Technical University of Munich
in cooperation with *EADS*

Software-Development Project
in the field Computer Science (Informatik)

Combination of Different Tracking Systems

Benjamin Becker
Hegelstr. 26, 81739 Munich
Tel. (0163) 6601095

Written in the:	11th Semester
Finished:	31st December 2004
First Advisor (TU Munich):	Prof. Gudrun Klinker, Ph.D. Technical University of Munich Faculty of Computer Science Boltzmannstr. 3, 85748 Garching Tel. (089) 289-18215
Second Advisor (TU Munich):	Dipl. Inf. Martin Bauer
Practical Advisor (EADS):	Holger Schmidt *EADS* Deutschland GmbH Corporate Research Center Germany 81663 München Tel. (089) 608-21238

Contents

List of Figures

1 Abstract

Current tracking systems have reached a high level of sophistication with many companies and enterprises participating in the research and development. Unfortunately, none of these products offers quality and accuracy at the same time or at least does not have some other major disadvantage.

This work introduces a *Combination of Different Tracking Systems* to create a robust and accurate service that utilizes the advantages of the used tracking subsystems and that compensates their weaknesses. The idea is realised in the *Generic Tracker*. It is based on the open *VRPN* interface which offers a front-end to access and control multiple tracking systems. Featuring both, extrapolation and interpolation as well as a filter to increase accurray and reliability, the *Generic Tracker* uses the collected measurements and creates a robust and generic service.

To evaluate the capabilities, the quality and the usability, the generic tracker is tested on two representative tracking scenarios:

- A work-guiding task in cooperation with *AMIRE*, an *Authoring Framework*. This allows it to work as a part of an *Augmented Reality Solution*, running a workflow supporting service.
- An implementation into *InView*, a *Realtime Raytracer*, to support it's functionality in a *Virtual Reality Scenario*.

2 Fundamentals

2.1 Tracking and its Uses

Tracking is a basic element for augmenting reality with computer graphics. This can be done by tracking a device or an object and compute its pose, the combination of position and orientation. The calculated pose can be used as a coordinate system null point with its axes rotated by the computed directions. The entire virtual scene can be attached to this coordinate center. The other possible way is to use the computed position and orientation transformations to place a single object at this position and orientation so it shown on the appropriate place in three dimensional space.

Recognizing objects in a video image or knowing where they must be offers the chance to assign information directly to them. The worker receives exact pointers on the screen to what parts the message boxes are attached too. Tracking also offers the chance to realise a dynamic and intelligent manual that knows what parts are relevant for the current task and where they can be found. This helps to guide a worker and give him a hand on complex workflows.

This makes tracking very helpful in a wide range of situations, solving problems of different scenarios, ranging from medicine to mechanics.

2.2 Tracking Systems

The idea to track objects with a computer is not new. It has been realised on several different physical bases, each trying to increase compatibility, accuray and handling. The following will give a brief introduction, description and overview of the available systems.

2.2.1 Optical Tracking

Optical Tracking available as two different types.

Marker Based Tracking - The first and already fully implemented approach is to track certain marker fields, usually squares with unique markings to determine the pose. These marker fields are visible sheets of paper or plastic with printed signs on them. For every frame the computer scans the captured video image and uses picture processing to find the characteristic marker fields on the image.

This idea is widely implemented and tested and it is stable but poorly reliable. Occasional losses of the tracked object occur in case the markers are hidden or not directed towards

the camera, since in a perspective sight the marker fields warp and distort. There are algorithms implemented in the picture processing with compensate this effect but naturally the less a marker is visible on the video the more likely recognition fails. Furthermore, the use of tracking marker fields might not be welcome or practical everywhere, since they must be attached to the tracked objects. Considering the delay arising from the video image processing, this approach also is not appropriate for time critical tracking. A marker field can be seen in picture to the right. The standard marker based tracking implementation is the *AR Toolkit*, found in [Kato and Billinghurst, 1999].

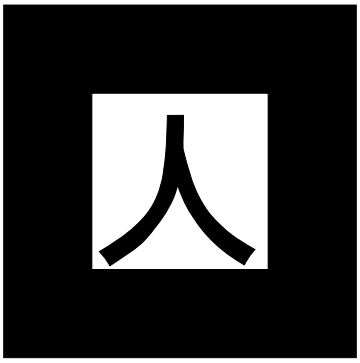


Figure 2.1: Optical Marker

Markerless Tracking - A second way to achive optical tracking is to build a tracker that recognizes objects by their shape. This idea follows the human way to track and determine position and orientation of objects. The computer must be able to find the object in what ever pose it is shown. The objects are not specially marked in any way, so the algorithm tries to learn the significant parts and characteristics of the chosen object. Implementations of markerless tracking are usually based on neuronal networks which are designed for the training on measurement data. They can learn to recognize objects with their current position from video streams.

In spite of recent progress in this area the concept has lots of disadvantages. A neuronal network needs a long training period to produce good results, since it only learns very slowly. Also the neuronal network can only be trained on a single object, which means each tracked object has to have its own. Markerless tracking, therefore, can only track a single object in the scene, unlike the marker based approach that can track multiple targets in its picture processing routine. This leads to a bad portability due to the long initial learning processes. Currently, even after lots of training, the neuronal networks still produce a high rate of failure. Additionally, the use of neuronal networks leads to complicated validation and verification problems, since they are neither deterministic nor predictable. Because of that, it is complicated to improve the quality of the tracking or try a logic way of optimization. Although an evolutionary algorithm offers an approach to alter a neuronal network to recive better tracking results on a special scenario might decrease the performance and quality in the generic case. Due to all these problems, there is no standard implementation available yet.

2.2.2 Infrared Marker Tracking

A similar idea is to track special light reflecting or light emitting markers, using multiple cameras to catch the reflections or impulses and calculating distance and position

from the computed stereo differention. The picture to the right shows such a marker compound, a geomertric model with several marker spots. The tracker's performance increases the more markers are on a target but they need to have a minimum distance so it can distinguish them. More information is available at [Klaus, 2004].
This results in a very fast and exact prediction of the pose. A high degree of stability produces valid data even on partially hidden marker compounds which makes the tracking robust, reliable and superior to the other optical approaches mentioned above. However, mounting cameras might not be welcome or practical everyhwere. The less available the higher the chance of a total occludency is and the less information is gathered to compute the positions. In such a case the tracking fails.
So this very accurate tracking has the handicap of a possible failure.

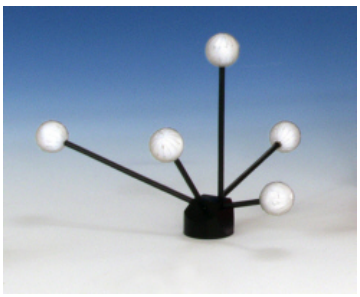


Figure 2.2: Infrared Marker

2.2.3 Gyroscopic Tracking

Another method of tracking is the use of gyroscopic orientation trackers. Internally they have a magnetic gyroscope which rotates due to its inertia against the movement of the device. Its position is calculated on the basis of the magnetic field it creates so it possesses a friction close to zero. The latest generation even compensates the drift created by earth's magnetic field and returns quite valid measurement values. A gyroscopic device is shown in the picture to the right.
The huge advantage is that it does not need additional hardware mounted in the scene, the small cube is totally independant. The disadvantages are the slowly increasing measurement error due to drifting, as well as the lack of knowledge about the position. Basically, it is the ideal fallback for other tracking systems if they fail by moving out of sight or their calculation is not exact enough to compute the pose.
More information is available in [Wormell, 2003].

Figure 2.3: Gyroscopic Tracker

2.2.4 Magnetic Tracking

One of the first ideas to implement tracking was to span magnetic fields through a room. Without the need of direct sight to the tracked object it has a fundamental advantage over all the optical methods and, in contrast to the gyroscope tracking, it measures orientation and position.

1. AC Magnetic Tracking uses a normal magnetic field to compute the pose relative to the magnetic field generator. Electro magnetic tracking is only good for low ranges. There it computes valid and reliable measurements. Unfortunately, it fails in case of metal bodies or other disturbing electronic devices in the environment. Due to these characteristics as well as to the high resolution, the short range and the ability to track even in organic bodies, it is often used for medical purposes.

2. DC Magnetic Tracking is based on pulsed magnetic fields. This increases the effective range and the noise on the measurements is said to be less. So it can be used as a mid-ranged tracking system. Though large senders and semsors may cause additional problems.

3. Radio Frequency Identification (RFID) magnetic tracking works with the tracking of RFIDs. These were introduced to offer detailed information about the object they are attached to but since they send information their position can be computed. Naturally this tracking suffers from the same bad characteristics as the two above mentioned. Though, the huge advantage is, that most companies introduce the small RFIDs to help keeping the inventory listed or have information on components. The availability of many RFIDs allows to increase the formerly poor quality of position determination by tracking multiple targets, filtering the measurement results. This increases the quality and leads to far better result. Since RFID markers are already available in industrial scenarios this allows an easy and fast integration into current workflows. Results of tests concerned with the quality of the possible trackings are not yet available.

2.3 Combination of Tracking Systems

A combination of different tracking systems results in an optimal behavior. This decreases the tracking errors or offers a fallback in case one of the used tracking types fails. Another benefit is the possibility to move through different areas and switch between tracker systems on-the-fly, increasing the field of activity.

An example is the use of RFIDs with a supporting *Gyrotracker*. This computes a more exact position and has a guaranteed orientation measurement. A different but also very good combination is to use infrared tracking for an exact positioning and a gyroscopic tracker to have a minimum tracking available. This is needed in the case the opical system fails due to occludency. The implementation of such a tracker device should not care about the underlying systems but simply combine the available sources to gain the best possible result.

The two trackers used to test the composition of tracking systems and its capabilities are the *AR-Tracking* (described in section 2.2.2 and [Klaus, 2004]) infrared marker tracking and the InertiaCube2 gyroscopic tracking (see section 2.2.3 and [Wormell, 2003]). An attached camera (see [Stricklc, 2001]) returns the video stream which was than augmented with the virtual elements. In this video image the tracked objects are marked and information is attached in the way chapter 2.1 describes. This is achived by tracking the assembled tracking object, consisting of the markers, the gyrotracker and the camera, to compute its current position. This pose is used to transform the coordinate system of the computer graphics scene so the real objects match with the virtually overlayed objects. The handling of such environments is possible with an authoring tool such as AMIRE. A picture of the assembled tracking systems is shown to the right.

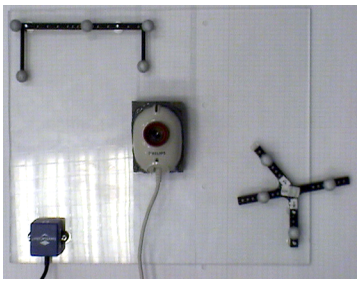


Figure 2.4: Infrared and Gyroscopic Tracker and Camera

2.4 VRPN

The *Virtual Reality Peripheral Network* VRPN, developed by the *University of North Carolina*, offers a generic interface to many trackers. Using it simplifies the interfacing of the trackers and helps focusing the attention on the procession of the data. Standard callback functions handle the trackers and forward information to the applications. Additional information can be found at [Taylor and Yang, 2004a]. A short introduction on the implementation of a VRPN client is available at [Taylor and Yang, 2004b].

The concept of VRPN and its communications is shown in figure 2.5. VRPN consists of a system independent library which can be statically linked into any tracking application. Internally, mainly socket communications are used to establish connection with the different VRPN servers. These handle the communication with the chosen tracker subsystems. The servers are specific for every tracking device, the client offers a generic interface to handle any server compatible to the VRPN standard.

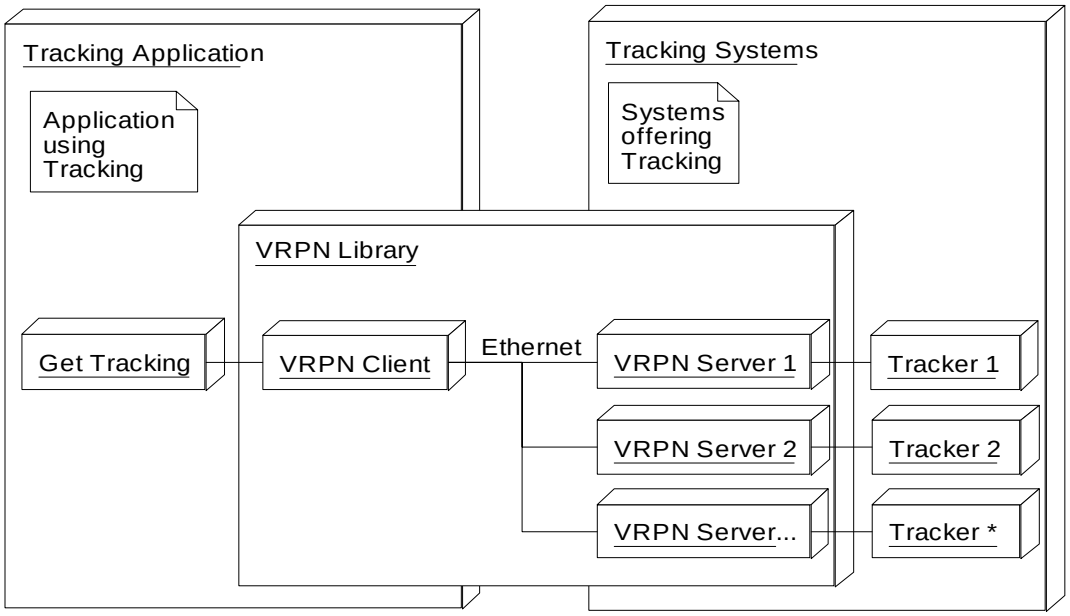


Figure 2.5: VRPN Communication Diagram

3 Generic Tracker

The following chapter introduces the *Generic Tracker* component. This component is able to connect to several differen tracking systems and collect their measurements for further *interpolation, extrapolation* and *validation*.

The tracking systems are controlled and used with the VRPN interface. Its general assembly is defined in [Taylor and Yang, 2004b]. The component has a second interface implemented, the ISD Tracker. This is needed for the *Inertia Cube 2* [Wormell, 2003] and other InterSense products which do not offer a good VRPN implementation.

The component is configured with an extended markup language (XML) file that allows different tracker systems to be chosen and the basic coordinate system to be defined. Depending on the selected tracking subsystems, the component collects the data and interpolates and filters the most accurate value. In case of a total loss of tracking, the component does not receive any further information and will continue to extrapolate the movement from the recently measured values.

3.1 General Specifications

3.1.1 Multi-Threading

The component is written multi-threaded. Because of this, it works non-blocking and calculates the latest position and orientation while the main application fulfills its tasks. This is very important for the drawing routines, which are time critical. In the recent implementation the component's CPU cost is below 1%. For safe multi-threading the program sequences are checked and validated to secure access on shared variables. These are additionally secured with mutexes and semaphores. This avoids deadlocks and race conditions to guarantee a thread-safe execution.

3.1.2 System Independent

Available in three possible configurations the generic tracking component is fully system independent. The possible choices are:

1. Dynamic Linked Libray (DLL): The standard Windows implementation. A dynamically linked library is created and used without header files, allowing it to loaded and unloaded at use.

2. Shared Object (SO): A Linux implementation as a shared object allows the use in Linux applications by using the dynamical linker it in the makefile. This also does the loading at runtime.

3. Socket Communications: The socket based implementation is designed for use on networks. This allows the application and the tracking component to be on physically

different hosts. It is currently designed for linux but can be ported to work on windows aswell. This design increases independency from the application from which it is being used. A simple and small protocol is defined to specify the data packages and the communication commands.

3.1.3 Network Based

Not only the socket based implementation of the *Generic Tracker* is meant to run on any network, but also the VRPN Client and VRPN Server as well as most of the server to tracker system communication are socket based. The communication structure of the VRPN library is outlined in figure 2.5. All this allows a distribution over different hosts on a network, increasing its usability.

3.2 Implemented Features

3.2.1 Validation

Data is collected from all the subsystems that track the device. Since not all of them return valid information, their use might corrupt the computed pose. The validation to determine if the calculated or measured values are correct is achived in the following steps.

1. The tracking subsystems return special values in the case that they have lost the object. These values are ignored. If that happens the system either tries to use the result of the other trackers or uses an extrapolation to keep track.

2. The generic component compares the different measured values to decide if the value is corrupted. In this case, depending on the system, it either sets a correction factor or entirely drops the corrupted data.

The position validation of the *Generic Tracker* is based on the behavior of the subsystem. In case they do not return valid information, the component ignores it. This is implemented for VRPN as well as for ISD Tracker systems. If the *Generic Tracker* looses all position tracking information it uses an extrapolation, as described in chapter 3.2.3.

3.2.2 Interpolation

Interpolation is needed if more than one tracking component returns valid information. Two different algorithms can be used to compute the results:

1. Linear:

   Linear interpolation can be used if the errors of the sensors are rather small and the noise in the measurement is already reduced.

2. Kalman Filter:

   Kalman's filter is used in case there are many tracked devices and the noise is high on the results. The filter reduces the high frequent noises and smooths the result.

The *Generic Tracker* implements a linear interpolation, since most of the tracker subsystems already implement a Kalman filter. Filtering a second time would smooth the results too much and produce artifacts and larger error values.

Interpolation is used for the calculations of the position. There is a spherical interpolation on the orientation quaternions (Quaternions -> [Kolb, 2002]), which removes jerky movements in the change of the pose and returns the most accurate value to a certain time.

However, Kalman filters are not designed to filter objects as as orientations. Orientations are quadruples of values describing the position on a sphere. Since there is no specified way to filter the quaternion's four values, the result will most likely mutate and morph. Thus the orientation must be interpolated linearly.

3.2.3 Extrapolation

Extrapolation is always computed if enough data has been collected without loss. This feature is only implemented on movement, not on orientation, because the gyroscopic tracker does usually not fail. Additionally, extrapolation on orientation is only possible in a linear way, as described above.

The application collects the last twelve measurements of the position to predict the movement in the near future. A simple linear extrapolation of movement would not match normal movement. With optical trackers this happens if someone steps in between the tracking system and the tracked objects. The tracked object will not neccessary move on in a linear but possibly in some more complex way. To avoid simple linear movement approximation for each axis a fifth grade polynomial fitting is computed. Basically that means that the algorithm searches a polynom of grade five that fits the twelve points best, meaning it has a minimum square error. Figure 3.1(a) shows a polynom fitted into points.

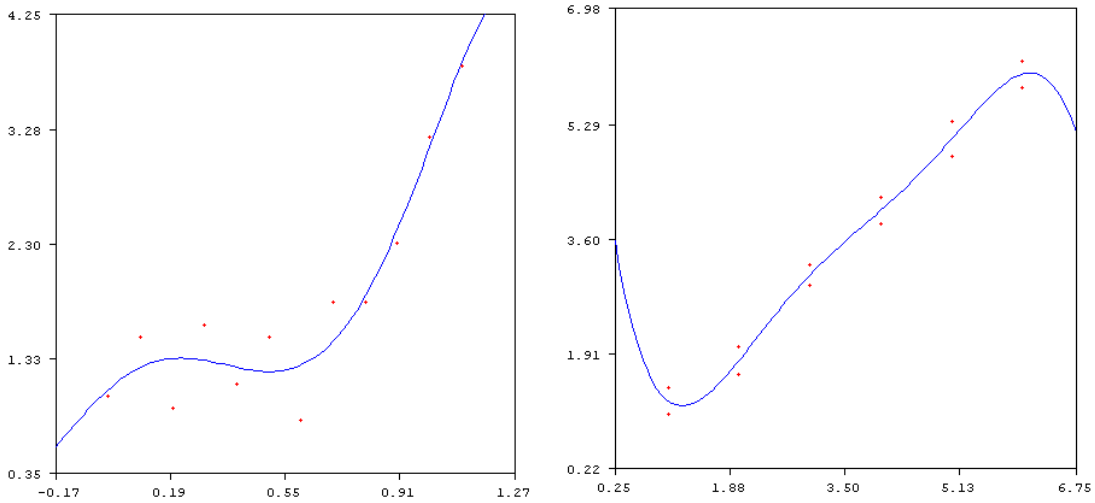
Another advantage of the polynomial approach is that it is possible to compute 1st and 2nd derivatives. These allow an approximation of the tracked object's speed, which allows a far better prediction of the current movement. This particular extrapolation uses the derivatives to start a physical equation of the accelerated motion to fit real movements.

As seen in figure 3.1(b), polynomial fitting, *Polyfit*, automatically takes care of interpolation of the dataset. The data it uses to build the polynom into can have multiple points for a single time value, which will be interpolated in the result. In case there are not enough valid recent measurements the interpolation is skipped, since it would produce faulty results. This happens if there are less than ten good measurements. Unfortunately, polynomial fit produces corrupted boundary values, so the approximation is not computed for the last measurement. Instead the time between the last two collected position values is used.

Polyfit internally uses Gaussian equations to solve the minimum square error problem. Its general concept is described at [Kalies, 2003] and some additional information is found on [Gibson, 2004].

3.3 Composition - The Client's Point Of View

This chapter describes the functions and parameters the user is able to access. The internal functions are described in detail in the *Server's Point Of View* in section 3.4. The user interface of the *Generic Tracker* differs, depending on which of the three implementated versions chosen. Detailed information on implementing the back-ends for the different interface types is given in chapter 3.4.2.



(a) Standard (b) Filter (Interpolation)

Figure 3.1: Tracker - Polyfit Example [Kalies, 2003]

3.3.1 User Interface Functionality

1. The dynamic linked library and windows based version of the generic tracker exports three functions to handle the component.

   - The `InitialiseTracking()` function is used to start the tracking and initialise all the connections and components.
   - `UpdateTracking()` collects the current position and orientation from the component and fills the information into the matrix.
   - Use of the tracked position is implemented with the `double GetTrackingValue (int i)` operation, which exports one of the sixteen matrix values. To get the complete transformation matrix this needs to be called sixteen times.

   The first step in employing this component is to initialise it. Then `UpdateTracking()` and the `double GetTrackingValue(int i)` loop should be called alternatingly. The update function is used to make the tracker compute the recent tracking position and orientation. If this is skipped the values will be out-dated. The matrix describing the orientation and position is polled stepwise, returning a single value per call. The entire matrix needs sixteen calls. To collect the tracking value again later it only needs the update, not the initialisation. Stopping the component and deleting its members is done by unloading the library. Detailed information on loading and unloading as well as on the use can be found in chapter 4.5.1 where this DLL based version of the generic tracker is implemented.

2. The shared-object implementation for Linux uses only two functions to handle the component and uses a slightly different syntax. It is not based on matrices to describe the position and orientation, but uses a position vector and an orientation quaternion.

   - `InitialiseTracking()` initialises the component and its structure, just as in the DLL front-end configuration.

- The function `getTracking(&double,&..)` internally handles two tasks. First, it updates and calculates the current values and, in the second step, it fills the the computed values into the variables addressed by the passed pointers.

The components use starts with initialisation. To obtain the current tracking state it is sufficient to call the get function with the parameters, which will be filled with the computed position and orientation. This can be done anytime after the tracking has been initialised.

This implementation is fundamental for the socket implementation and is implemented into the socket server. Otherwise it is unused at this stage.

3. The stand-alone server implementation is socked based. It works with a server / client structure, which is the most complicated interface. The protocol is kept as small as possible and implements only the very basic features. The use of the standalone server needs an implemented client. The server can be started without parameters.

   The server receives commands from the client and sends back the tracking information. The communication of the client and server is shown in the unified modelling language (UML) diagram 3.3.

   The component's `init_client()` and `get_tracking()` tell the client to communicate with the server. The initialisation starts the client and tests the connection to the server, which should be started beforehand. In case it does not fail, the current tracking can be collected with calling the `get_tracking()` function.

   For more information on this very complicated implementation refer to chapter 3.3.2. A detailed description of the process and its neccessary elements is given there.

3.3.2 Front-End Implementation

The Dynamic Linked Library Front-End

The DLL's front-end is designed to load the library during runtime. Since it does not offer *include files* or *static library headers*, it is neccessary to map them directly. The loading is achieved with the function `LoadLibrary("Tracking.dll")`. This opens the DLL and loads it into the current application. To access its functionality, the library functions need to be mapped to pointers. This is done by calling `GetProcAddress`, which searches for a function with a certain name and casts it on a predefined function type. Defining its type is done with `typedef`, which can describe a function pointer with its *return type* and *parameter list*.

Example code:

```
typedef void (*cfunc_v)();
typedef double (*cfunc_d)(int);

HINSTANCE hLibrary;
cfunc_v UpdateTracking;
cfunc_d GetTrackingValue;
```

```
hLibrary=LoadLibrary("Tracking.dll");

InitTrack=(cfunc_v)GetProcAddress((HMODULE)hLibrary, "Initiali..");
UpdTrack =(cfunc_v)GetProcAddress((HMODULE)hLibrary, "Update....");
GetTrack =(cfunc_d)GetProcAddress((HMODULE)hLibrary, "GetTrack..");
```

As seen in the example code, the operation to map the library functions out of the DLL is the `GetProcAddress`. It maps the function from the library that is matching the name onto the function pointer. After this, the function can be accessed. The function pointer type is defined before with a typedef. The declaration for a function which is returning void and has an empty parameter list is defined with `typedef void (*cfunc_v)()`. Detailed information on DLLs and their functionality as well as on the exporting and importing is provided in [Miller, 1999].

The Shared Object Front-End

The use of shared objects in Linux is based on GNU's *make* command. The makefile used for compilation needs to link against the shared object. Use of an already compiled shared library is realised by telling the C++ compiler to include that library. Additional to the normal compilation parameters the phrase `-L<path> -ltracker` is added. `Path` represents the directory in which to find the library. The library nomenclature requires it to start with the word *lib* and have the type *so*. So the name in our case is `libtracker.so`.

To recompile the shared library the compiler parameter `-shared` is used. It tells the C++ compiler to create a shared object which will contain all the object files.

The Socket Communication Client's Front-End

This interface is used by including the headerfiles and the objects into the current project. The makefile needs to compile the socket communication code as well as the client. The client takes care of the exceptions that can occur on network transmission. This implementation links the client statically into the project. The server runs as a separate program and must be started from the command line before use.

3.4 Composition - The Server's Point Of View

This chapter describes the implementation of the generic tracker component in detail. It consists of the central component, which is identical on all the three different implementations, and the interfaces. These so called back-ends implement the different behavior, required for the work as DLL, SO or Socket Server.

3.4.1 Generic Tracker's Central Component

The main component of the *Generic Tracker* has several separate classes. Their correlation, hierarchy and assembly is shown in detail in the UML (Unified Markup Language) diagram 3.2.

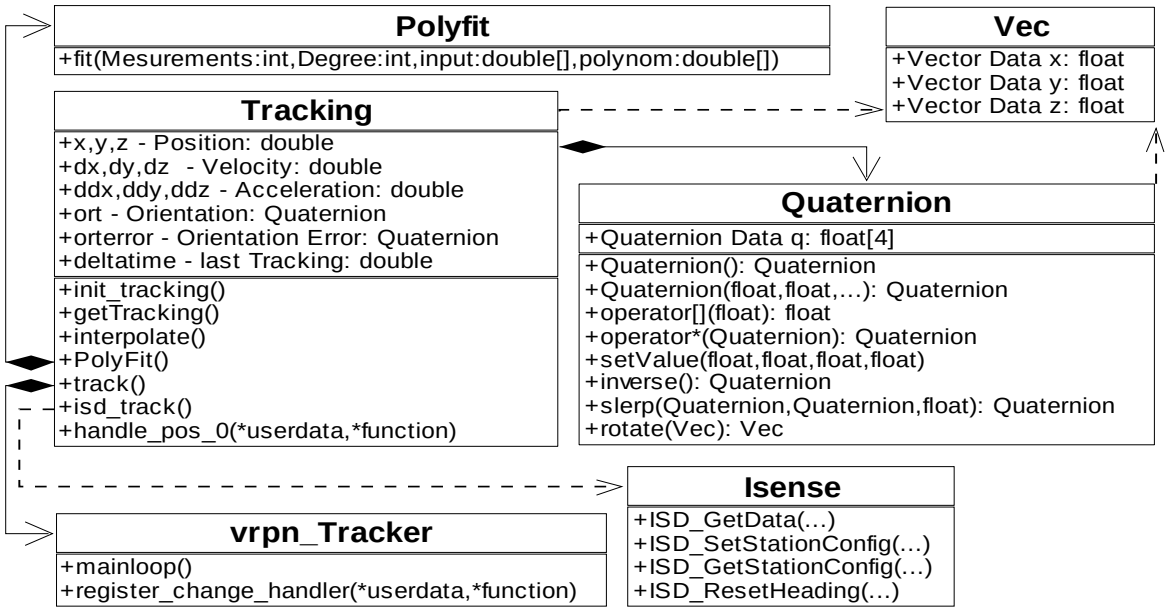


Figure 3.2: Generic Tracker - UML Design

Class - Tracking

The class `Tracking` (3.2) is the main part of the application. It offers routines to initiate the tracking and for returning the current pose. Also the thread', handling the tracking and the data processing, is located here.

Functions

- The initialisation is done by calling `init_tracking()`, setting up data structures and registering the trackers. It creates an *ISD Tracker* and a *VRPN Tracker* device. The initialised *ISD Tracker* has a function implemented to return the measured orientation. The *VRPN Tracker* needs to register a special tracking handler with the (`vrpn_Tracker->register_change_handler()`) (`handle_pos_0()` is such a handler).

- The data processing thread `track()` runs in an endless loop and takes care of the basic tracking, until program termination calls its `cleanup()` function. It checks the different sensors and tracking systems for new data and fills their data structures.

  The function call of `vrpn_Tracker->mainloop()` (3.2) collects all the VRPN based tracker measurements, as well as the `isd_track()` which collects the measurement values of the attached ISD trackers. It wraps the `Isense->ISD_GetData()`. After the data has been grabbed, the interpolation routine is called. A short sleep takes care of avoiding *busy waits* for new tracking data and prevents high cpu usage. For thread-safe execution the system independent functions `lock()` and `unlock()` offer blocking locks, to avoid race conditions and to guarantee valid data.

- The operation `interpolate()` implements the process of handling measurement data.

First, a linear interpolation between the different sensors and their responses is performed. Linear interpolation on the orientation quaternions is implemented by a spherical interpolation. Measurements are always marked with a time stamp by the system independent function `currenttime()`. The computed data is filled into a buffer of 12 data samples required by the polynomial fit.

- The so called `PolyFit()` operation is run if at least the 10 last measurements were in order. It computes a polynomial fit to minimise the square error. The explanation of this routine can be found at 3.2.3.
- For using the component call `getTracking()` to fetch the position and orientation of the tracked object. The setup and configuration of the *Generic Tracker* is realised with a configuration file.

Attributes	Type	Functionality
`x,dx,y,dy,z,dz`	Double	These static variables store the position and the speed in the direction of the three axis
`ort`	Quaternion	The orientation, interpolated of all valid measurements, is stored here.
`orterror`	Quaternion	The orientation difference between the measurement of the *ISD Tracking* and the one of *VRPN Tracking*. The error of the *ISD Tracking* arises from the gyroscope's drift.
`Deltatime`	Double	Deltatime, the elapsed time since the last measurement, is used for extrapolation of the positions.

Class - vrpn_Tracker

The class `vrpn_Tracker` (3.2) embeds the *VRPN Client* code. It offers routines for adding handlers to process the returned measurement values and to call the main *VRPN Tracking loop*.

Functions

- To collect all data and handle the *VRPN Tracker* the `mainloop()` is called. It sends packages to all available and registered *VRPN Servers* and collects their data using the registered change handlers.
- For adding new change handler functions the `vrpn_Tracker` implements the operation `register_change_handler()`. This function takes a pointer to a handler function and a pointer on user data as parameters. The userdata can be used in the handler to influence the result. The handlers are called each time a new package is received.

Class - Isense

The class `Isense` (3.2) embeds the *ISD Tracker* client code. It offers routines to reset the orientation or to alter the configuration data. Additionally, it implements a routine to collect the recent measurement.

Functions

- To return the collected data the function `ISD_GetData()` is called. Its called with a pointer on an empty data structure as parameter. Then it fills it with either an orientation quaternion or Euler angles, depending on the configuration settings made.
- Getting the current configuration of the tracker is done with `ISD_GetStationData()`.
- After altering the configuration, `ISD_SetStationData()` will save the new configuration.
- Resetting the orientations heading is implemented with the `ISD_ResetHeading()` function.

Class - Polyfit

The class `Polyfit` (3.2) embeds the *Polynomial Fitting* code. It offers the `fit()` function, which computes the corresponding minimum square polynom of a certain degree and a given set of points.

Class - Quaternion

The class `Quaternion` (3.2) is used to work with the orientation quaternions. It offers various functions for simple mathematical and numeric tasks.

Functions

- The constructor creates a new quaternion, either with the default zero rotation (0.0, 0.0, 0.0, 1.0) or with the four given values.
- The `[]` operator allows reading access to the values of the quaternion
- The `*` operator implements the multiplication that represents the added rotation of two orientations.
- The `inverse()` function returns the inverse rotation.
- The `slerp()` function is used to do spherical interpolation between two quaternions. A scaling factor defines how much each quaternion influences the new orientation.
- The `rotate()` function takes a vector and rotates it with the current orientation.

Class - Vec

The class `Vec` (3.2) is used to cooperate with the `Quaternion` class. It is useful for rotating and modifying vectors with directions and orientations.

3.4.2 Back-End Structure

The Dynamic Linked Library Back-End

The back-end for the Windows library is exporting the used functions. This can only work as a C function export since C++ has different binary formats, depending on the used compiler. Importing will fail if library and application use different compilers. Functions are exported with `extern "C" __declspec(dllexport)`. These usually wrap the C++ functions of the library's main part. More information on DLLs and their creation is available at [Miller, 1999].

The Shared Object Back-End

Creating a shared object is very simple. It is basically done like compiling a normal application with the difference that shared objects do not contain a main routine. The compiler prefix is `-shared` and the output nomenclature is lib`NAME`.so.

The Client Server Communcation and Back-End

This section gives a detailed view into the client - server communication. The server and client exchange information over the network by using a small protocol. The header `datagram.h` contains a data package(`tpackage`) that can be transmitted over the connection. It is sent as an array of characters, which implicates the need of serialisation of the package. This is achieved by casting it into a character buffer. The received package normally contains the position and the orientation, as seen in figure 3.4. The communication between server and client is displayed in scheme 3.3.

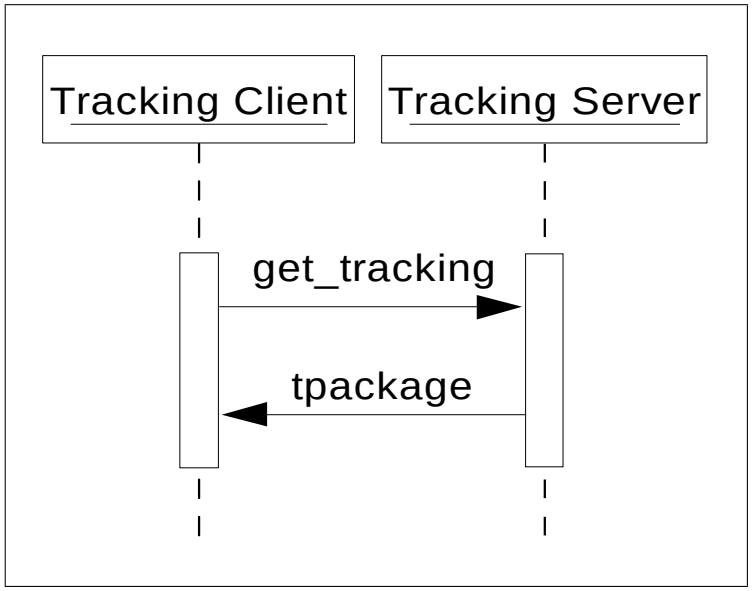


Figure 3.3: Tracker - Socket Communication Diagram

The basic commands implemented in the protocol are:

- "`get_tracking`". It is used to tell the server to fetch the latest tracking position and orientation and send a package with the gathered information over the network towards the client.
- The keyword "`error`". This is sent in case the tracking failed to inform the client that the tracking system went down or the communication between server and tracking subsystem failed.

As already mentioned, the server sends the `tpackage` as a character buffer. To use the received packages, the client must cast them back into the tpackage structure, to regain their functionality. As shown in scheme 3.4 the datagram has position and orientation members which can be accessed directly after the package has been cast.

In implementing the server - client model of the generic tracker, special attention needs to be given to the error cases. Socket creation, connection and communication is likely to fail, so the implementation needs to be encapsuled into try-catch expressions. The process of interconnection is structured in the following manner:

- First start the tracking server.
- The next step is to start the client. This tries to connect to the server's socket with the appropriate ip and port. Since the connection request can fail, in case the server was not able to bind the port or the client can not reach the server, the failure needs to be handled.
- If the socket connection was opened correctly the tracking tries to establish communication. For further failures this is also inserted into try-catch expressions which offer failure handling and compensation. Sending and receiving is done with the "`<<`" and "`>>`" commands, which simply forward a package of characters over the network. The commands sent between the server and the client are shown in the scheme 3.3: The client sends the keyword "`get_tracking`" to make the server fetch the recent tracking values and send a package.
- The server receives the command and tries to collect the required information from the tracking subsystems. In case the tracking fails the server sends the keyword "`error`", otherwise a `tpackage` with the tracked pose is sent.
- The return package waits on the socket till the client fills it into the buffer. Now the client needs to check if the return buffer is valid. This test is implemeted with a string compare to the already mentioned keyword "`error`". In case the tracking was correct, the return

```
datagram.h / tpackage:

struct tpackage {
float posx; float posy; float posz;
float ort0; float ort1; float ort2; float ort3;
};
```

Figure 3.4:

buffer is cast into the tpackage structure (diagram: 3.4) which then is accessable in the client.

For further computations on position or orientation, the quaternion and a vector library are included. This allows rotation and modifications of the returned vectors.

This implementation with a standalone tracking server is quite complicated, but it is required in special limited cases as in InView, described in chapter 5.

4 AMIRE

Authoring Mixed Reality, AMIRE, is a *authoring* tool to create and administrate augmented and mixed reality scenarios. It is used to create projects that contain information about a certain procedure. These scenarios are made up of geometric objects like text windows with annotations, simple control interfaces and other usefull components. AMIRE overlays a realtime video image with its virtual output to assist a worker in his task and tells what steps he needs to take. AMIRE is able to mark objects and give additional information to help perform certain actions or warn about difficulties. Workers are able to pick the scenario fitting the job or action they need to accomplish and can start working without searching for a documentation.

Usually AMIRE is installed on a tablet PC with an attached camera device. The worker sees the video image on the screen and AMIRE overlays this with the virtual elements. The worker can direct the tablet pc and the camera in any direction while AMIRE searches and marks the critical parts of the video image with the according information.

The environments AMIRE creates are used by workers to assist them getting through a certain process. To achieve this AMIRE recognizes certain objects or knows their position depending on the implemented and used tracking system. It attaches information boxes to the important objects depending on the step of the workflow. This makes AMIRE an interactive manual and working guide, naturally superior to a standard documentation that would only have images and lacks interaction with the worker.

Usually AMIRE works with a marker based tracking system. It implements a picture processing routine that searches a captured video image for a certain characteristic marker object. The *Generic Tracker* offers an approach that is the other way round: the position of the camera and its orientation is computed. The computed pose can be used in AMIRE to place the objects at the positions relative to the camera.

The scenarios and use cases implemented to demonstrate the capabilities and possibilities of AMIRE are described in detail in section 4.7. Further, for general use of tracking and more basic information, have a look at paragraph 2.1.)

4.1 The Execution Modes

AMIRE combines authoring tool and user application. Launching it without any parameters brings up the normal graphical user interfacet which is used to model and create the scene and its controls. Changing the start values can call AMIRE in a use mode where only AMIRE's output and the implemented controls are visible.

1. The *Operator Mode* (picture 4.1) displays all the elements of AMIRE's framework. These can be used to alter the scene's components and change the scenario's work procedure. This is the way an operator would use AMIRE to add new scenarios or modify and update existing ones. The elements and control interfaces shown in this mode should not be accessable for the normal user.

2. The *User Mode* (picture 4.1) is the way a worker sees AMIRE. It only displays the components the operator added to a scene, which are located in the output window. That is usually the video output, the objects offering controls and user interaction, as well as those that give information to the user. Detailed information about the components is contained in the following section.

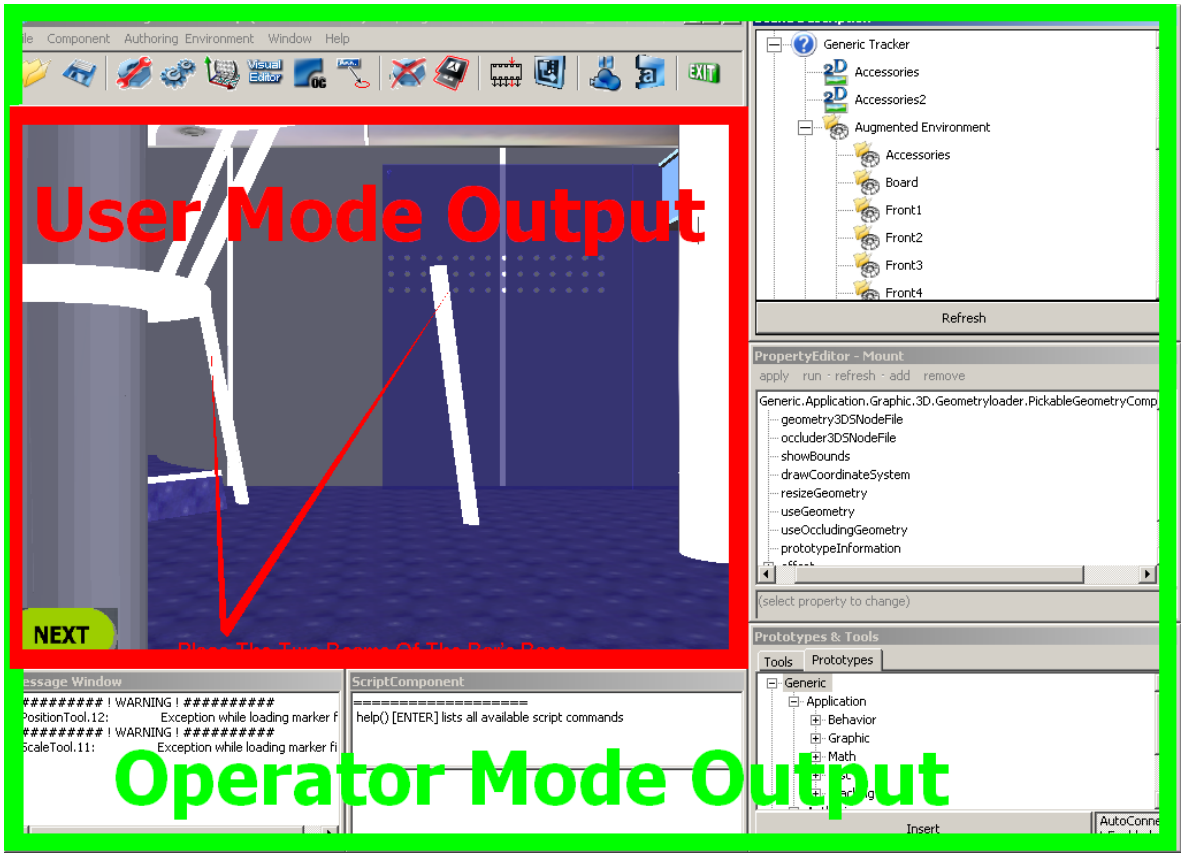


Figure 4.1: The User And Operator Execution Mode Output

More information on the different elements and windows of the *Operator Mode* are given in section 4.2.3, where they are introduced and their functionality and use is explained.

4.2 The Component System

Users of AMIRE need not know about the details of components that can be added to the scenarios, but for an operator this is crucial. AMIRE implements lots of different components which have either input, output or arithmetical / logical purposes. Although they follow different tasks, they share some common concepts.

4.2.1 Component Structure

The component model AMIRE implements is based on interconnections. Every component has several in- and outslots which can be connected to exchange and forward information.

The *Framework Components*, which represent the graphical user interface (GUI) and other fundamental application parts, are statically connected and can not be changed. These components offer services to the operator.

The *Application Components*, which make up the current project, are manually inserted and connected. Creating such an object places it into the scene but leaves its slots empty. These are handled with the *Connection Editor* 4.2.3. This framework component service allows interconnection of components. This is done to realise certain behavior or output.

Three dimensional objects require a position and orientation in order to be placed in the scene, so such components need a connection to a component offering such a value. Two dimensional objects either have an input for the two coordinates or use their offset attributes. For example, it is possible to attach a textured cube (3d object) to a computed position of a tracked device by interconnecting the transformation inslot of the cube with the transformation output of the tracker component. This will place the object into world space at the assigned position and orientation. Mathematical or logic components help influence the data before it reaches an inslot and are usually inserted between two other components.

4.2.2 Component Hierarchy

1. Authoring:

   Components to handle the objects and to administrate the current scene are part of the basic system and available by default. It is not possible to add, remove or change them. All the GUI elements are also part of these components which allow even the unsophisticated user to create and modify scenes.

2. Application:

   These components are addable and usually make up the scenario. The user can select components to place them into the current scene and interconnect them to build up the scenario's requested environment and the augmented elements. The application components are grouped by their different objectives and support either *mathematical* or *logic* computing, *displaying* 2 or 3d models or offer user *interfaces* and control elements.

   - Behavior: a state machine to create states and workflows. It has several in- and outslots to allow complex stepping and branching. (see section 4.3)
   - Graphic: everything that is displayable, ranging from simple models in 2D or 3D space with certain textures to user interface elements like buttons and levers. The operator employs these elements to realise the interaction with the user. Text boxes and graphical objects offer output to the user, while levers, buttons and switches implement the input.
   - Math: offering complex computations on matrices, simple mathematical equations and logical operations on boolean values. These components are often switched in between to modify values. They neither have user interaction nor a graphical representation in the scene.

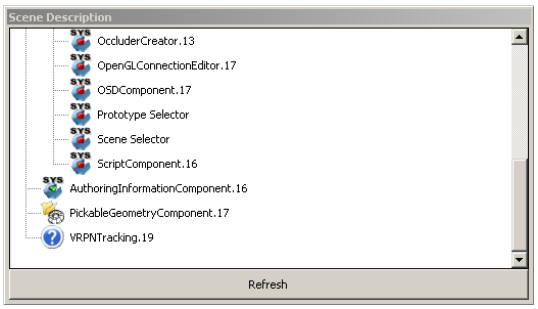


Figure 4.2: The Scene Description Window

- Misc: components that do not fit into the other groups. Offering services ranging from screenshot tools to audio output devices.
- Tracking: the components offering a position and an orientation. There is no graphical representation, the components work in the background offering their result and information on one of their outslots.

4.2.3 Framework Components

Framework Components are AMIRE's fundament, offering all kinds of different services and are taking care of the environment. These components are hidden from the user and only accessible for the operator. The different elements are:

- *Scene Descriptor* (picture 4.2.3), giving an overview of the current scene. It lists all elements of the current scene, the *Application*, the *Authoring* and the *Misc Components*. It allows selection of a single application component to modify its properties in the *Property Editor*. In case the chosen component has a matrix component, the *Matrix Editor* allows changing of the matrix's rotation, scaling and translation. Marking two application components allows the user to modify their interconnections with the connection editor (picture 4.2.3).
- The *Property Editor* (picture 4.2.3) gives access to the properties of the currently selected component. It allows the type specific settings of this component to be altered. It influences only the selected component and not all the components of a certain type. In the *Property Editor* it is possible to specify all attributes wich are often offset translations, rotations, model files, textures or display messages. This is the place to specify the inserted component to fit its task.
- The *Matrix Editor* (picture 4.2.3) handles the modification of translation or offset matrices. It offers a graphical user interface (GUI) to modify the selected component's translation, scaling and rotation.
- *Prototypes & Tools* (picture 4.2.3) is used to add new components. The component is chosen from the types folder and the *insert* button pressed. Usually component's attributes are changed in the property editor to specify its values. The final step is to connect it to other components, depending on the components slots and function. This is done by selecting the two objects with the connection editor (picture *Connection Editor*).

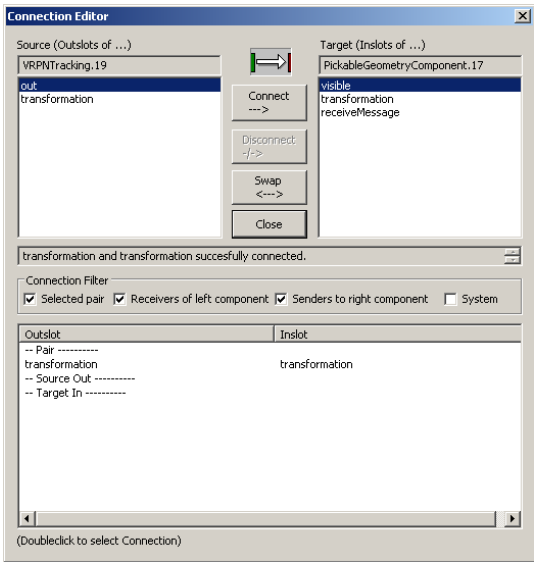


Figure 4.3: The Connection Editor Window

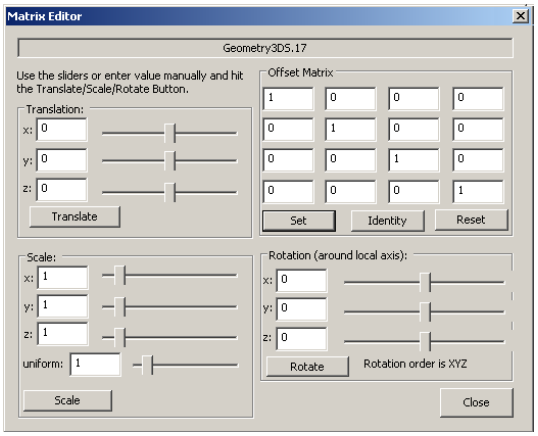


Figure 4.4: The Matrix Editor Window

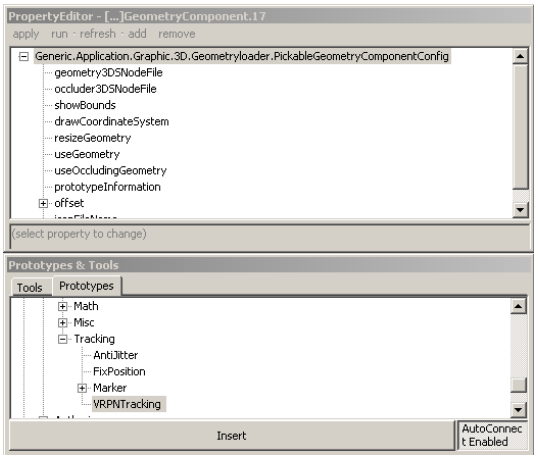


Figure 4.5: The Prototypes & Tools Window

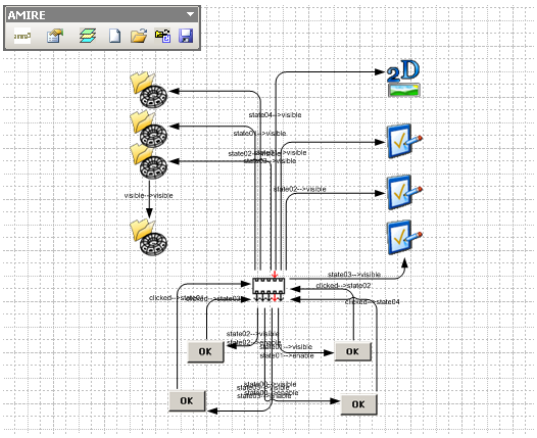


Figure 4.6: The Scene Imported Into Visio

- Some framework components can be accessed from the main control bar, such as the XML export (chapter 4.2.4) or AMIRE's state machine (chapter 4.3).

4.2.4 Exporting and Importing the Scene

After modeling the scene with the authorising components, it is possible to save the scene with an XML export. This generic output offers various possibileties to administer the scene in either self written XML parsers or with the use of Microsoft Visio. AMIRE has some *Visual Basic Script* (VBS) macros included to import / export the scene into Visio, where it can be displayed and modified. As yet, the implementation of this export is not of good quality. At the moment there is no sense in using it for large scenes. The different steps of the state machine are not displayed on different pages, so the export is really chaotic in Visio. A simple scene imported into Visio is displayed in figure 4.6.

4.3 The State Machine

The *State Machine* is a fundamental component for creating a new scene. To allow AMIRE to guide through a workflow, the framework offers a state machine. It has two lists of slots to, on the one hand, connect it to logic components that switch between different states and, on the other hand, a group of components that are influnced in the selected state. Influencing usually means certain components are displayed and others are hidden when a step is activated. This is used to follow the process step by step and to keep the user updated with valid information on the task he currently tries to accomplish.

The state machine's user interface is shown in picture 4.7. There is a listing of all existing steps which each can be selected and modified. The interface allows to characterize the steps by assigning new names and hitting the *Confirm New Name* button. Another button, the *Actiuvate Selected State*, changes the scene's state to the currently selected one. *State Activators* brings up a list of slots, usually user interface components, that cause the state machine to proceed to a different step. To finish the state machines implementation it has another list, the so called *State Dependants*. They represent the elements influenced by the

current state. This is basically used to attach the visual slots of elements to a state to make them display while the unconnected elements of the other state stay hidden.

Summing up, the state machine is in control of the scenes process. Usually the steps influence a list of components by displaying them and hiding the rest. Each step has activators to jump to a new step.

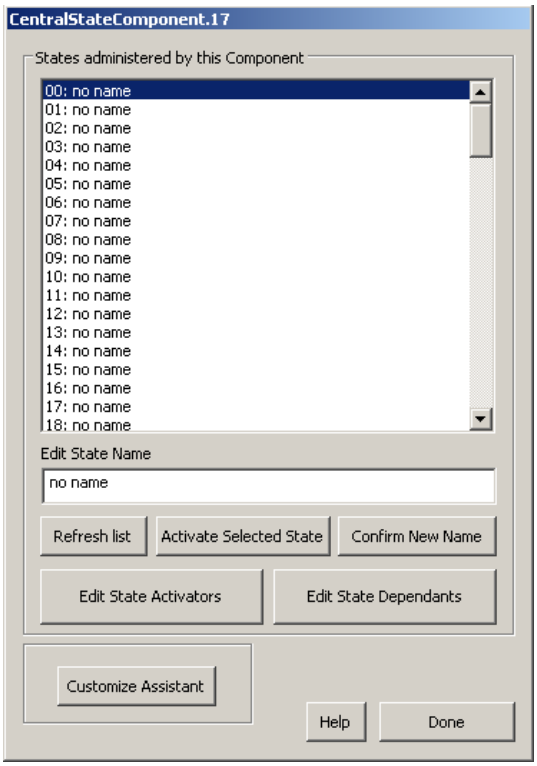


Figure 4.7: The State Machine Control Window

4.4 The Component Writing Concept

This section outlines the basic concepts a programmer needs to consider when wanting to create or modify components. It is usually wise to only change the application component and leave the framework untouched. This guarantees an easy portation of the created component into later versions of AMIRE.

AMIRE is open source. Adding or modifying components is done directly in the AMIRE project. That means it has to be rebuilt after any modification and that components can only be inserted if you have the current source code. There is a *Component Writer Guide* ([AMIRE, 2004]) added in the source code distribution to outline the essential steps in creating a new component.

4.4.1 Component Types

For the programmer there are only two different component types.

- Display components appear in AMIRE's output. Again there are two different ways to create these objects:
  - The fast way for simple objects is to use the `displayCallback()` function which executes OpenGL code. This display function is called in four passes:
    1. `PREPARE RENDER MODE`: the mode is added to prepare the coming calls
    2. `SETTING MODE`: the mode to make all the settings, for example the light-sources and the render modes
    3. `OPAQUE RENDER MODE`: the mode to draw non transparent objects
    4. `BLENDING RENDER MODE`: final mode to draw the semi transparent objects and the 2D objects
  - The preferable way to draw complex models or meshes is to use the Open Scene Graph (OSG), which is part of AMIRE's 3d engine. OSGs are trees that carry geometric information in their nodes (`osg::Node`). These again can be roots to new OSG trees. There are numerous different node types offering meshes and geometry information, translations and other settings. In such a tree an entire geometric environment with lots of elements can be stored. The OSG library offers routines to modify the nodes, to walk through the scene or to draw the entire tree (consult: [OpenSceneGraph, 2004] for detailed information). To create a new displaying component a new node can be added into AMIRE's already existing OSG tree. The OSG library is capable of parsing many different file types (like `.obj`, `.3ds`, `.flt` etc.) and transforming them automatically into an `osg::Node`. This is done with the `osgDB::readNodeFile()` function.

  Display components inherit the position matrix. This can be changed by using the frameworks matrix editor. Modifying this so called *offset* matrix moves, scales or rotates the object. Also the other type dependant properties of the object can be edited, which can influence texturising, texture scaling, lighting and much more. There are no rules or limits when using attributes to control the newly created component.

  Beside changing the attributes, a display component can be influenced by attaching its translation and visibilety inslots to the outslots of a component that offers appropriate data, like, for example, a translation matrix. These forward their output through the display component's inslots. The display components use the received data in their internal drawing routine. In this way it is possible to promote the offset matrix from a tracking component forward to other objects to position them in the scene.

- Functional components run their code in the `functionalCallback()` function. The computed information is forwarded to the outslots. In case these are connected to the inslots of another component the information is available there. Functional components usually do simple logic or mathematical equasions to change the forwarded information. Some of them create data on their own by calculating positions with a tracking.

Summing up, AMIRE uses a network of connections with its components representing the nodes. The interconnections forward information from outslots to inslots giving the scene the ability to have a complex behavior.

4.4.2 Inheritance

The above mentioned concepts have influence on the way components are created. The concept of interconnecting all the different components means, that some unitary standard must exist, how components are constructed. If a component does not show the correct behavior the whole system might crash. Additionally, it confuses when using a new component if it is not behaving the required or expected way. So, usability and communication structure force components to inherit most of their functionality from generic components deeply rooted in AMIRE's framework.

Components the programmer adds are added in the *genericComponents* project part of AMIRE, while the basic components they are inherited from, are located directly in AMIRE's framework.

4.5 Implementing the Tracking Component

The *VRPNTrackingComponent* is inherited from the generic `Component`. It implements the additional methods and constructors the framework requires.

4.5.1 VRPN Tracking Component

- Constructor: It initiates the Property type so it has an entry in the *Prototypes & Tools* section. In a second step it creates the components attributes, the transformation matrix and a boolean and adds them to the outslot. The inslot is initialised, but at the moment unused.
- Destructor: Unloads the tracking library by calling `FreeLibrary()` on the DLL
- newInstance: Loads the DLL by calling `LoadLibrary()` and maps its services to function pointers by calling `GetProcAddress()` on the DLLs exported functions. Also it starts the tracking system with the `InitialiseTracking()`. See chapter 3.3.2.
- reinitializeEngine: Reconnects the out and inslots.
- functionalCallback: Performs the basic tracking. Calling `UpdateTracking()` computes the new position and orientation. The translation matrix is updated with the `GetTrackingValue()` function. Afterwards the results are forwarded to the outslot. See chapter 3.3.1.
- emitOutSlotProperty: Connects the translation matrix to the inslot of a different component. This component will be positioned and oriented according to the input matrix. This operation forwards the tracking.
- receiveInSlotProperty: Unused.

4.5.2 Registering new Components

The component is registered at the AMIRE framework by modifying AMIRE's *Generic Components* project (the class *genericLabeinComponents* registeres new components). After including the header files a new instance is created with `amire::component::Component`. Next the component is registered on the component manager with `componentManager ->`

`registerComponentPrototype`, becoming available in the *Tools & Prototypes* window as seen in picture 4.2.3.

The *Generic Trackers* created to track with VRPN can be found with the prototype name *VRPN_ Tracking* in the prototypes & tools component (4.2.3).

4.5.3 Performance and CPU Time of Components

Including slow computations is not a good idea since the callbacks are time critical. Outsourcing of different tasks into multi-threaded DLLs is a way to keep AMIRE's augmented reality engine fast and reliable. This is the way the *Generic Tracker* was written and which is implemented into the *VRPN Tracking Component*.

The *VRPN Tracking Component* loads the DLL by calling `LoadLibrary()`. A second step is to extract the services with the `GetProcAddress()` which assigns the functions to pointers. After this you can call the DLL's exported functions and starts running in a seperate thread.

Basically the component loads the DLL to run the tracking in a second thread. This allows to replace the VRPN Tracking DLL without chaning AMIRE's source. That is very important on a huge and complex program as AMIRE whose compilation takes hours.

4.6 Using the Tracking

As mentioned in chapter 2.3, in this project the camera is the tracked object. Having its position and orientation allows to compute the translation from the coordinate null point. Multiplying the translation matrices of any object in the scene with the tracking components output places it into the scene exactly on the null point of the world coordinate system. Further translations to the objects can be done by changing their properties.

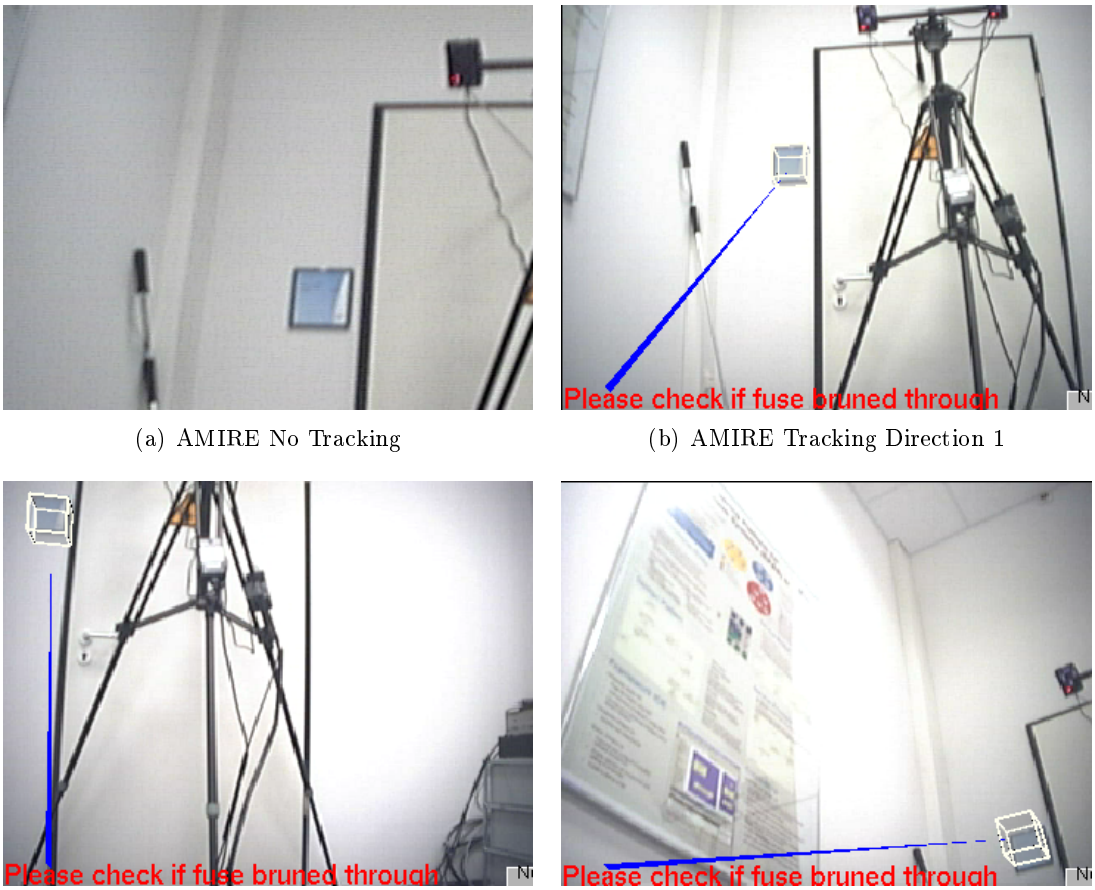
4.6.1 Building up the Tracked Scenegraph

You can add objects to the scene by connecting their translation inslot to the matrix outslot of the tracking component. This is done with the *Connection Editor* 4.2.3. The connection is internally realised as matrix multiplication of the inslot matrix and the internal matrix of the 3d object.

The scene is built up as a tree, the root of which is the tracking component. All display objects that need to be tracked are attached to it. The displayed objects can be moved to their final position by changing their internal offset matrix with the *Matrix Editor* 4.2.3. Changing the translation of the x,y and z axis would move the object along the predefined axes of the world coordinate system. These axis were initially calibrated with the tracking systems.

4.6.2 Matching Video and Computer Graphics

Since we are combining the camera view with certain graphical elements, the video image and the graphics must correlate. The movement of the camera is tracked and the translation matrix of the scene graph root is changed automatically. So the tracking keeps the objects positioned correctly in the space.



(a) AMIRE No Tracking

(b) AMIRE Tracking Direction 1

(c) AMIRE Tracking Direction 2

(d) AMIRE Tracking Direction 3

Figure 4.8: AMIRE Tracking Example

Camera projection and the openGL projection matrix are identical (field of view (FOV) 40 degrees from the specification of the digital camera [Stricklc, 2001]) resulting in a quite good match of camera video stream and the virtual 3d graphics.

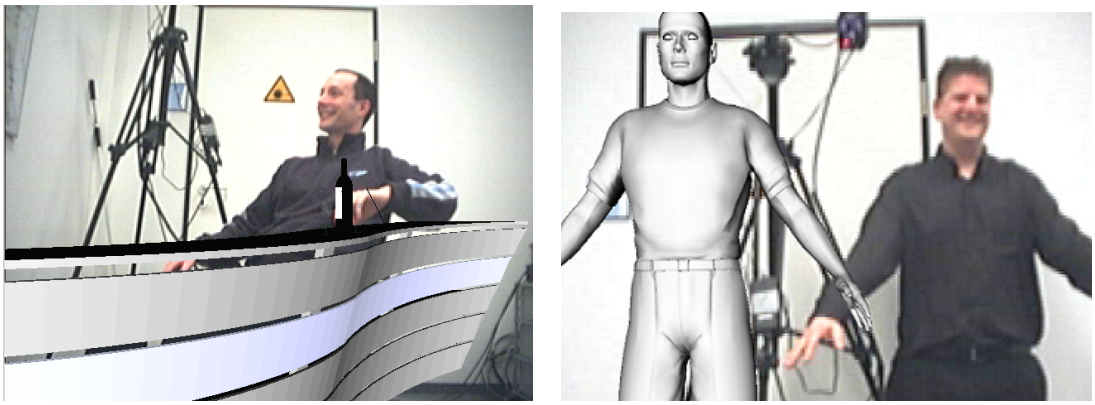
4.6.3 Tracking Performance

The capability of the tracking system is shown in the following screenshots from AMIRE. The first shows the unaugmented sceme. the room with a door and a blue sign at its side (figure 4.8(a)). In the demo we track the sign, assuming it is a fusebox, which should be checked. The tracking works at 60Hz, which brings an update every seventeen milliseconds, and is very precise as the pictures 4.8(b), 4.8(c) and 4.8(d) illustrate.

4.7 Use Cases

4.7.1 Augmented Reality

The possibility to track objects allows one to augment reality. AMIRE supports the use of really complex 3d files by using the *OpenSceneGraph* library. You can see a low polygon



(a) AMIRE Simple Model (b) AMIRE Complex Model

Figure 4.9: AMIRE Augmented Reality

example of a bar in figure 4.9(a) or a highly complex model in picture 4.9(b).

4.7.2 Guided Workflow Scenario

The main goal of this scenario was to show what is possible to track the camera instead of objects in the scene and to derive the position of objects in the scene. This all was implemented into a work guiding project to illustrate the capabilities. For detailed information and use of this demonstration refer to the modified AMIRE version on the CD.

5 Intrace

5.1 General Information

5.1.1 Raytracing

Intrace is a raytracer designed for the use of cpu resources on distributed networks. Raytracing's main advantage over usual computer graphics is a physical correct result in the scenes. There are two fundamentally different types of raytracers. The first is to track the light cast from lightsources through a scene, to compute the output. This is the so called backward raytracing. The *InView* raytracer implements a forward raytracing and follows the direction of the ray starting at the camera. It is cast from the camera and is heading in the direction the camera is pointed at. When it hits an object the ray is changed, depending on the material of the surface. The changes on surfaces are described with material shaders, which simulate a certain physical effect. This allows to compute reflections, refractions and shades in an incredibly realistic and exact way. It offers an easy way to implement refractions of glass, shading and shadows as well as glare. Usually computer graphics is working on OpenGL or DirectX. These are using pixel and vertex shaders which try to imitate physics by reducing the complexity of the effect with approximation. Normal shaders are optimized for games and try to produce good looking output at high framerates, which means they can not be realistic at such a degree.

In short, ray tracing can be used to give a realistic impression of how objects will really look without constructing them. This leads to a faster development and the possibility to change construction and design in real time. Another very appealing idea is to use raytracing to find negative effects of lights and glare to increase the usability. Calculating where reflections show up gives the chance to reduce disturbing glare on screens and reflecting objects. Raytracing reduces cost-intensive prototype construction and averts negative surprises.

Raytracing is based on the idea to follow the line of sight in a certain direction. Since the eye has only a certain angle of sight the rays are only sent in a certain direction and through a certain volume. To simplifice this and to fit the needs of a rectangular window the volume is a pyramide frustum. The rays sent through the scene intersect with objects. These carry shaders, small algorithms describing what influence the surface has on the ray and what color it returns.

5.1.2 Tracking in Raytracing

Advantages

The first idea now was to implement tracking into the raytracer. A better immersion should be gained by using the tracking as an input device. Raytracers with high resolution output consumes a lot of CPU power so the output jerks often. This makes navigation very inaccurate and tricky. A high frame rate solves this problem, but it might not always be possible to access the needed resources and to combine 30 CPUs. To save expensive resources and still have a

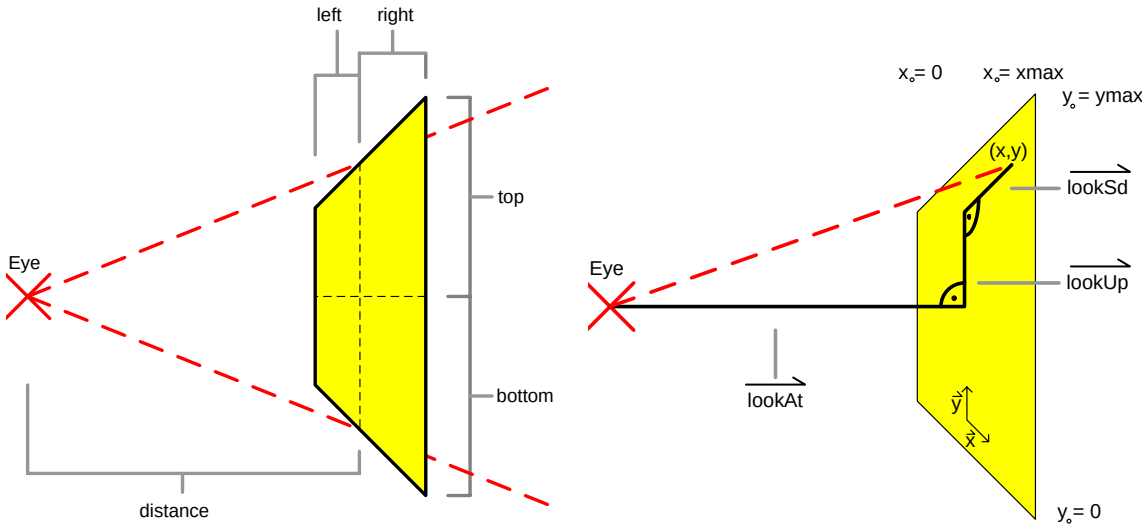
(a) Example 1 (b) Example 2

Figure 5.1: Intrace Raytracing

good feedback when controling the camera, it is practical to track flysticks, headmounted displays or other possible input devices. Tracking offers a very good feedback of what movement is made which is increasing the usability.

5.1.3 Camera

As already mentioned, raytracing is a physical approach to computer graphics, so that effects such as reflections, refractions and shadows arise automatically. The camera object of a raytracer just decides form which point the ray is cast and which direction if follows by defining a volume, a frustum, which is traversed by the rays.



(a) Camera Frustum (b) Ray hitting the Viewplane

Figure 5.2: Raytracing Camera

The geometric concepts behind the raytracer camera are illustrated in figure 5.2. The camera is based on vectors to represent the rays. The eye vector is the starting point from

which the rays are cast. These rays penetrate the viewplane and traverse the scene, calculating data when hitting objects by running their assigned shaders.

Camera Creation

In the *Camera Creation* the ray-independant vectors of the camera are created and scaled. They are needed to compute the ray's direction and specific vectors in the *Ray Creation* section (as described in chapter 5.1.3). The camera creation algorithm implements the frustum with its direction and opening angle. The frustum and its composition with the basic vectors are displayed in picture 5.2(a). The computed directional vectors that define the camera coordinate system are shown in figure 5.2(b). This coordinate system is essential for a fast ray creation routine.

1. Calculate the three coordinate vectors:

   The $\vec{lookUp}$ and the $\vec{lookAt}$ vectors are given by the camera parameters. As seen in 5.2(b), the third vector, $\vec{lookSd}$, is perpendicular to the other two vectors and can be computed with a cross product:

   $$\vec{lookSd} = \vec{lookAt} \times \vec{lookUp} \tag{5.1}$$

2. Scaling:

   Distance scales the $\vec{lookAt}$, hight the $\vec{lookUp}$ and width the $\vec{lookSd}$ vector:

   $$\vec{lookAt} = distance \cdot \vec{lookAt} \tag{5.2}$$

   $$\vec{lookUp} = (top - bottom) \cdot \vec{lookUp} \tag{5.3}$$

   $$\vec{lookSd} = (left - right) \cdot \vec{lookSd} \tag{5.4}$$

Ray Creation

Ray creation computes the direction vector from the eye towards every point on the viewplane. The viewplane represents the window that is later shown on the screen. The geometric basics are outlined in figure 5.2(b). The x and y values in the viewplane specify the rays and range in these intervals:

$$x \in [0, xmax] \wedge y \in [0, ymax] \tag{5.5}$$

To compute the direction vector for the corresponding x and y coordinates, the following equation is solved:

$$\vec{dir} = \vec{lookAt} + \frac{ymax - y}{ymax} \cdot \vec{lookUp} + \frac{xmax - x}{xmax} \cdot \vec{lookSd} \tag{5.6}$$

This function is called for each x,y-pair in the viewplane to determine the direction of the ray that penetrates this point. The ray's source is always the same, the eye, which is computed in the camera creation routine (5.1.3).

5.1.4 Tracking in the Camera

General Idea

The eye vector is the position and in the tracking it will simply be translated with the position vector returned by the *Generic Tracker*. The $\vec{lookAt}$ vector in computer graphics initially looks along the negative z axis and is rotated with the orientation the tracker computes. The same occurs with the $\vec{lookUp}$ vector, initially looking along the positive y axis. The $\vec{lookSd}$ vector is automatically rotated since it is computed with the vector product of the other two vectors (see above in section 5.1.3).

Implementation

The *Generic Tracker* written for AMIRE is also portable to Linux. Intrace is closed source and allows only the changing of small parts by dynamically loading them at runtime. It is possible to insert different material or camera shaders as shared objects. This permits a certain degree of influence on the raytracer, but still blocks most serious modifications. The first approach to implement the tracking was the idea to load a camera shader that uses the tracking as a shared object. This failed since Intrace's enviroment did not allow the VRPN communication sockets (introduced in chapter 2.4). To circumvent this problem a small socket based implementation is offered that allows the control and the use of a tracker even in a closed environment such as InView. The different configurations of the generic component are described in detail in section 3.1.2.

Accordingly, a camera shader was written implementing the theoretical basics of section 5.1.3, one which implements frustum and the coordinate vectors was written. The next step was to implement the tracking. Using the tracked position is simply achieved by moving the eye position to a different location. This is done by translating the eye vector with the computed offset. The orientation the tracking returns can be integrated by rotating the cameras coordinate system vectors. In the computer graphics standard case, the $\vec{lookAt}$ vector is oriented along the negative $\vec{Z}$-Axis, the $\vec{lookUp}$ vector points along the positive $\vec{Y}$-Axis and the $\vec{lookSd}$ is heading towards the positive $\vec{X}$-Axis. The first two vectors are rotated with trackings newly computed orientation and determine $\vec{lookSd}$ with their cross product.

The communication implementation is outlined in picture 5.3.

5.2 Stereoscopic Raytracing

5.2.1 Basic Idea

The easy approach to creating a tracked camera for InView led to a new thought: is it possible to render a stereoscopic output for two eye positions?

The theoretical basics are taken from chapter 5.1.3 which is also illustrated in picture 5.2. The idea is to duplicate the window size along the $\vec{X}$-Axis and plot the normal output twice in the new window. The left and right side of the new window will have different sources and orientations of the rays. The geometric concept of how to alter the camera is explained in picture 5.4.

The InView traces the left half of the screen first, then changes the ray origin by translating the eye a bit to the side and proceeds to render the second part. The resulting screen is a single window with the scene the left eye sees on the left side and the right eye picture on the

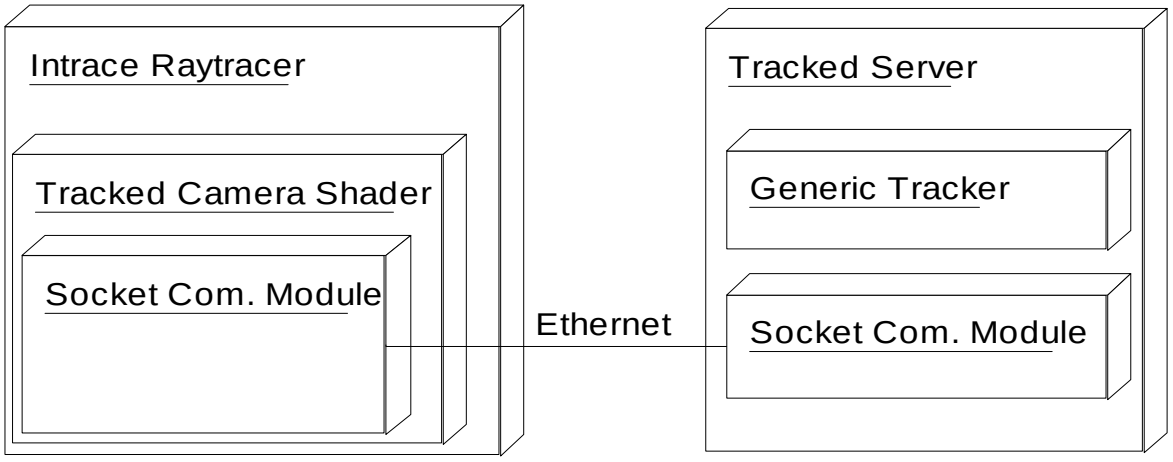


Figure 5.3: Communication within the Tracked-Camera Shader

other. The vectors of the camera coordinate systems do not change. The direction is only influenced by the new x and y values since they now have a different range.

$$x \in left : [0.25 * xmax, 0.75 * xmax] \wedge y \in [0, ymax] \tag{5.7}$$

To compute the direction vector for the corresponding x and y the following equation is solved:

$$\vec{dir} = \vec{lookAt} + \frac{ymax - y}{ymax} \cdot \vec{lookUp} + \frac{xmax - x}{xmax} \cdot \vec{lookSd} \tag{5.8}$$

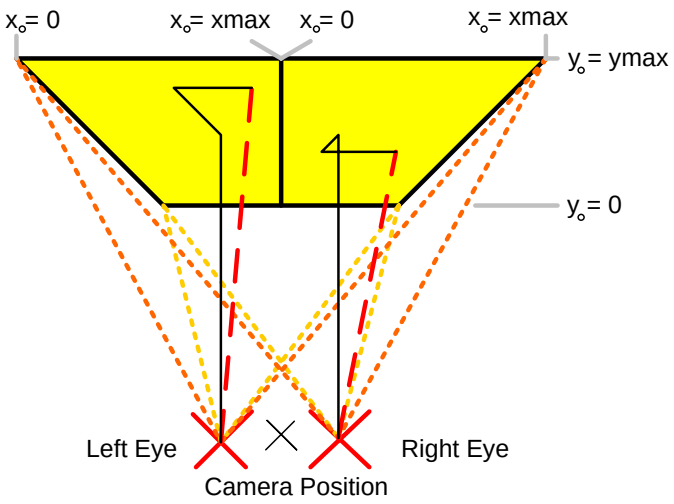


Figure 5.4: Stereoscopic Raytracer Camera

The technical implementation to use this output as a stereo 3d scene is accomplished with a desktop extension to fill two screens. The result is not visible on a normal computer, since it shows two screens with a slightly different view. For the 3D view a Head Mounted Display (HMD) was used that positions a small LCD screen infront of each eye.

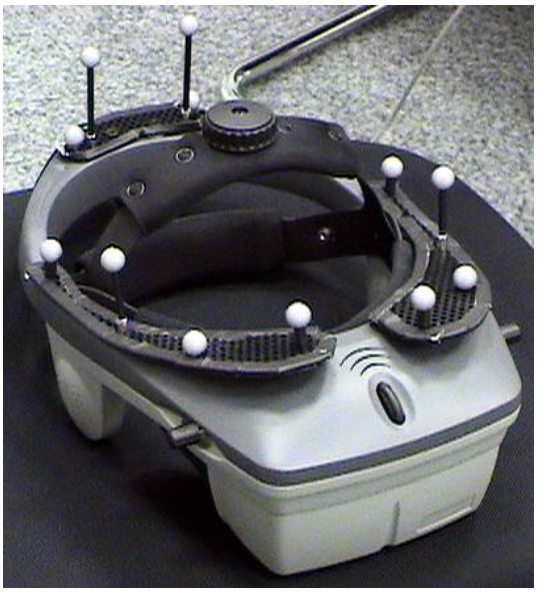


Figure 5.5: Head Mounted Display (HMD) with Infrared Markers

Rendering the raytracer on full screen moves the left half of the rendered output to the left screen of the HMD and places the right part on the other display side attached infront of the right eye. This creates a stereoscopic picture for the user. The HMD is trackable with attached infrared markers for use as input control device. That means when the user's head moves, the raytracer camera is moving along. For better understanding of how a stereoscopic output looks like, refer to picture 5.6. Since the distance between the eyes is small, the differences in the picture between left and right are also slight.

This implementation of raytracing into a virtual reality environment offers engineers and designers the chance of seeing the product beeing generated in a realistic and accurate way.

5.2.2 Performance

Although the newly designed camera makes a couple of new computations the CPU time per pixel is less. The tracking routine is only called once per frame and therefore is not as time critical as the pixel processing routines. Anyway, tracking only takes a few milliseconds. The ray casting routine is now fully optimized and is about 20% faster than the original one, which is also increasing the framerate by one fifth.

The stereoscopic view renders twice the number of pixels for the same window. This naturally divides the usable screen size, so raytracing with stereo needs twice the resolution (screen left+right) and, thus, still looses a lot of the original frame rate.

These results are average values, because it is difficult to let the stereo and the mono camera render exactly the same scene and situation.

Figure 5.6: Stereoscopic View of a Tornado Cockpit

Bibliography

[AMIRE, 2004] AMIRE (2004). Component writer's guide. http://www.amire.net/.

[Gibson, 2004] Gibson, B. (2004). Numerical methods: Interpolation and curve fitting. http://www.maths.usyd.edu.au:8000/u/billg/MATH2052/tutorials/tutorial10/tutorial10.html.

[Kalies, 2003] Kalies, B. (2003). Notes on least-squares approximation. http://www.math.fau.edu/kalies/mad3400/lab4.pdf.

[Kato and Billinghurst, 1999] Kato, H. and Billinghurst, M. (1999). Marker tracking and hmd calibration for a video-based augmented reality conferencing system. *Proceedings of the 2nd International Workshop on Augmented Reality (IWAR 99)*.

[Klaus, 2004] Klaus (2004). *ARTtrack & DTrack Manual*. Advanced Realtime Tracking, www.AR-Tracking.de, 1.22 edition.

[Kolb, 2002] Kolb, A. (2002). Geometrische Modellierung & Animation I.

[Miller, 1999] Miller, K. (1999). Creating and using dlls - using the dll (without an import library). http://www.flipcode.com/articles/article_creatingdlls.shtml.

[OpenSceneGraph, 2004] OpenSceneGraph (2004). Osg project documentation. http://openscenegraph.org/osgwiki/pmwiki.php/Documentation/Documentation.

[Stricklc, 2001] Stricklc (2001). *USB PC Video Camera Toucam XS*. Phillips.

[Taylor and Yang, 2004a] Taylor, R. and Yang, R. (2001-2004a). Virtual reality peripheral network. http://www.cs.unc.edu/Research/vrpn/.

[Taylor and Yang, 2004b] Taylor, R. and Yang, R. (2001-2004b). Vrpn from the application's point of view. http://www.cs.unc.edu/Research/vrpn/app_point_of_view.html.

[Wormell, 2003] Wormell, D. (2003). *Product Manual for use with InertiaCube2 Serial and USB Interface*. InterSense, www.isense.com.