relative positions of the patterns, we only need to find the position of one of them in the world to know the positions of all the rest. There may be more than one frame present at one time – for example, in the same room we may have several more patterns printed out on a sheet of card – since the card is not fixed to the walls of the room and can move independently, it consitites a separate co-ordinate frame. To summarise, the mxrFrame structure describes the smallest independent tracking element. In general, we pass the current video image to the tracking routines, which return the positions of all the frames in the world.

The frame structure contains the following components

```
struct mxrFrame{
       int                 nTrack;             //number of individual components
       mxrTrackType        *trackType;         //type of each component
       mxrTransform        *offset;            //transformation of each to
                                                 centre of frames co-ord system
       bool                *offsetKnown;       //do we know this transformation?
       mxrTrackPtr         *trackPtr;          //pointer to structure describing
                                                 each component
       bool                *active;            //are components currently used?
       bool                planar;             //are components planar?
       bool                foundFlag;          //successfully tracked?
       mxrTransform        T;                  //transformation to frame.
};
```

The enumeration mxrTrackType allows several different types of visual tracking to be selected, although only marker tracking is currently supported.

```
typedef enum mxrTrackType{
   MXR_WIDE_AREA,                              // wide area tracking scheme
   MXR_MARKER,                                 // marker tracking
   MXR_NATURAL_FEATURES,                       // natural feature tracking
};
```

The tracking library currently exposes only a few simple routines:

- mxrFrameTrack – tracks a given number of frames and returns their positions in the world
- mxrFrameRead – loads in data concerning a tracking frame from a stored file

- mxrFrameWrite – saves data concerning a tracking frame that can later be loaded from disk
- mxrFrameChangeOrigin – transforms the position and orientation of the effective centre of the frame
- mxrFrameCreateFromImage – creates a frame structure for tracking two-dimensional patterns of markers from an image file.
- mxrFrameDelete – releases memory associated with frame

mxrFrameTrack *Release Version:* MXRToolkit V1.0

Short Description:

Takes a number of tracking frame structures and returns their estimated position and orientation relative to the camera.

Function declaration:

```
bool   mxrFrameTrack(mxrFrame *frame,mxrImage *im, int nFrames);
```

Parameter Interpretation:

- `frame` is a pointer to the first element of an array of tracking frames.
- `im` is a pointer to an mxrImage structure holding the current video image
- `nFrames` is the number of frames that are currently being tracked
- `returns` 1 if the tracking completed successfully and the position of at least one of the frames was established, otherwise returns 0.

Example:

```
mxrFrame frame;                                  // declare tracking object
mxrImage im;                                     // declare image object

mxrImageRead(&im,"test.jpg",MXR_RGB);            // load in image file
mxrFrameRead(&frame,"test.frame");               // load in frame details

mxrFrameTrack(&frame,&im,1);                     // find position of frame in image
mxrFrameDelete(&frame);                          // release frame information
```

Notes:

On return from the routines, the field frame.foundFlag informs the user if the transformation to the frame was successfully found. If this was indeed the case then the transformation itself can be found in the field frame.T. The routine prefers images to be in MXR_RGB or MXR_BGR format and will perform an internal conversion if this is not the case.

See also:

mxrFrameRead, mxrFrameDelete

mxrFrameChangeOrigin *Release Version:* MXRToolkit V1.0

Short Description:

Changes the origin of the frame – this will have the effect of displacing all objects currently attached to the frame.

Function declaration:

```
void   mxrFrameChangeOrigin (mxrFrame *frame, mxrTransform *transMat);
```

Parameter Interpretation:

- `frame` is a pointer to an mxrFrame structure which contains the frame we wish to modify
- `transform` is a pointer to an mxrTransform structure that describes how the origin of the frame should be changed

Example:

```
mxrFrame frame;                                    // declare tracking object
mxrFrameRead(&frame,"test.frame");                 // load in frame details

mxrTransform T;                                    // declare tansformation
mxrTansformLoadIdentity(&T);
T.x = -20;                                         // fill transform structure

mxrFrameChangeOrigin(&frame,&T);                   // change origin by 20mm to the left

mxrFrameDelete(&frame);                            // delete frame data
```

Notes:

This routine only has the cosmetic effect of shifting the position of the origin of the frame co-ordinate system. It is particularly useful for setting the co-ordinate system to the desired position after using mxrFrameCreateFromImage.

See also:

mxrFrameCreateFromImage , mxrFrameDelete

mxrFrameRead *Release Version:* MXRToolkit V1.0

Short Description:

Loads in tracking details from file into a frame structure.

Function declaration:

```
bool mxrFrameRead(mxrFrame *frame, char *filename);
```

Parameter Interpretation:

- `frame` is a pointer to an mxrFrame structure into which the information will be loaded.
- `filename` is a c-style null-terminated string which contains the name of the frame file to be loaded
- returns 1 if successful or 0 if not successful

Example:

```
mxrFrame frame;                          // declare tracking object
mxrFrameRead(&frame,"test.frame");       // load in frame details

mxrFrameDelete(&frame);                  // delete memory associated with frame
```

Notes:

Loads in information about what type of real-world objects constitute the frame, and how the system should track them.

See also:

mxrFrameCreateFromImage , mxrFrameDelete

mxrFrameWrite *Release Version:* MXRToolkit V1.0

Short Description:

Writes tracking details from memory into a frame structure.

Function declaration:

```
bool mxrFrameWrite(char *filename, mxrFrame *frame);
```

Parameter Interpretation:

- `filename` is a c-style null-terminated string which contains the name of the frame file to be saved
- `frame` is a pointer to an mxrFrame structure from which the information will be saved
- returns 1 if the data was successfully written and zero if it was not.

Example:

```
mxrFrame frame;                          // declare tracking object
mxrFrameRead (&frame, "test.frame");     // load in tracking details

mxrTransform T;                          // declare transformation
mxrTransformLoadIdentity(&T);            // fill in transformation
T.x = -20;

mxrFrameChangeOrigin(&frame,T);          // change origin
mxrFrameWrite("test.frame", &frame,);    // save frame details
mxrFrameDelete(&frame);                  // delete memory associated with frame
```

Notes:

Allows modifications to frame data to be saved back to file.

See also:

mxrFrameCreateFromImage , mxrFrameDelete

mxrFrameCreateFromImage *Release Version:* MXRToolkit V1.0

Short Description:

Takes a bitmap image file containing tracking patterns and fills in a frame structure with which to track it.

Function declaration:

```
bool   mxrFrameCreateFromImage(mxrFrame *frame,char *filename, float dpi);
```

Parameter Interpretation:

- `frame` is a pointer to an mxrFrame structure into which the information will be loaded.
- `filename` is a c-style null-terminated string which contains the name of the image to be loaded
- `dpi` is a floating point value representing the number of dots per inch in image file – this relates the size of the image in pixels to the size of the printed version in the real world.
- returns 1 if the operation was successful or 0 upon failure

Example:

```
mxrFrame frame;                                    // declare tracking object
mxrFrameCreateFromImage(&frame, "test.jpg",300);     // load in tracking details

mxrFrameWrite("test.frame", &frame);         // save tracking details
mxrFrameDelete(&frame);                      // delete memory associated with frame
```

Notes:

This handy routine allows a complete frame to be created by passing in an image file containing a number of fiducial markers. The routine analyses the frame and fills out the structure. The user can then simply print out the image and use it for tracking. This process takes longer than simply loading in frame information directly from disk, so it may be desirable to analyse the JPG and then save it as a frame structure as in the code fragment above.

See also:

mxrFrameCreateFromImage , mxrFrameDelete