

Uppsala Master's Theses in
Computing Science 238
Examensarbete TF3
2002-12-12
ISSN 1100-1836
UPTEC F03 022

Components for Application Sharing in 3D Graphical Environments based on VNC

Robin Stridh

**Information Technology
Computing Science Department
Uppsala University
Box 337
S-751 05 Uppsala
Sweden**

**This work has been carried out at
Uppsala University
Department of Information Technology
Human-Computer Interaction
Box 337
S-751 05 Uppsala
Sweden**

Supervisor: Stefan Seipel
Examiner: Stefan Seipel

Passed: 030217

Abstract

This Master's thesis describes a method for using traditional 2D applications in interactive 3D environments. By using Virtual Network Computing (VNC), which is a system for application sharing over the network, the applications can also be shared between several users on different physical computers. This paper describes the implementation of a generic library of functions for communication with a VNC server, and also specific implementations of 3D VNC clients for the Virtual Reality Toolkit (VRT) and the Visualisation And Simulation Environment (VASE).

Contents

1	Introduction	5
2	Previous work	5
2.1	Application sharing over the network	6
2.2	Virtual Network Computing - VNC	6
2.3	The Remote Frame Buffer Protocol	8
2.4	Performance of different VNC servers	10
2.5	Using 2D applications in a 3D environment	11
2.6	VRT - Virtual Reality Toolkit	12
2.7	VASE - Visualisation And Simulation Environment	12
3	Design issues	13
3.1	First approach: Redirecting an output stream from vncviewer to a shared memory, read by a separate process	14
3.2	Second approach: Using components from vncviewer	15
3.3	Third approach: Rewritten, system independent RFB library functions	15
3.4	VRT VNC Window	16
4	Implementation of the RFB Library	17
4.1	Types and Endians	18
4.2	Initial handshaking and Authentication	19
4.3	The Update functions	20
4.4	Mouse and keyboard interaction	22
4.5	Closing the connection	23
4.6	Version 1 - blocking	23
4.7	Version 2 - non blocking	25
4.8	Version 3 - subtexturing	27
5	Implementation of VRT VNC Window	31

6	Implementation of the VASE Plug-in	32
7	Results and Conclutions	34
7.1	Blocking vs. non-blocking read of update messages	34
7.2	Replacing the whole frame buffer vs. replacing individual rectangles	35
7.3	Performance of the final 3D VNC client	37
7.4	Usage situations	43
8	Future work	45
8.1	Further optimisation of the library functions	46
8.2	API functions for the GLUT platform	46
8.3	Evaluating the possibilities for other types of usage	46
A	RFB reference manual	49
A.1	RFB_state Struct Reference	49
	A.1.1 Detailed Description	49
A.2	rfb.h File Reference	52
	A.2.1 Detailed Description	52
	A.2.2 Function Documentation	55
B	VRT VNC reference manual	72
B.1	vRT_VNCWindow Struct Reference	72
	B.1.1 Detailed Description	72
B.2	vrt_vnc.h File Reference	73
	B.2.1 Detailed Description	73
	B.2.2 Function Documentation	75
C	An example of a VRT VNC application	82
D	Pixel formats	85
E	VNC Server Usage	85

1 Introduction

While 3D computer graphics is mostly used in games and entertainment applications, many educational and scientific applications can benefit from 3D visualisations. Especially in medical science and engineering 3D graphics has been used for quite some time.

In many 3D-applications, there is a desire to reuse existing 2D applications. Both since most currently available applications are in 2D, and would be expensive or difficult to convert into 3D applications, but also since some applications that will be used in 3D environments are naturally more suited for 2D. Those include for example text editing and web browsing, applications that can also be given a new perspective if used in a 3D environment.

There is also a desire to share applications between users, in order for many users to be able to work together with the same application on different computers. Especially in educational scenarios, there is often a need to share a common environment for demonstrations etc.

A way of combining the two goals above is described here. The solution is based on a shared, virtual 2D terminal that can be used by several users on different physical computers, but mapped into a virtual 3D environment.

For ease of use, some Application Programmers Interface (API) functions for the Virtual Reality Toolkit (VRT) have also been implemented. To evaluate the function library developed during this thesis work in a real usage situation, a plug-in application has also been written for the Visualisation And Simulation Environment (VASE). VRT and VASE are described briefly in the Previous Work section below.

2 Previous work

There has been a lot of work done on application sharing over the network, but very little on using 2D applications in a 3D environment. This section will focus on Virtual Network Computing (VNC), that is a system for sharing applications over the network, and that has been used in this project. It will also, very briefly, describe some projects dealing with 2D applications in 3D environments. At the

end, there will be a few words about the Virtual Reality Toolkit (VRT) and the Visualisation And Simulation Environment (VASE), because this project is primarily designed to work in these environments.

2.1 Application sharing over the network

There are basically two approaches to application sharing over the network where both are based on the client-server model. In both cases, we assume the application is run at the server.

The first approach is to let the server send all graphics commands directly to the client and let the client perform the graphics processing. This is, for example, done by the X Windows system [1]. This approach assumes that the client has the same programs installed as the server or that it somehow knows exactly how to handle all specific graphics commands. It can also be difficult if the server and client are running different operating systems. If done smart, the approach mentioned above can be implemented using very little network resources, but demanding a lot of functionality at the client side.

The second approach, which is used by VNC, is to perform all graphics processing directly at the server and send the appearance of the frame buffer to the client. This will demand very little functionality from the client, but will at first sight cause a lot of network traffic. As shall be explained later, this need not be the case though since the system can make use of very efficient encodings of the data. This will however require a lot of functionality from the server and the operating system the server works on, but will in contrast require very little from the client.

2.2 Virtual Network Computing - VNC

Virtual Network Computing [2] is a system for sharing applications over the network by sharing the whole frame buffer. The system was developed by people at the AT&T Laboratories Cambridge in 1999 as a tool in their daily work. It is based on the functionality provided by the X window system that allows displaying X applications on a remote machine. There are several limitations to the sharing of X applications, that have been possible to remove in VNC. First X was only available in UNIX and Linux systems (although there nowadays exist versions of XFree86 for both Windows and Mac OS) which made it very system dependent.

Secondly, an X server, which is a heavyweight process, had to be run on the client machine, which could not be expected by the thin client computers for which the system was developed.

The VNC system introduced a different approach to the problem of application sharing over the network. The VNC server is run as an X server performing all the graphics processing of the application, but instead of sending the calculated frame buffer data to the local frame buffer, the server sends it to a remote computer, the VNC client. Since the VNC server gets all the X commands, it will know exactly how the frame buffer has changed, and only needs to send the changes between every frame to the client. The VNC client only has to implement functionality to display the data sent by the server on the local frame buffer and thus need not run the system dependent, heavy X server locally. If the mouse and keyboard actions at the client are sent directly to the server and are used as input to the X server, this approach will give the client complete control over the remote computer.

At first sight, this approach would seem very network traffic intense, but since the server knows exactly how the data in the remote frame buffer looks, the data can be coded very efficiently, as long as the client can handle the different encodings. A standardized protocol for the communication between the server and client has been developed at AT&T, the Remote Frame Buffer (RFB) protocol described below, which defines the different encodings used.

VNC is currently a widespread, freeware, system for accessing a remote computer over the network, distributed as two programs, `vncserver` and `vncviewer`. In X systems, `vncserver` is a complete X server, run on the remote computer you like to control. `Vncviewer` is a smaller program which only takes care of displaying the remote frame buffer on the local computer and passing mouse and keyboard interactions to the server. There are currently servers and viewers available for several operating systems, including UNIX, Linux, Windows and MacOS and these programs are even installed as default on some new Linux distributions. Unfortunately, the Window server, `WinVnc`, is not nearly as fast and reliable as the X based one, `XVnc`, due to reasons discussed later.

VNC is also open source, which makes it interesting to use in this project. Since most of the complexity is in the server, it would be possible only to modify or rewrite the viewer and use the VNC server unmodified.

A screen shot of a normal VNC usage situation is shown in figure 1. Here a Linux VNC server is controlled from Windows.

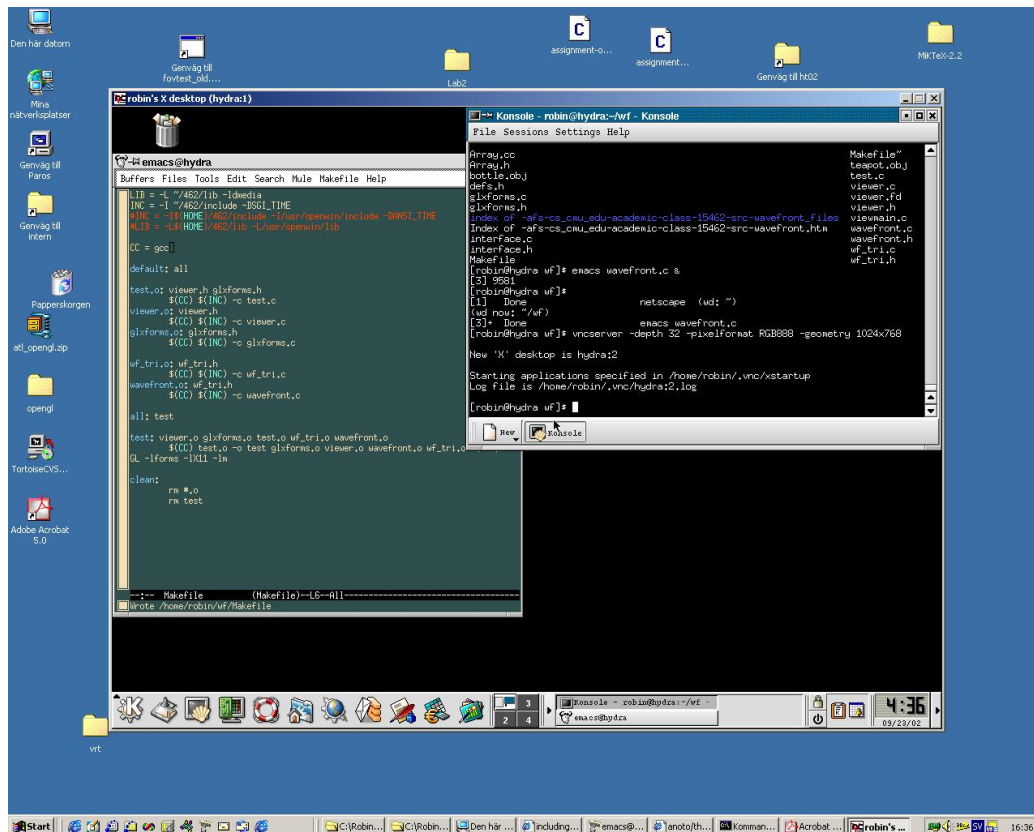


Figure 1: The classic VNC client, *vncviewer*, run under Windows connected to a VNC server running under Linux.

2.3 The Remote Frame Buffer Protocol

The Remote Frame Buffer (RFB) protocol [3] completely defines the communication between the VNC server and the client. This includes the data formats required by the different encodings used for the pixel data, but also the message formats for all messages sent between the VNC server and the client, for example mouse and keyboard interaction messages.

There are five different encoding schemes defined by the RFB protocol. These are Raw Encoding, Copy Rectangle encoding, RRE encoding, CoRRE encoding and Hextile encoding and they are described briefly below. The schemes all have different performance and are useful in different situations and they are described briefly below.

Raw encoding is the simplest scheme and does not compress the data at all. If the size of the sent rectangle is m times n pixels, the pixel data are simply sent as $m \times n$ pixel values line by line from left to right. Raw encoding is seldomly used, but has to be implemented by all clients and servers for compatibility, according to the protocol definition.

Copy Rectangle encoding is a simple, but sometimes very efficient encoding. It simply tells the client that a rectangle should be copied from one position to another in the local frame buffer. This can be very useful when for example moving windows and scrolling, but can also be useful for transmitting text and other repetitive patterns. A Copy Rectangle encoded rectangle simply consists of the coordinates, in the local frame buffer, from where the rectangle data is to be copied.

RRE encoding is a 2D version of run-length encoding. A RRE encoded rectangle consists of one pixel value for the background colour and a number of foreground rectangles, each consisting of a pixel value, its position and its size. RRE is not the most efficient encoding scheme, but it is very simple for the client to decompress.

CoRRE encoding is an extension of RRE where each rectangle is limited to 255×255 pixels in size. This reduces the memory used to represent the size and position of the rectangle to four bytes.

Hextile encoding is the most efficient encoding scheme for usual desktop data, but it is also the most complicated. It is quite similar to CoRRE, but the rectangles may have any size. Each rectangle is first divided into sub rectangles, or hextiles, that all are 16×16 pixels in size. Since the sizes of the rectangles are uniform and their size predefined, there is no need to send that information over the network, which will reduce the network traffic. Each tile begins with a header consisting of a single byte telling which sub encoding the tile is encoded in. The header is built up as a bit mask with five bits defining the encoding. If the first bit is set, the tile is **raw encoded** and the pixel data is simply sent as 256 pixel values. If the second bit, **background specified**, is set the following bytes will be the background colour used by this tile, but also every following tile until a new background colour is specified. The next bit, **foreground specified**, tells if the next piece of data consists of the foreground colour for possible sub rectangles within the tile. The fourth bit is called **any subrects** and if set the next byte will tell how many sub rectangles will follow. If it is not set, the whole tile will be filled with the background colour. The last bit, **subrects coloured**, determines if any sub rectangles in the tile will have different colours. If set, all sub rectangles will consist of one pixel value and two bytes telling the position and size of the

sub rectangle. If not set, each sub rectangle will simply consist of the position and size. After the one byte hextile header with the mentioned flags and possibly some extra data depending on the flags, data for the sub rectangles will follow. This data will in the simplest case consists of only two bytes each.

Apart from the encodings of the pixel data, the RFB protocol also defines the formats for all messages sent between the client and server. The messages the client is allowed to send to the server are: **SetPixelFormat**, **FixColourMapEntries**, **SetEncodings**, **FramebufferUpdateRequest**, **KeyEvent**, **PointerEvent** and **ClientCutText**. The client is required to handle the following messages from the server: **FramebufferUpdate**, **SetColourMapEntries**, **Bell** and **ServerCutText**.

For more details about the RFB protocol and the different encodings, see [3].

2.4 Performance of different VNC servers

As mentioned earlier the performance of the VNC servers differs significantly. A first explanation to this might be that the system was developed under UNIX and only was intended for use in a UNIX environment. Therefore, one might argue that more time and effort was spent on developing the UNIX version of the server and that it therefore should be more optimized than the Windows version.

A more correct explanation is probably that of the developers themselves, that it depends on the platform it is built on. Since the UNIX version is built on X, which is open source, the X VNC server could be built as a complete X server of its own. This means that all graphics commands sent from different applications to the operating system are sent directly to the X server (which is also the VNC server) and the server will then know exactly what changes have been made and when. It is then a simple task for the server to make an efficient encoding of the data that has to be sent to the VNC client. In the Windows operating system, on the other hand, there is no obvious way for a stand alone application to know what graphics commands the different applications send to the operating system, and there is therefore difficult to know how the appearance of the screen changes. It is also difficult to insert hooks in the system to probe for updates, since the code for Windows is not available. The result is that the only way for the Windows VNC server to know when the screen has changed is to poll the local frame buffer for changes. This does not only make the server slower, but also introduces a lot of room for errors, since the server cannot know what graphics commands generated

the changes and it is impossible to poll every single pixel in the frame buffer at a reasonable speed.

These differences between the servers is the reason to why there are very fast and efficient servers for X based systems (UNIX and Linux) but only a slow and erroneous server for Windows systems. There has however been work done on a Windows server that is camouflaged as a display adapter, that then will receive all graphics commands directly, but that server is currently not better than the original, polling version.

Another nice thing about the X VNC server is that since the X server is a virtual server, it does not need to be connected to a physical desktop at the host. It is therefore possible to run several X based VNC servers at the same physical computer, while each Windows based server must be run on a separate physical computer.

2.5 Using 2D applications in a 3D environment

As mentioned earlier, very little information is available on transforming existing 2D application for use in 3D environments. Two examples of 2D applications used in 3D environments have been found when searching the Web and are presented briefly here.

The first example, “Windows on the World” is presented in [4] and describes a way of mapping X windows onto the physical world. This system makes use of a see-through head-mounted display on which a complete X server is run, making it possible to display any X application as an overlay in the users normal field of view. Since the system was developed in 1993, and the work seems to have been discontinued, the visual appearance and performance seems to be quite limiting with a monochromatic display with limited resolution.

A second example focusing on a more 3D oriented desktop is Microsoft’s “TaskGallery” described in [5]. Here the classic 2D desktop in Windows is replaced with a 3D “office”, where the application windows are placed on the walls instead of on a flat 2D surface. This approach makes room for a new way of thinking and organisation of the work on the computer.

Neither of the mentioned systems gives the possibility of sharing applications between users on different computers, and both are tightly bound to a specific

system (X and Windows respectively).

2.6 VRT - Virtual Reality Toolkit

The Virtual Reality Toolkit [6] is a scene graph library, developed at Uppsala University. The development was started in 1997 and the toolkit is continuously updated. It is a toolkit for easy development of advanced 3D environments with support for many Virtual Reality (VR) input and output devices including different kinds of motion tracking devices and stereoscopic displays, but the toolkit is also suitable for developing desktop applications. VRT is built on top of OpenGL and support is added for high-level 3D constructions based on a scene graph. All native OpenGL functions can still be used and VRT takes advantage of OpenGL's hardware assisted graphics acceleration. The toolkit is written in C on Microsoft's WIN32 platform, which allows the use of functions from the Windows API together with the VRT API functions. It is not object oriented, but can easily be used in an object-oriented manner in C++, in the same way as OpenGL.

VRT provides the application programmer with many high level structures including nodes, geometric objects, textures, cameras, light sources and much more. The most important entities are the abstract VRT nodes building up the scene graph. Each node holds information about its parent and children, together with a transformation matrix, defining the node's orientation relative to its parent. Nodes may also have geometries, light sources, cameras or other physical objects bound to them and in this way build up the physical scene.

Currently the toolkit is used in several projects at Uppsala University and also in the education in a course in interactive graphical systems.

2.7 VASE - Visualisation And Simulation Environment

Visualisation And Simulation Environment [7] is a framework for developing networked virtual environments. It was developed as a thesis project at Uppsala University in 2002 and provides a method for building 3D environments in a dynamic way. The environment is built up of modular plug-in components and the configuration is scripted in the Extensible Mark-up Language (XML). It is also easy to develop new plug-in modules, which are compiled separately into Dynamic Linked Libraries (DLLs) and can be integrated into the main application

without the need of recompiling it.

VASE is developed in the VRT environment, described above, on Microsoft's WIN32 platform. Currently there exists a VASE implementation of a virtual classroom and a number of plug-ins for that purpose. The VASE plug-in developed in this thesis work is also used in this classroom environment. A screen shot of the classroom application can be seen in figure 10

3 Design issues

The choice of VNC as a basis for application sharing in 3D environments was natural, not only since it already provides most of the functionality for application sharing over the network, but also since it handles the data in a way that is easily integrated in computer graphics applications.

Most of the complexity with the VNC system lies in the server, and the possibility of using the server unmodified was very appealing. As hoped it turned out that the only code that had to be rewritten was on the client side.

Application sharing between clients is completely taken care of by the VNC server that has total control over the different clients. Each client does not have to know anything about any other possible clients, and can act as if it is the only client. The VNC server also has control of the current state at the clients, and knows what data the clients have been sent, and that they are supposed to have in their local frame buffers.

The fact that there exists VNC servers for a lot of different operating systems, and the server and clients can run different operating systems, makes it possible for the user to select which operating system to work with even if the implementation of the 3D client is bound to a certain system.

Some different approaches were investigated during the early stages of development, and are described here with arguments about why each approach was rejected or finally selected.

3.1 First approach: Redirecting an output stream from vncviewer to a shared memory, read by a separate process

The first idea for displaying a remote terminal in a 3D environment was to use the existing VNC client, vncviewer, almost without modifications and only redirect the decoded data stream to some format that could be read into the 3D environment. Since vncviewer outputs the changes to the screen directly as each small piece of data is received, this approach turned out to be difficult. from the server. The methods used for output are also very system dependent. The approach could however be implemented quite easy under Windows, by modifying the functions that output data to the screen, and instead let them send the output to a Windows BITMAP object. It was then possible to extract the pixel data from the bitmap object and write it to a shared memory. Another application could then read from the shared memory and use the data in a 3D environment.

There are however several drawbacks of this method. First, there is an obvious problem getting the relative mouse and keyboard actions, filtered by the 3D environment into vncviewer instead of the actual mouse and keyboard that by default goes directly to vncviewer. This would require a big deal of changes in the vncviewer code. Secondly, this approach requires at least two separate processes to run and communicate, which could cause priority problems, since both vncviewer and the 3D graphical application probably both assume they are the only heavy process running at a time. Third, and most important, this approach would be bound to the use of the VNC server that vncviewer implies, which assumes the only thing it does is receiving and displaying pixel data from the remote desktop as fast as possible. In a 3D application, however, there would typically also be many other things performed at the same time, and the reception of pixel data might not be the most important task. The approach vncviewer is using would then block the 3D application while a complete frame is updated, or alternatively display old data several times, while blocking the reception of a new frame. This would either give a very “jumpy” appearance of the 3D application or a virtual desktop that would flicker, or both. A last, important aspect of this approach is that it would make the program tightly Windows dependent, which does not comply with the fact that VNC was developed for UNIX and the UNIX/Linux VNC servers are superior the Windows ones.

A working test version of this approach was implemented, but was rejected since it had poor performance compared to the other approaches described later. It also lacks input capabilities.

3.2 Second approach: Using components from vncviewer

The second approach was to use working components directly from vncviewer and only rewrite the functions that deals with the displaying of data and the mouse and keyboard interaction. This would still have the drawback that data will be received in a way predefined by vncviewer and would block the 3D application while receiving new data. By this method, it is however possible to run the application as a single process that polls the server for updates.

Since the different parts of the vncviewer code are tightly bound together, this approach turned out to be very difficult to implement. A new frame of data is received a small piece at a time (depending on the encoding used) and vncviewer outputs every piece directly to the screen on a per pixel basis, before a new piece of data is received. In the most efficient encodings, each piece of data does not contain a complete piece of the screen, but can for example contain the background colour for a certain area. In a 3D application, it is desirable to receive the data on a per frame basis, or at least a complete piece of the screen, and map a whole frame onto a texture in the 3D environment.

To implement a working prototype using this approach, it turned out that almost the whole program had to be rewritten to be able to control the server in a way suitable for 3D applications. This approach was therefore rejected without implementing a prototype.

3.3 Third approach: Rewritten, system independent RFB library functions

The third approach, the one that was finally selected, was to write an own, generic library of functions for retrieving data from the server. By this approach, it was possible to have complete control over how the received data was handled and returned to the user.

In the first version of the library the data reception functions were more or less copied from the vncviewer code, but rewritten so the data was saved in a memory array instead of outputted on the screen. Once a whole frame was received, a pointer to the memory array was returned to the calling function. This would cause the calling function to block its execution in the same way as if the vncviewer functions had been used as in approach one or two above, but in a bit more effi-

cient way, since there is no unnecessary output to and copying from a Windows bitmap object. Avoiding the use of Windows objects also allowed the library to be platform independent, and it could now be tested under Linux as well. Simple mouse and keyboard input functions were implemented that allowed the user to filter the mouse movements and keyboard characters before they were sent to the VNC server, using the messages defined in the RFB protocol. Just retransmitting the mouse coordinates would of course give wrong mouse inputs if the virtual screen is rotated in the 3D environment, but this must be taken care of by the 3D application, since the virtual screen does not know anything about its position in the 3D environment. Also, just retransmitting the keyboard input from Windows, would send some keys incorrect since Windows uses a different key table than the VNC server (which uses X key codes), but it would at least work for ASCII characters and some control keys like the return key.

In a second version of the library, the problem that the data reception blocks the 3D application was addressed. This was done by letting the update function (that receives data from the server) return the control to the main application after receiving only a small part of the update message from the server. Next time the update function is called it continues the reception at the same place it ended and after several calls, when a whole frame is received this is signalled to the calling application, which then can make use of the data. This will slow down the reception a little, but in return, it will make the interaction with the 3D application a lot better. The performance of the data reception depends a lot on the 3D application, since that will determine how often the update function will be called. This problem was addressed by making it possible to select how much data to receive in every call to the update function, by setting the priority for the communication with the VNC server. In this version, the keyboard function was improved by using a key map table to map key codes from Windows or GLUT [8] to the X key codes used by RFB.

3.4 VRT VNC Window

To make the RFB library functions easier to use in 3D applications, wrapper functions for the Virtual Reality Toolkit (VRT) [6] have also been implemented. The basic object for these functions is `vRT_VNCWindow`, consisting of a rectangle with a RFB texture mapped onto it. There is also an update function that can be run in both blocking and non-blocking mode and functions for mouse and keyboard input. The mouse input function also handles the 3D aspect of the mouse movements, since the `vRT_VNCWindow`-object knows its position in space. The

function first calculates a ray from the point of observation, through the mouse pointer and onto the texture. The corresponding texture coordinates are then sent to the VNC server, instead of the direct mouse input. This makes mouse movements in the texture possible. The keyboard input function is extended with a function for sending strings to the VNC server.

The `vRT_VNCWindow` functions are compiled into a separate library that is linked into the VRT library. See the VRT VNC reference manual for details.

4 Implementation of the RFB Library

In order to achieve full control over the communication with the VNC server, all necessary basic communication functions have been rewritten. In this work only the most important functions from `vncviewer` has been implemented, while many less important functions have been left out. Those include for example support for the use of different colour depths since only the 32-bit pixel data format is used in the applications the library was intentionally written for. Another functionality left out is the support for some encodings (for example the RRE and CoRRE encodings) that can possibly be useful in some applications, but would be inefficient for the type of applications interesting here. These things could easily be implemented in a future version.

Some straightforward routines have been directly adapted from `vncviewer`. These include for example the encryption/decryption of the authentication key, used for the connection. The RFB protocol description, found in the file `rfbproto.h` has also been directly included. This file contains all structures and message type definitions used by the RFB protocol.

As new ideas came up the library has been developed in three different versions, of which the third version is the one finally used. All three versions are described here for comparison. They are also evaluated quantitatively in the results-section. This section starts with descriptions of what is common for the three versions and then the differences between the versions are described separately at the end.

For detailed information about how the functions are implemented, see the RFB Library Reference Manual in Appendix [A](#).

4.1 Types and Endians

The RFB protocol definition file, `rfbproto.h`, uses standard unsigned data types for most the variables sent over the network. These types have been redefined as the following three types, and they are used by the RFB library functions as well.

Integer types used by the RFB protocol:

```
typedef unsigned long CARD32;
typedef unsigned short CARD16;
typedef unsigned char CARD8;
```

Since the VNC system is developed to run on multiple platforms, different endians (order of the bytes within a word) are allowed for representing integer types taking up more than one byte. Usually the internal representation of the types is of little interest, but when sending data between computers with different endians, these details have to be taken into account. The x86 architecture on most PCs use little endian (the least significant byte is stored at the lowest memory address) Suns SPARC processors used in many UNIX systems and Motorola's processors used in many Mac computers, on the other hand, use big endian (the most significant byte is stored at the lowest memory address). By definition, the RFB protocol always uses big endian for storing data types consisting of more than one byte (for example `CARD16` and `CARD32`). Therefore, when used on a little endian machine, the data must be endian-swapped after reception. This is done with the following macros, where `LITTLE_ENDIAN` is a flag that is true if the local machine used little endian.

Macros for swapping endians of integer types:

```
#define Swap16(s) (((s) & 0xff) << 8) | (((s) >> 8) & 0xff)
#define Swap32(l) (((l) >> 24) | \
    ((l) & 0x00ff0000) >> 8) | \
    ((l) & 0x0000ff00) << 8) | \
    (l) << 24)
#define Swap16IfLE(s) (LITTLE_ENDIAN ? Swap16(s) : (s))
#define Swap32IfLE(l) (LITTLE_ENDIAN ? Swap32(l) : (l))
```

4.2 Initial handshaking and Authentication

Before the RFB client can start receiving pixel data from the server and send mouse and keyboard actions to it, a client - server connection has to be properly initialized and the client and server must agree on the format for the data. All this is taken care of by the user function:

```
RFB_state *RFB_Init(char *server, int port, char *password)
```

It takes as parameters a server name and a port number that are used for the connection setup and a password for the authentication to the server. The function will create an object of type `RFB_state` that will contain all necessary variables defining the connection, once it has been established, and information about the status with the remote desktop. `RFB_Init` will run the functions `RFB_SetupConnection`, `RFB_InitMessages` and `RFB_SetEncodings` described below, in order and then return a pointer to the `RFB_state` variable created earlier.

`RFB_SetupConnection` creates a usual TCP connection to the server using the standard network functions `gethostbyname`, `socket` and `connect` [9]. The socket descriptor is saved in the `RFB_state` variable `*s` that is taken as parameter.

`RFB_InitMessages` takes care of the initial handshaking with the server once a connection exists. It also takes care of the client authentication at the server.

First, the server protocol version is sent from the server. This message has to be received and answered by the protocol version the client can handle, and that will actually be used. (The server is supposed to be backward compatible with older protocol versions, so as long as the client's version is older than the server's is, the client will decide which version to use.) Since version 3.3 is the latest version the library is currently developed for, that will be returned.

Next follows the client's authentication to the server. VNC authentication uses DES encryption with a 16-byte challenge [10]. The challenge is sent from the server and encrypted by the client using `password` as key. The resulting, encrypted key is returned to the server for verification. If the key is accepted by the server the communication is allowed to continue, otherwise the client is disconnected from the server. The code for the encryption scheme is taken directly from `vncviewer` by linking the files `d3des.h`, `d3des.c`, `vncauth.h` and `vncauth.c` to the `rfb` project.

After the authentication, the handshaking is continued by the client sending an initialization message telling the server whether the server shall be dedicated to this client alone, or if it shall be shared between all clients connected to it. In the `rfb` library the default is to always allow the server to be shared.

Finally, the server sends an initialization message containing information about the remote desktop. This information contains the server name, the size of the remote desktop and the pixel format. The pixel format contains information about the colour depth, the endian of the pixel data and some other information used for non true-colour pixel formats that is ignored in the RFB library, since only true colour is currently implemented.

After the handshaking is completed, the client is free to start sending messages to the server. The first message sent is a message telling the server which encodings of the pixel data the client can handle. This is done by the function `RFB_SetEncodings`. It is then up to the server to send the pixel data using a combination of the allowed encodings. Currently raw encoding, copy rectangle encoding and hextile encoding are supported by the `rfb` library.

4.3 The Update functions

The VNC server is totally client controlled, which means it will not send any updated pixel data unless it is specifically told to do so by the client. This means that the client must request new pixel data every time it wants to update the screen. The server expects a `FramebufferUpdateRequest` message from the client telling it for which part of the screen an update is requested. The server then sends a `FramebufferUpdate` message back to the client, consisting of a header for the entire message telling how many sub rectangles the message consists of, then the sub rectangles are sent one by one, each starting with an own header telling the size and encoding of that particular rectangle.

The way the original VNC client, `vncviewer`, handles the communication with the VNC server is by running an endless loop (the simulation loop in Windows). Every time it gets the control, it sends a `FramebufferUpdateRequest` message and receives a `FramebufferUpdate` message, if one is sent. `Vncviewer` updates a small piece of the screen after every piece of data is received. After the whole update message has been received and if no mouse or keyboard actions has occurred, it sends a new update request message and so on. This approach works fine in `vncviewer` since all it does is sending mouse and keyboard input messages, send-

ing update requests, receiving updates and updating the screen. However, in a 3D application, there might also be a lot of other time consuming processing to take care of, and the communication with the VNC server might not be the most important task in this application. Therefore, a somewhat different approach had to be taken.

In order to use the data from a VNC server in a 3D application, one wishes to receive a whole frame at a time and replace the current texture data in the 3D environment. Since the data is sent in small pieces, there is no way to know in advance how much data an entire `FramebufferUpdate` message consists of and the time for receiving a whole update message may vary a lot. These variations may affect the interactivity of the application and cause it to hang in small intervals, which can be very annoying for the user. In the first version of the RFB library this is ignored, but in the second version, a non-blocking update function has been implemented. Common for both versions is the update function that has to be put in the simulation loop and that will take care of sending an update request and receiving data from the server. The data is stored in a memory buffer instead of drawn directly to the screen. In the first version, the update function blocks until the whole update message has been received and then returns a pointer to the memory buffer with the new pixel data. In the second version, the update function will return after a piece of the update message has been received (how much data this piece contains is selectable by the user) and when the whole message has been received, a pointer to the memory buffer can be received with `RFB_GetTexels`. If the update function returns before the whole update message has been received it will return 0 and when the whole update message has been received and decoded, 1 will be returned.

The simplest way to update the screen is to copy the whole contents of the memory buffer where the update function has saved the remote data into the texture memory (with for example `glTexImage2D` if OpenGL is used). This will however cause a big amount of unnecessary data to be copied to the texture memory, since most of the texture probably is unchanged. It is therefore possible to copy only the data for the rectangles that have changed, and use some partial texture update function (like for example `glTexSubImage2D`) instead. The locations of the updated rectangles can be retrieved with the function `RFB_GetUpdatedRects`. This has experimentally turned out to more than double the update rate for usual desktop applications. This is discussed more in the Results section.

If you know in advance that only parts of the remote frame buffer has changed, or if you are only interested in a specific part of it, for example a specific window, it can be useful to receive only that part. The function `RFB_Update` will request an

update of a specified rectangle, while `RFB_updateFullScreen` will request an update for the entire remote frame buffer.

Syntax for the update functions:

```
int RFB_Update (RFB_state *s, int blocking, CARD16 x,
               CARD16 y, CARD16 w, CARD16 h);
int RFB_UpdateFullScreen (RFB_state *s, int blocking);
CARD8 *RFB_GetTexels (RFB_state *s);
```

4.4 Mouse and keyboard interaction

In the original VNC client, `vncviewer`, whenever a mouse action is performed, a message is sent to the VNC server. This message contains information about what action was received (mouse moved/dragged or button(s) pressed/released) together with the current coordinates of the mouse pointer within the `vncviewer` application window. The VNC server sends these actions directly to the right application run on the server. The changes at the server, caused by the mouse action will be sent to the client when it asks for a new screen update later.

If the VNC window is part of a 3D application, things are a bit different. Since the VNC window is not necessarily located in parallel with the plane of the physical screen (or other output device), the mouse coordinates received from the operating system on a mouse action will not translate directly to coordinates in the VNC window. It is not even necessary that the mouse action should be sent to the VNC server at all. This would be the case when the mouse pointer hits the application window, but not the VNC window-object inside that window. To calculate the right coordinates in the VNC window, a ray in the 3D space must be used. This ray originates from the point of the observer and goes through the mouse pointer, in the plane of the screen. If this ray, when followed into the 3D scene, hits the VNC window, then this hit point is transformed into coordinates in the VNC window and these coordinates are sent to the VNC server.

The position of the VNC window and the observer in the 3D scene, is dependent of the implementation of the 3D scene. Since the RFB library was supposed to be general, and independent of the 3D implementation, there was no way of implementing this functionality at this level. The transformation of physical screen coordinates to coordinates in the VNC window, therefore has to be done on the application level. To avoid this, and some other problems, an implementation specific library for VRT was developed and is described later. On the RFB library

level is implemented a function, `RFB_HandleMouseEvent`, that takes care of sending a mouse event to the VNC server, once the coordinates in the VNC window is known. In addition to the mouse coordinates, it takes as parameter a bit mask that defines the current states of the buttons.

Syntax for the mouse input function:

```
int RFB_HandleMouseEvent (RFB_state *s, CARD16 x, CARD16 y,
                          CARD8 mask);
```

Syntax for the keyboard input function:

```
int RFB_HandleKeyEvent (RFB_state *s, CARD32 key, CARD8 down);
```

4.5 Closing the connection

The connection with the VNC server is closed by a call to `RFB_Quit`. This function will leave the VNC server in the current state until it will be connected to again. `RFB_Quit` will also free all memory allocated by the `RFB_state` variable.

Syntax for the closing function:

```
void RFB_Quit (RFB_state *s)
```

4.6 Version 1 - blocking

In the first blocking version of RFB library, the reception of pixel data from the remote server is handled by a blocking read function that blocks the program execution until a whole frame has been received. The pixel data from the server is always sent rectangle by rectangle so there is no way to know in advance, how much data will be sent for each frame. Therefore, the data has to be received one rectangle at a time, starting with a rectangle header for each rectangle, holding information about how much data that specific rectangle consists of. When the size of a rectangle's data is known the blocking read function is called, and will not return until the whole rectangle has been received. The blocking read function `ReadExact` takes an instance of `RFB_state` as parameter, from where it

extracts the socket descriptor to receive data on. It also takes a pointer to a buffer to put the received data in and finally the number of bytes to receive before return. `ReadExact` runs the standard `recv` function from inside a loop until exactly the requested number of bytes has been received.

How the read function is called and how many times per rectangle depends on the encoding of the rectangle currently being received. For a **raw encoded** rectangle `ReadExact` is only called once receiving $width \times height \times bytes_per_pixels$ bytes of data that is copied into the virtual frame buffer.

For a **copy rectangle encoded** rectangle `ReadExact` is also only called once receiving the coordinates to copy the rectangle data from in the virtual frame buffer. If the source and destination memory overlap, the copying has to be done in the right order, in order not to overwrite data not yet copied.

The case with **hextile encoding** is a bit more difficult. Since the rectangle is split up into 16×16 pixel wide squares or hex tiles, the number of sub rectangles can be calculated from the size of the rectangle. Each sub rectangle's one-byte header is read and evaluated. Depending of the encoding, the right amount of data is read by the `ReadExact` function that can have to be called several times for each sub rectangle. In the blocking update function, all this can be done from within a loop that covers all sub rectangles. This is also the way it is done by `vncviewer`.

After all sub rectangles have been read and the virtual frame buffer has been updated, a pointer to the data is returned to the 3D application. The application programmer is then free to take care of the data in any suitable way, the most obvious being to map the data onto a texture in the application scene.

The first version of the `RFB_state` struct had the following fields:

```
typedef struct {
    int serverSocket;           /* Socket for network communication. */
    CARD16 port;               /* Port number offset. */
    CARD16 RFB_width;         /* Width of remote screen. */
    CARD16 RFB_height;        /* Height of remote screen. */
    CARD8 RFB_bigEndian;      /* Endian of colour data from server. */
    int first;                 /* Flag indicating this is the first frame. */
    /* Unformatted texture data. */
    CARD32 netbuf[TEXTURE_SIZE * TEXTURE_SIZE];
    /* Formatted texture data. */
    CARD32 texbuf[TEXTURE_SIZE * TEXTURE_SIZE];
} RFB_state;
```

4.7 Version 2 - non blocking

One result when evaluating the first version of the library functions in a simple 3D application, was that the update frequency of the VNC image, when texture mapped onto a surface in the scene, was noticeably slower than that of vncviewer. Another result was that the interaction with the 3D application was quite slow and unpredictable.

To address the problem with slow and unpredictable interaction a second, non-blocking, version of the update function was implemented. This function is forced to return the control to the calling application, after a certain time (or in reality when a certain number of bytes have been received), even if the whole update message has not been received. This gives the application programmer more control over the behaviour of the program. If the update frequency of the VNC window and the interaction with the VNC server is most important, one can let the update function read more data each time it is called. If instead the interaction between the user and the 3D application or the rendering speed of the rest of the 3D application is most important, one can force the update function to return the control to the application more often. This way the texture cannot be updated until the whole update message has been received, by sufficiently many calls to the update function.

To make it possible to resume an interrupted reading of an update message, many state variables have to be saved between the calls to the update function. All those state variables have been added to the `RFB_state` structure that now contains the following:

```
typedef struct {
    /** Common state variables */
    int serverSocket;
    CARD16 port;
    CARD16 RFB_width;
    CARD16 RFB_height;
    CARD8 RFB_bigEndian;
    int first;
    CARD32 netbuf[TEXTURE_SIZE * TEXTURE_SIZE];
    CARD32 texbuf[TEXTURE_SIZE * TEXTURE_SIZE];
}
```

```

    /** Special state variables for non blocking read */
    int non_blocking_priority;
    int request_sent;
    int reading;
    int reading_subrects;
    int current_subrect;
    int bytes_to_read;
    int read_bytes;
    char *buffer;
    CARD8 msgType;
    rfbFramebufferUpdateMsg Update;
    rfbFramebufferUpdateRectHeader RectHeader;

    /** Special state variables for non blocking hextile. Most of them are only
        interesting locally, but have to be preserved between calls. */
    CARD8 hex_flags;
    rfbRectangle hex_r;
    CARD8 hex_subencoding;
    CARD8 hex_nSubrects;
    CARD32 hex_bg;
    CARD32 hex_fg;
    CARD8 *hex_ptr;
} RFB_state;

```

For a more detailed description of the different fields in the struct, see the RFB reference manual in appendix [A](#).

Instead of using the `ReadExact` function described earlier, a number of new functions and macros had to be implemented. A non blocking read is initiated by a call to the function `BeginRead` that takes the same parameters as `ReadExact`, but instead of reading the specified number of bytes, it only reads the number of bytes specified by the priority field in the `RFB_state`-struct. The next time the update function is called, the function `ContinueRead` is called instead, that will resume the read where it was interrupted. To make this behaviour more transparent in the code, the `ContinueRead` function is run from within a macro `READ_AND_RETURN` that automatically makes the whole update function return if the whole message is not yet read. These special functions and macros are described in detail in the RFB Library reference manual in appendix [A](#).

Syntax of the special read functions and macros:

```
void BeginRead(RFB_state *s, char *buf, int len);
int ContinueRead(RFB_state *s);

#define READ_AND_RETURN(s) \
{ \
    if ((s)->reading) { \
        if (ContinueRead((s)) == -1) \
            return -1; \
        if ((s)->reading) \
            return 0; \
    } \
}
```

For the hextile-encoded rectangles, the non-blocking read is a bit more complicated, since there are so many state variables that have to be preserved between the calls. Many new fields had to be introduced in the `RFB_state`-struct and a new macro, `CONTINUE_HEX`, had to be implemented as well. All this is described in the reference manual.

4.8 Version 3 - subtexturing

As mentioned before, the simple approach of replacing the whole texture's raw data every time a new update message has been received from the server is inefficient. Most of the time, only a small part of the remote frame buffer has changed, and thus only a small amount of data is received. The received and decoded data is always inserted into the right memory location in the local, virtual frame buffer. That is the buffer pointed to by `s->texbuf` where `s` is the `RFB_state` pointer returned by `RFB_Init`. Always saving the decoded data in the same location makes it possible to extract only selected parts of it for replacement. See figure 2 for an overview of how the data is saved. This figure tries to explain the way a small change at the server propagates to the client. If the person in the right figure moves his hand, the pixel data within the small rectangle will be encoded by the server. It is up to the server to select the encoding, as long as that encoding is supported by the client. The encoded data is sent to the server and saved in `s->netbuf` (where `s` is the current `RFB_state`). It is then decoded and saved in `s->texbuf`, which will now contain the whole remote frame buffer, not only the changed parts. Information about the updated rectangle will be saved at the

same time, allowing extraction of that particular data later. This makes it possible for the 3D application to select whether to replace the whole texture or only the changed parts.

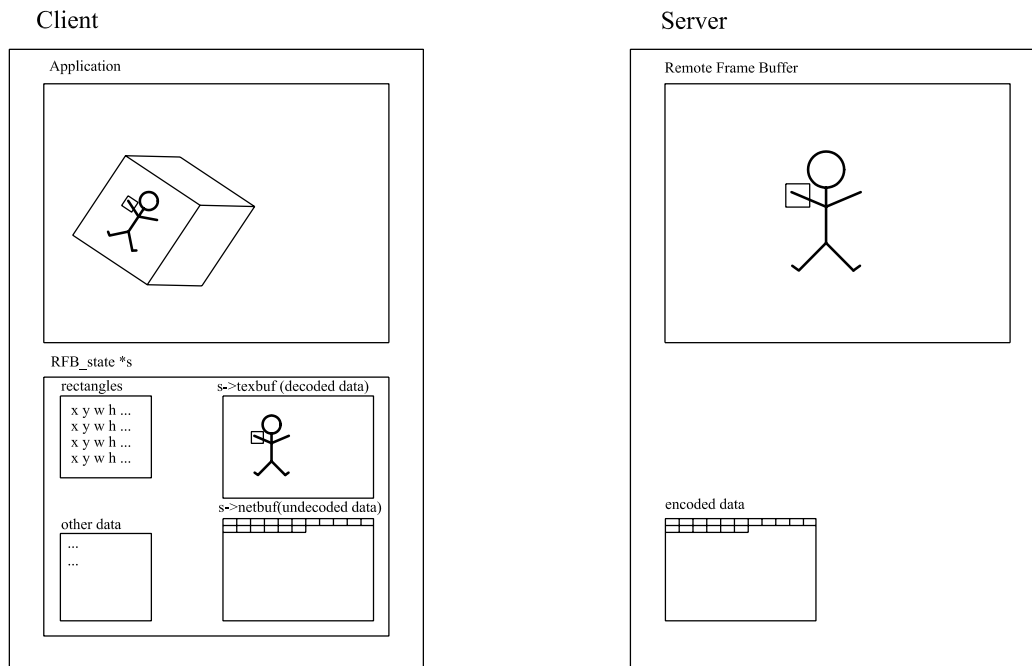
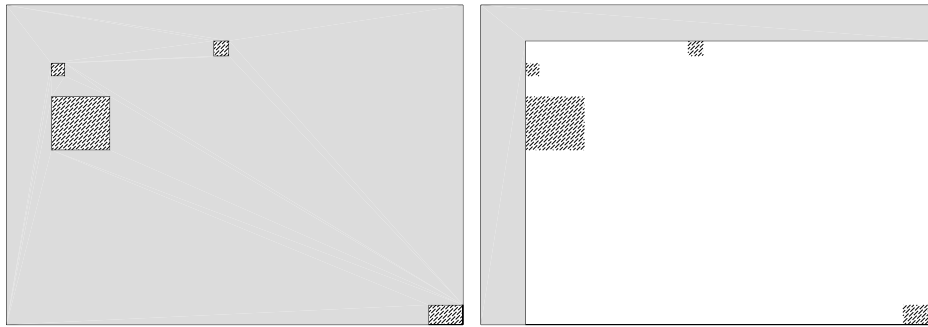


Figure 2: *The data storage at the client and server*

Fortunately, OpenGL (and probably most other graphical APIs) support replacing only parts of the texture. What had to be done in the RFB library to support this was adding a way of telling the application what parts of the screen has changed and need to be updated in the texture memory. Since the position and size of each rectangle received from the VNC server is known at decode time, all that had to be done was to save this information in the `RFB_state`-struct and provide functions for extracting this information later. The rectangle information is needed again when the whole update message has been received. This can be after only one call of the update function if blocking mode is used, or several hundred calls if non-blocking mode is used with low priority.

Depending on which encodings the server uses, the number of rectangles each update message consists of varies from one to several hundreds. In addition, the location of the changed rectangles in the screen varies a lot. In order to determine if partial replacement of the texture data is efficient some experiments were performed. These experiments also helped to determine if each rectangle should be

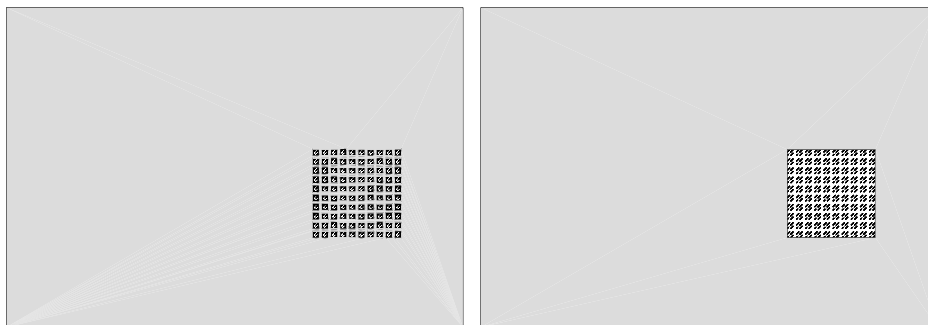
replaced with an individual call to the replacement function (for example `glTexSubImage2D` in OpenGL) or if only one call should be made, replacing all pixels within a bounding rectangle. See figure 3 and figure 4 for two possible situations.



(a) Each rectangle is replaced individually.

(b) All rectangles are replaced at the same time.

Figure 3: A situation where the use of a bounding box would be unefficient.



(a) Each rectangle is replaced individually.

(b) All rectangles are replaced at the same time.

Figure 4: A situation where the use of a bounding box would be preferable.

The situation in figure 3 is a common situation and can for example represent mouse movements in the upper left section of the screen while a clock is simultaneously updated in the lower right corner. Here it is natural to assume that it would more efficient to replace each rectangle individually, since the bounding rectangle would contain a big amount of data that is already up to date.

In the situation in figure 4 on the other hand, it would probably be more efficient to replace all data within the bounding rectangle at once. This can for example

represent text editing, when a line is filled and all characters are moved up one line. The VNC server usually represents a line break in text this way, by sending each character as a CopyRect encoded rectangle. (This would only require 16 bytes per character, independent of the font and character size, which is a very efficient use by comparison.)

The experimental setup and the results can be found in the Results section at the end. These results show that the update rate, when replacing sub rectangles, is more than doubled in some situations and never worse than the update rate when always replacing the whole texture at once. The results also show that even when the update message consists of several hundred rectangles, like in figure 4, the update rate is not worse for replacing each rectangle individually, than replacing the whole texture at once. More unexpected is that it is actually faster to replace each sub rectangle individually than to replace the data within the bounding rectangle at once. This is probably due to the limitations of the experiments. In the text editing situations tested, the updates with hundreds of rectangles only appeared once for each line of text. Updates with only a few rectangles, on the other hand, appeared once for each character. Since only average times were measured, the updates with many rectangles had little influence on the results. Since this is also the normal situation, the updates with many rectangles will have little influence on the behaviour of the applications.

Concerning these results the new user function `RFB_GetUpdatedRects` was implemented. It allows the application programmer to extract only the updated parts of the texture from the local, virtual frame buffer. Since this is a bit more complicated to use, it is still possible to replace the whole buffer at once. A limit is set to 200 sub rectangles, and for update messages with less than 200 rectangles, each rectangle can be returned by `RFB_GetUpdatedRects`. For update messages with more than 200 rectangles the bounding rectangle is calculated and returned instead. This will give the application the illusion that the whole bounding rectangle has changed and need to be replaced.

Syntax for receiving the updated rectangles:

```
int RFB_GetUpdatedRects (RFB_state *s, rfbRectangle** rects);
```

5 Implementation of VRT VNC Window

To increase the usability of the RFB library functions, another library was developed for the Virtual Reality Toolkit described earlier. Some of the functions in this library are only wrappers around the functions in the RFB library, but some of them add additional functionality. The basic structure in the VRT VNC library is the `vRT_VNCWindow` struct. It holds all information necessary for the communication with the VNC server, together with some additional information about the VRT geometry onto which the VNC data is mapped.

Members of the `vRT_VNCWindow` struct:

```
typedef struct {
    RFB_state *state;
    VRT_Geometry *g_win;
    VRT_Htx t_win;
    VRT_HPlg p_win;
    unsigned char texels[VNC_TEXTURE_SIZE * VNC_TEXTURE_SIZE * 4];
} vRT_VNCWindow;
```

In addition to an instance of `RFB_state`, the `vRT_VNCWindow` struct holds pointers to a VRT geometry, a VRT polygon and a VRT texture. The pixel data from the VNC server is used as texel data in the VRT texture, which is mapped onto the VRT polygon. In turn, the VRT geometry contains the polygon and can be connected to an arbitrary VRT node in the 3D application.

The `vRT_VNCWindow` is controlled by the following API functions:

```
vRT_VNCWindow *vRT_VNCWindowNew(char *serverName, int portNo,
                                char *passwd);
int vRT_VNCWindowClose(vRT_VNCWindow *win);
int vRT_VNCSetPriority(vRT_VNCWindow *win, int priority);
int vRT_VNCWindowUpdate(vRT_VNCWindow *win, int blocking);
int vRT_VNCWindowUpdatePartial(vRT_VNCWindow *win, int blocking,
                               int x, int y, int w, int h);
int vRT_VNCHandleMouseEvent(VRT_Node *node, vRT_VNCWindow *win,
                            int x, int y, unsigned int winmask);
int vRT_VNCHandleKeyEvent(vRT_VNCWindow *win, int virtkey,
                          DWORD keyData);
int vRT_VNCSendString(vRT_VNCWindow *win, char *str);
int vRT_VNCSendStringR(vRT_VNCWindow *win, char *str);
int vRT_VNCGetLastError(char *error_message, int size);
```

`vRT_VNCWindowNew` creates a new instance of `vRT_VNCWindow` and sets up a connection with the specified VNC server. `vRT_VNCWindowClose` closes the connection and frees the memory allocated by the `vRT_VNCWindow`. As with the RFB library there are two update functions, `VNCWindowUpdate` and `vRT_VNCWindowUpdatePartial`, that updates the full remote screen and a specified part of it respectively. Both of them can be run in both blocking and non-blocking mode by specifying the `blocking` flag. If run in non-blocking mode, the amount of data read in every call is determined by the value set with `vRT_VNCSetPriority`. Mouse input will be filtered and sent to the VNC server with the function `vRT_VNCHandleMouseEvent`. This function should be given the mouse input directly from the Windows simulation loop. Only if the mouse cursor falls within the VNC window, the mouse position will be translated from local screen coordinates to VNC screen coordinates and sent to the server. Keyboard input messages from Windows should be inputted directly to the function `vRT_VNCHandleKeyEvent`, that will translate the key codes from Windows codes to the corresponding X codes used by VNC, and send them to the server. To send complete strings from the application it is easier to use `vRT_VNCSendString` or `vRT_VNCSendStringR` for unterminated and line feed terminated strings respectively. If an error occurs, an error message will be printed by VRT. The corresponding error code can be retrieved to the application with the function `vRT_VNCGetLastError`.

For complete descriptions of the `vRT_VNC` API functions, see the VRT VNC reference manual in appendix B. There is also an example of a basic VRT application with a VNC window in appendix C.

6 Implementation of the VASE Plug-in

One of the goals with the 3D version of VNC was to implement a plug-in application for the Visualisation And Simulation Environment (VASE) described briefly above. This application also served as a way of evaluating the 3D VNC client in a real usage situation, as well as way of evaluating the VASE plug-in system.

The VNC plug-in for VASE, `vpivnc`, was developed in two different versions of which the first used the RFB library functions directly. As the development of the VRT VNC API functions proceeded, a second version of the plug-in was developed, using these API function instead. In this work, most of the code from the first version became unnecessary, since the VRT VNC API had most of the

functionality needed. Some additional functionality was however needed for the plug-in application, which is not supported by the VRT VNC API.

One thing that is special in the VASE plug-in system, is that it is very dynamic, and reconfigurable at runtime. This means the VNC plug-in must be able to be deallocated and reallocated at runtime. It must be easy to log in to and out from different VNC servers at the same time and without leaking any system memory. Some fine-tuning of the library functions was necessary to achieve this, but all this might be useful for other applications as well.

In addition, the connection to the VNC server is to be shared between different users in the same VASE environment. In VASE many variables, like the positions of the users avatars for example, are shared over the network between all users in the same VASE environment. This means that ideally, the `vRT_VNCWindow` object should be a shared variable, but since it contains a big amount of data, including a large buffer of pixel data, this would not be feasible. Instead, only the server name, port number and password to the VNC server are shared, and each instance of the plug-in will make a separate connection to the VNC server. This way the VNC server will take care of the distribution of data between the plug-in instances. By this approach there was a need for the different instances of the plug-in to log in to and out from a VNC server simultaneously. Since VASE provides an excellent system for passing messages between plug-ins, both between different instances of the same plug-in and between plug-ins of different types, this was not difficult to implement.

In the VASE plug-in there are two ways to specify how to connect to a VNC server. The first is to specify server name, port number and password in the VASE configuration file for that environment. Each instance of the plug-in will then have this information and will individually log in to the VNC server. A VNC server specified in the script file will be logged in to directly when the application is started, and the VNC window will be in the environment from start.

The second, more dynamic, way of connecting to a VNC server, would be to let the user explicitly log in to the server from within the 3D application. To achieve this, a separate plug-in, `vpiTextArea`, was developed. This plug-in consists of a text area in the 3D environment, that is capable of taking input from the user, display it and send it to a specified plug-in. Its size, number of text lines etc, is specified in the VASE configuration file. In the script file, it is also possible to specify which plug-in to send the text to and which plug-in to receive text from that is displayed in the text area. The primary use for the text area plug-in is as a log in prompt for the VNC window. This way any user in this VASE environment

can type in information about the VNC server to log in to. The local instance of the VNC plug-in connected to this text area will then log in to the specified VNC server. At the same time, the text will be shared with all other instances of the same environment. The VNC plug-ins connected to the corresponding text areas in the other instances of the same environment will then log in to the same VNC server. This way all instances of the same environments will be logged in to the same VNC server. What one user inputs to the VNC window will be visible in the corresponding VNC windows at all other instances of the VASE environment. Logging out is done in the same way, by use of the text area, and then it is possible to log in to a different VNC server.

7 Results and Conclusions

7.1 Blocking vs. non-blocking read of update messages

To investigate if it is reasonable to use the non-blocking versions of the update functions described in section 4.7, the following tests were made. Here a comparison is made between blocking read, which means the application is blocked until the whole frame is received, and non-blocking read, where the control is returned to the application before the whole frame is received. In the case of non-blocking read, three different tests have been performed, allowing one, 10 and 100 rectangles to be read in each call to the update function. Four different usage situations have been tested and they are mouse movements over the whole screen, typing of nonsense text, scrolling a text window up and down and video streaming. The results are showed in figure 5. Here the frame rates are measured for the four different usage situations and the four different configurations (blocking and non-blocking with one, 10 and 100 rectangles per frame respectively). The bars represent the intervals between the 5% percentile and the 95% percentile, which means 90% of the measured values will lie inside these intervals. The vertical lines represent the minimum and maximum measured values respectively. Since the first few samples differ a lot from the rest, they are removed from the results.

The results show that there is no significant difference in the frame rates for the measured situations. It is therefore advisable to use non-blocking read as much as possible. The functions for blocking read of the update messages are kept in the library, allowing the application programmer to select which method to use, since these functions are somewhat easier to use.

7.2 Replacing the whole frame buffer vs. replacing individual rectangles 35

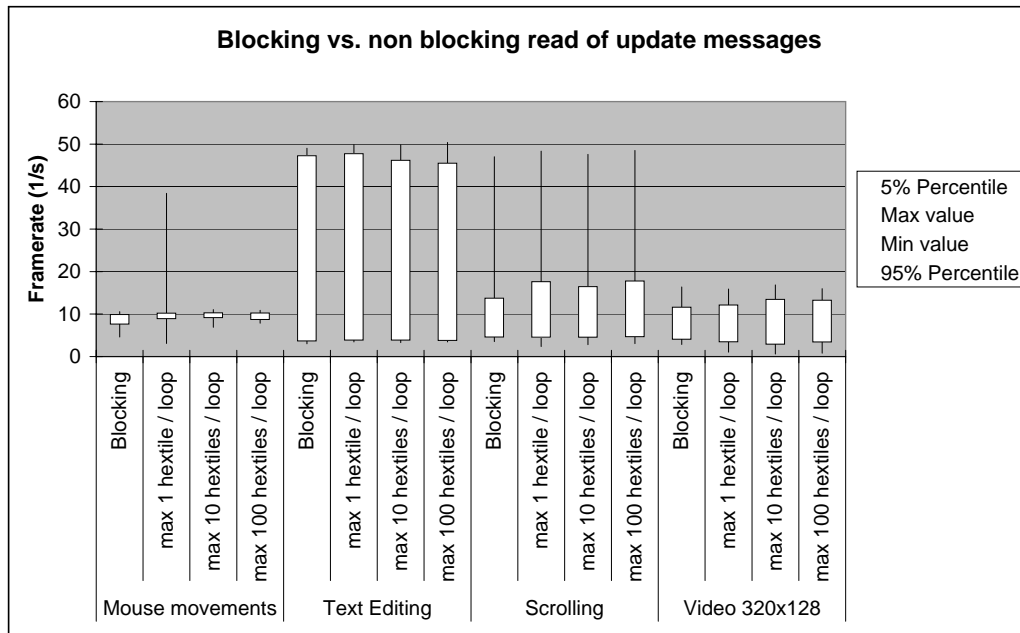


Figure 5: Comparison between blocking and non-blocking read of update messages.

Interactivity is difficult to measure. The general “feeling” is however that in programs using the non blocking functions, the interactivity with the 3D environment is a lot better compared to programs using blocking update.

7.2 Replacing the whole frame buffer vs. replacing individual rectangles

This section describes the tests performed to select which method to use for replacing the frame buffer data in the texture. Either the whole texture of frame buffer data is replaced after each update message has been received from the server, or only the data in the changed rectangles is replaced. A third approach would be to replace all data within the rectangle enclosing all the changed rectangles. For a description of the different scenarios, see section 4.8.

Three different tests have been performed, each of them in four different usage situations. All tests have been performed using blocking read of the update messages (see section 4.6). The RFB library functions have been used in the OpenGL

367.2 Replacing the whole frame buffer vs. replacing individual rectangles

Utility Toolkit (GLUT) [8] environment. (The reason for not using the VRT environment was that VRT did not have support for partial replacement of textures at the time of the tests.) `glTexImage2D` and `glTexSubImage2D` are the two functions used in the comparison. With `glTexImage2D` the whole texture is replaced at once, while with `glTexSubImage2D` it is possible to specify what part of the texture to replace.

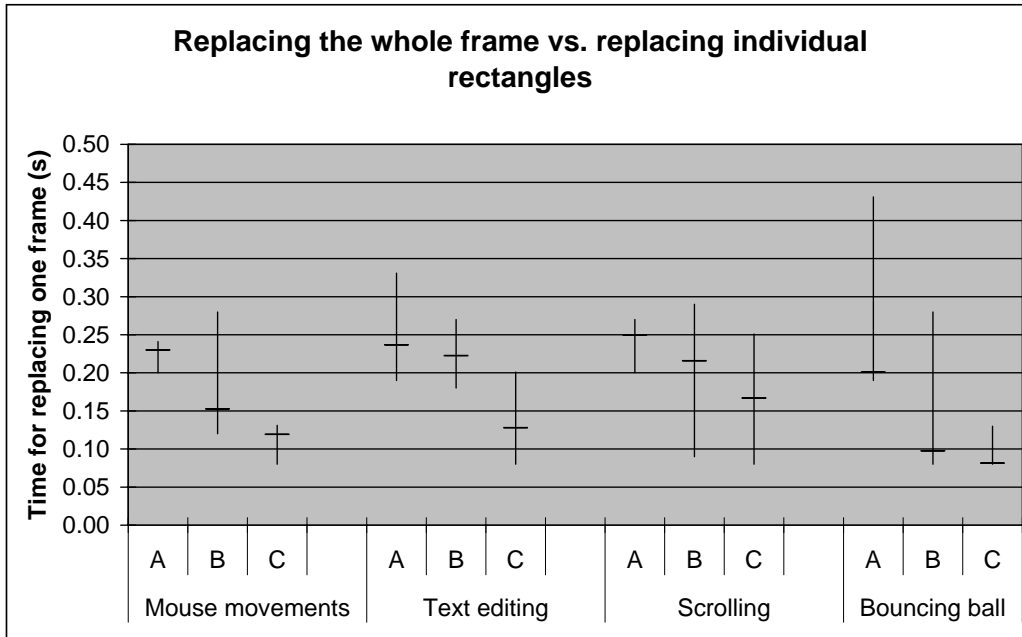


Figure 6: Comparison between `glTexImage2D` and `glTexSubImage2D`. In the situations marked A the whole frame is replaced with one call to `glTexImage2D`. In the situations marked B the bounding rectangle around the changed part of the frame is replaced with one call to `glTexSubImage2D`. Finally in the situations marked C each rectangle is replaced with a separate call to `glTexSubImage2D`.

The results of the tests are showed in figure 6. The horizontal black bars represent the mean time for a frame update, while the vertical lines represent the minimum and maximum times. In the tests marked A in the figure the whole texture was replaced with `glTexImage2D` after each update message had been received. In the tests marked B, all pixels within an enclosing rectangle around the changed rectangles were replaced with a single call to `glTexSubImage2D` for each update message. Finally, in the tests marked C, each rectangle was replaced with a separate call to `glTexSubImage2D`. The tested situations were mouse movements over the whole screen, typing nonsense text at high speed, scrolling a web

browser window up and down and finally running the simple bouncing ball game jezzball. Each test was done with between 300 and 400 updated frames, where the first few frames were removed, since they took significantly longer time.

It is obvious from the results in the diagram, that the use of partial replacement (B and C) is faster than replacing the whole texture at once (A). The replacement of each rectangle separately (C) is also faster than replacing the whole bounding rectangle at once (B). The last statement holds even when there are a big number of rectangles, as in the text-editing situation, where there were more than 100 rectangles in some frames. (The maximum time in the case C is lower than the mean time in case B.)

7.3 Performance of the final 3D VNC client

Measuring the performance of the final version of the library functions have been made by testing a few different usage situations. These situations are supposed to represent normal situations appearing in usual computer work. The tests have also been performed on a few different hardware setups, including different network configurations.

The following scenarios were tested:

Mouse movements: The mouse cursor was moved both within a small window and over the full screen.

Text editing: A predefined text from a book was typed in without errors.

Web browsing: The user started at a specific web location and, only by use of the mouse, browsed to another location, several links away.

Video streaming: A few MPEG and AVI video-clips with different resolutions were played at the server.

Web camera: A web camera was connected to the server and set for a few different resolutions.

All tests was performed running the VNC server under Linux on a standard PC (P3, 750 MHz) with a resolution of 1024x768 pixels and a colour depth of 32 bits/pixel. The client program was a simple 3D application in the VRT environment, consisting of a single `vRT_VNCWindow` object with a texture of 1024x1024 pixels. This program was run on a few different Windows computers, some of them in the same LAN as the VNC server and one of them in a different LAN, bridged with the LAN the server was running on.

Two quantities were measured, the frame rate and the request-reply time. The frame rate is the number of complete VNC update messages received per second. The request-reply time is the time between the sending of an update request and the reception of the first block of data in the update message. 1000 samples were measured and recorded for each test setup and the results from each test are recorded in the diagrams in figures 7 to 9. In the diagrams, as well as in some of the earlier diagrams, the bars represent the intervals between the 5% percentile and the 95% percentile, meaning 90% of the measured values will lie inside these intervals. The vertical lines represent the minimum and maximum measured values respectively. As in previous experiments, the first few samples were removed, since they differed a lot from the rest.

The diagrams show that the frame rates for mouse cursor movements are very high, about 25 Hz in all tests. This gives a smoothly moving cursor without flicker. Naturally, there will be a delay of the mouse cursor, due to the network traffic, and that is measured to about 10-20 ms for the cases with server and client on the same LAN and about 30 ms when they are located on different LANs. This is also a good result, and if the local mouse cursor is switched off, the network delay will not disturb the user significantly.

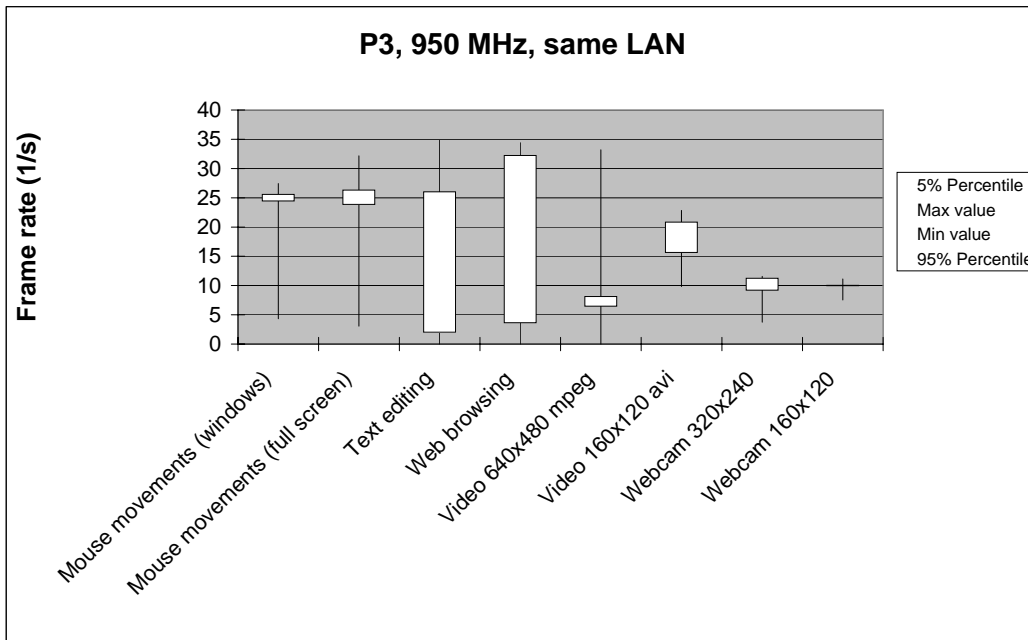
In the text editing tests, as well as in the web browsing tests, there are huge varieties in the frame rates, in some cases they differ from 3 to 120 Hz. The reason for these differences is mainly explained by the high amount of interactivity in this kind of usage. A lot of time passes, when the program is waiting for input from the user, especially in text editing, but also when the user is reading a web page looking for the right link to click. Values in the order of 100 frames per second arise when a lot of text is sent at the same time, for example when a line break occurs or when scrolling in a web browser window. Each update message will then contain very little data, and a lot of these will be sent before the user gives any input. Also the request-reply times varies a lot. This is explained by the fact that most of the time when the client sends a request, nothing has changed at the server. If so, the server will wait with sending the update message until something has changed.

For video streaming applications the results was surprisingly good. For a small (160x120 pixels) AVI video clip the frame rate was about 20 to 50 Hz and for an MPEG movie of full VGA resolution (640x480 pixels) the frame rate was about 8 to 15 Hz, which is very high and even comparable to locally, run video playing applications. The request-reply times are in the order of 3 to 9 ms for the high-resolution MPEG video streaming and a bit higher for the AVI video. This introduced delay has little or no effect on streaming applications, since there is no

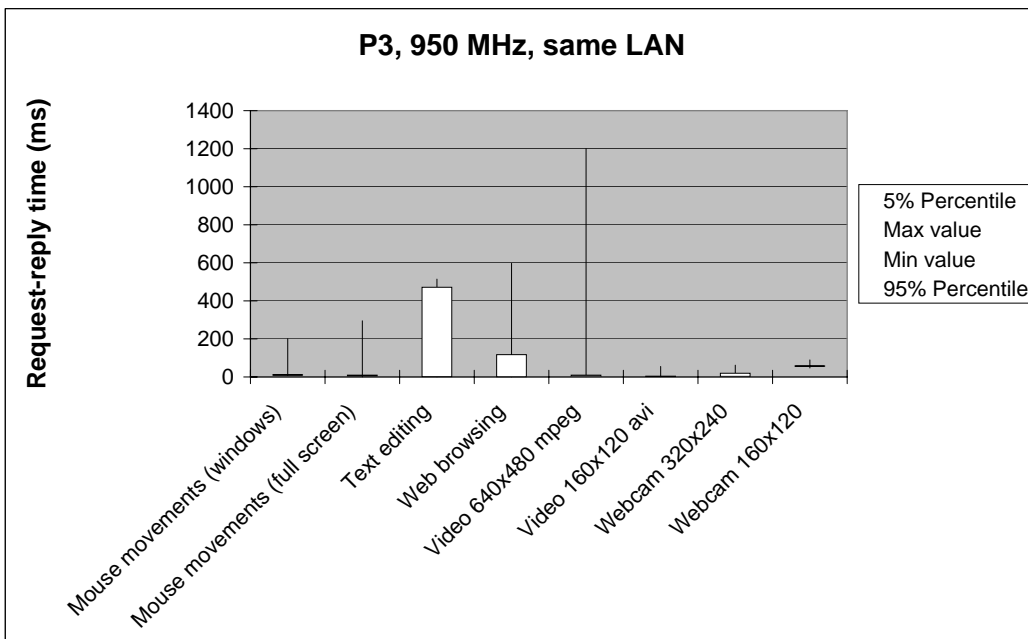
interaction from the user.

Finally, some tests were made with a web cam connected to the server. The results turned out to be about the same as for streaming MPEG and AVI video. Frame rates of about 10 Hz were reached for a resolution of 320x240 pixels, which is comparable to common video conferencing systems. This brings many possibilities, since it means it is possible to have a live video conferencing directly in the 3D environment.

What can also be concluded from the diagrams, is that the hardware on the client side has some influence on the results. The biggest difference between the classic VNC client, `vncviewer`, and the 3D VNC client used here, is that the 3D VNC client has to replace a texture for every frame. This task is done by hardware, and its efficiency is therefore highly dependent on the hardware. It can be seen that the clients with GeForce hardware generally performs better, especially in the video streaming applications. It can also be hinted from the diagrams that the network configuration has rather little influence on the results. Neither the frame rates nor the request-reply times differ significantly between the tests run within the same LAN and those run between different LANs.

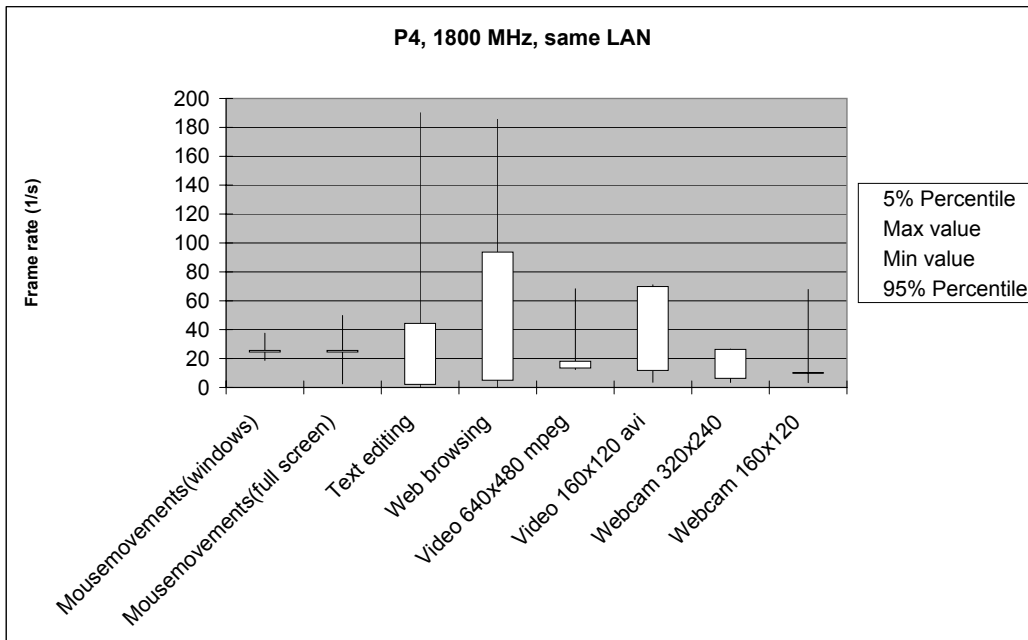


(a) Frame rates

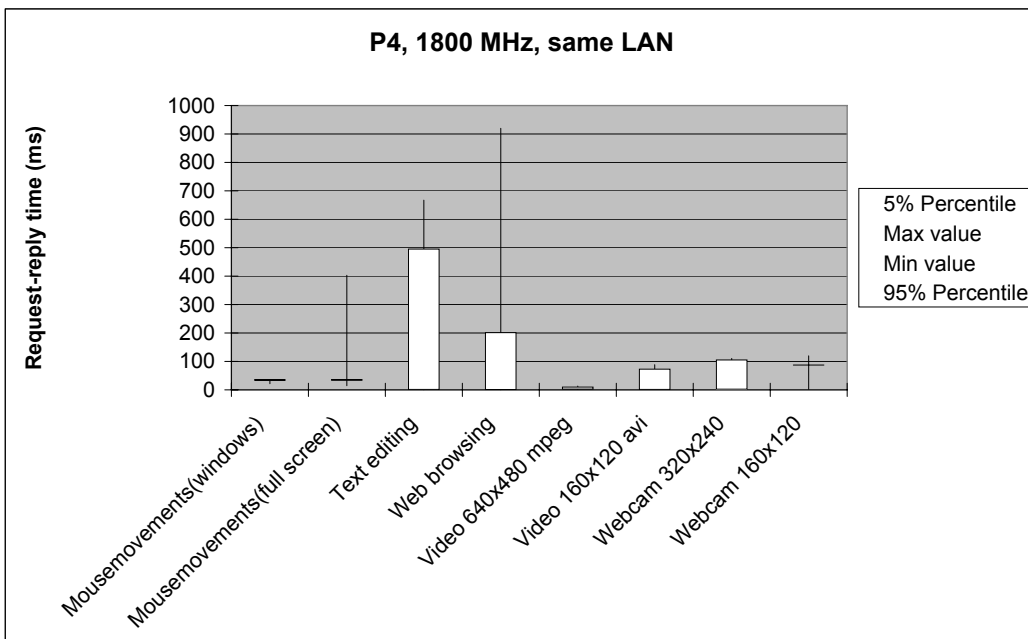


(b) Request-reply times

Figure 7: Performance of the final 3D VNC client on a Pentium 3, 950 MHz with an ATI Rage 128 graphics card on the same LAN as the server.



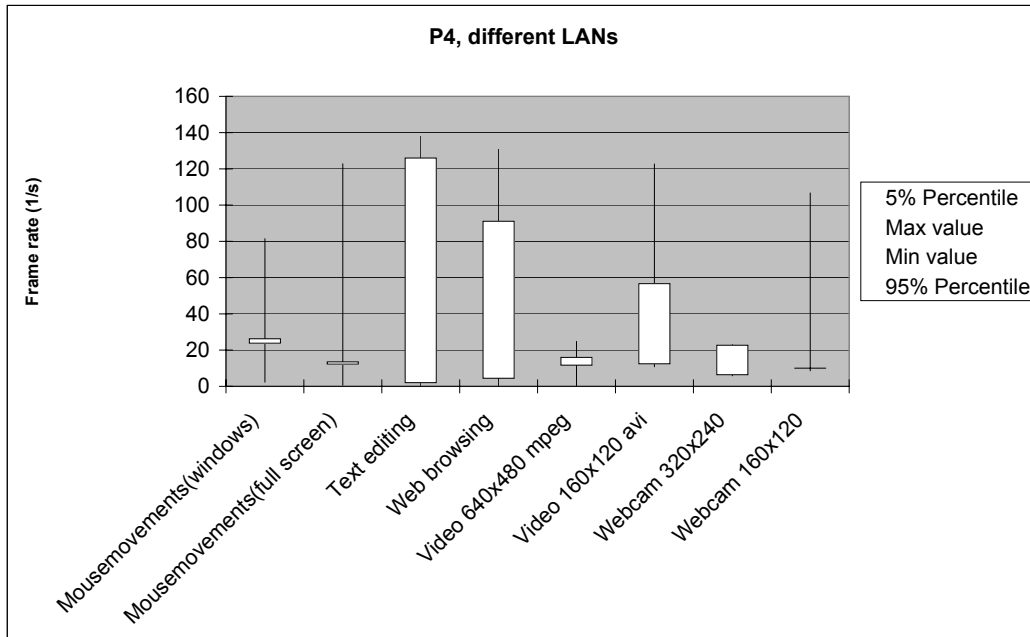
(a) Frame rates



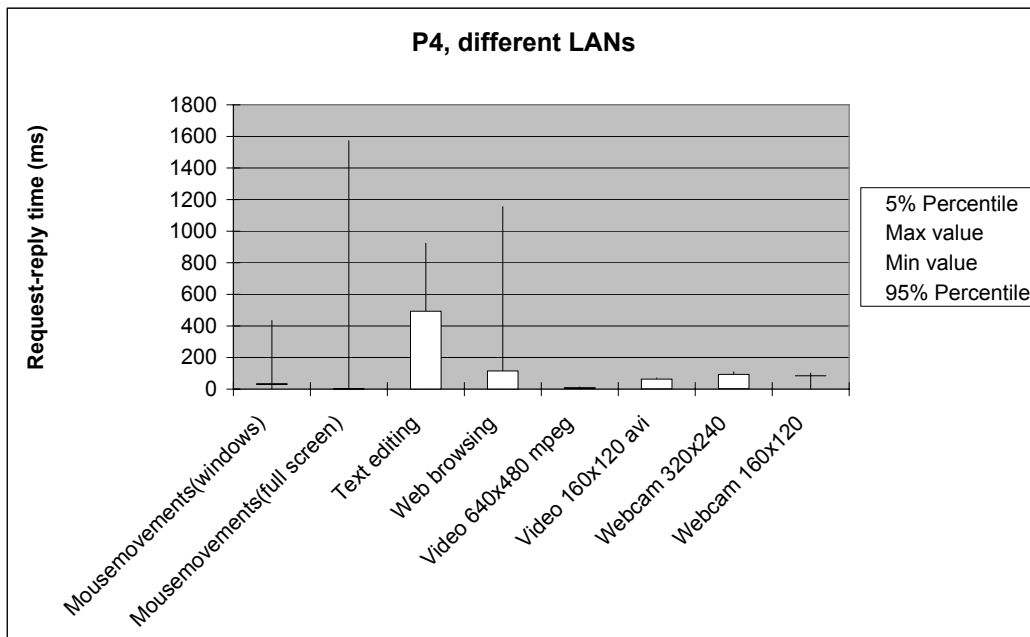
(b) Request-reply times

Figure 8: Performance of the final 3D VNC client on a Pentium 4, 1800 MHz with a GeForce4 graphics card on the same LAN as the server.

Components for Application Sharing in 3D Graphical Environments based on VNC



(a) Frame rates



(b) Request-reply times

Figure 9: Performance of the final 3D VNC client on a Pentium 4, 1900 MHz with a Geforce2 graphics card on different LAN from the server.

7.4 Usage situations

The 3D VNC library for the VRT environment is currently used in some projects at Uppsala University. This section briefly describes the current usage situations and presents some screen shots of the applications.

VASE

The plug-in module for VASE, described in section 6 is the primary application of the 3D VNC API. On the VASE platform it is possible to build up a complex, network shared, 3D environment with several users. In this 3D environment, a number of virtual VNC terminals can be placed. The positions, sizes and VNC servers to connect to can be specified in the VASE configuration file for that environment. It is also possible to place black, unconnected VNC terminals in the environment, together with text terminals. The user can then, at run time, log in to an arbitrary VNC server by use of the text terminals.

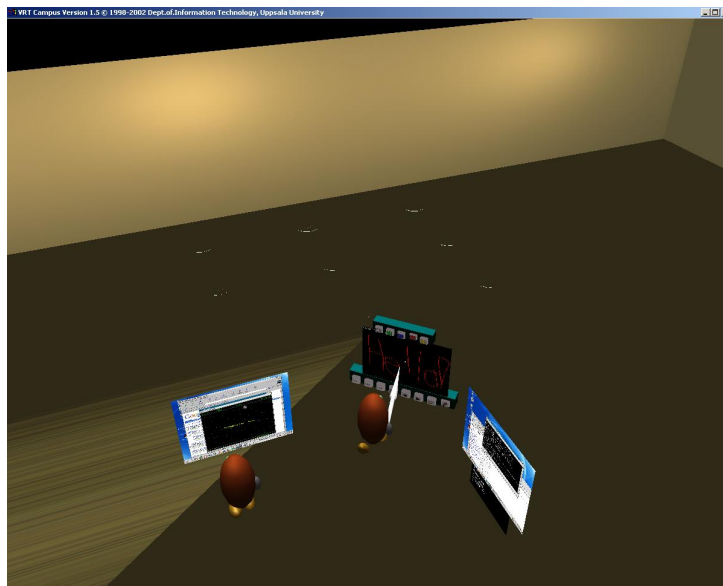


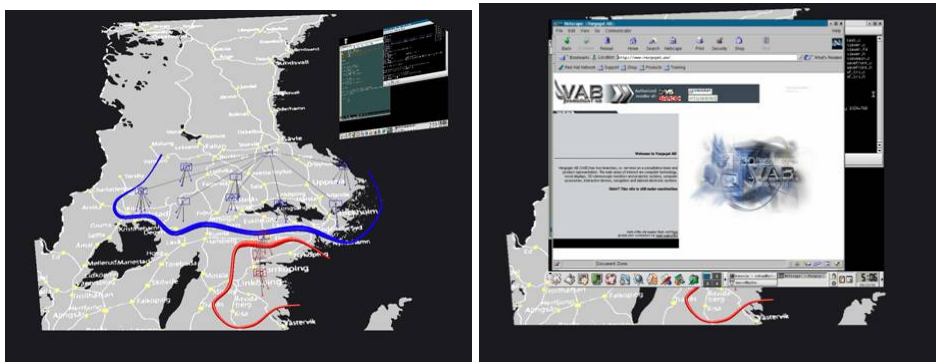
Figure 10: *3D VNC clients in the Visualisation And Simulation Environment.*

Currently a classroom environment is implemented in VASE. Several other plug-ins, in addition to the VNC terminal, exist for use in a classroom environment. These include the avatar that is a representation of the user, a white board, a slide show viewer and much more. In this environment, any user can move its avatar

and interact with the different plug-ins. If one user for example runs a program on one of the VNC terminals, it will run on the other users VNC terminals as well. Figure 10 shows a screen shot of the classroom application with two users, a white board (or actually a black board) and two VNC terminals.

AQUA

Another situation where the 3D VNC library has been used is in the first demonstrator in a project for the Swedish National Defence College, the AQUA project. Here a large stereoscopic display is used to display a map with military symbols and other strategic information. A 3D VNC window is used to display information about some of the symbols in the map. In the normal usage situation, the VNC window is placed in the background, as can be seen in figure 11(a). When the user selects a specific symbol in the map, the VNC window comes up in the foreground and a web browser run on the VNC server automatically browses to a web page with information related to that symbol. This is displayed in figure 11(b).



(a) In the normal usage situation the VNC window is in the background.

(b) When a certain symbol is selected, the VNC window comes up to the foreground.

Figure 11: *The AQUA project at the Swedish National Defence College*

Testing readability of tilted text

There are plans for using the 3D VNC client for testing the readability of tilted text. The idea is that a text that is tilted to a certain degree, can still be read without problems. This makes it possible to, in software, rotate a text area around the horizontal axle and in this way make it take up less space in the vertical direction. It would then be possible to squeeze in more text on small displays, in for example hand held devices like cell phones and PDAs.

With a 3D VNC client any application can be run on a VNC server, and it can be displayed in any orientation on the client side. This way it is possible to test many different types of text in many different orientations. An example of a tilted VNC screen is shown in figure 12.

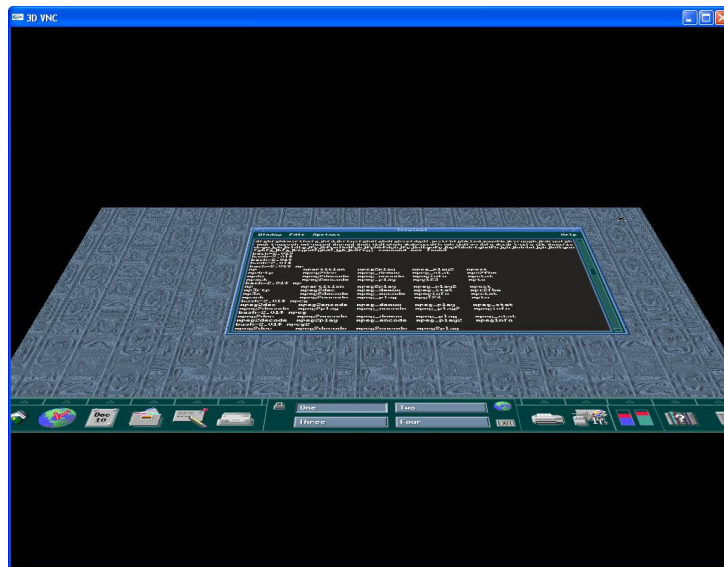


Figure 12: A demonstrator for testing readability of tilted text.

8 Future work

This section will mention obvious improvements and extensions that could be made to improve the performance and usability of the RFB library, as well as some ideas for completely different usage of the VNC system.

Components for Application Sharing in 3D Graphical Environments based on VNC

8.1 Further optimisation of the library functions

Although the tests in the previous section showed good results, the RFB library functions are not perfect. You can still notice a big difference in update speed between the 3D VNC client and the original 2D client, `vncviewer`. This difference is impossible to eliminate, due to fundamental differences in the way data is handled, but it would be theoretically possible to reduce it further. The optimisations described in the earlier sections have increased the update speed almost four times compared to the first prototypes. At the same time, most of the jumpy and flickery behaviour introduced in the 3D environment have been eliminated, almost without loss of speed.

Possible optimisations that can still be done include finding a method to directly copy the incoming, decoded data to the hardware texture buffer, instead of first copying it to a separate memory location. It would probably also be possible to optimise the low-level pixel manipulating routines in the library, possibly by introducing some inline assembler code.

It would also be useful to implement some of the functionality that is supported by the RFB protocol, but has been left out of the RFB library. This includes the use of some encodings and the support for other pixel depths than 32 bits.

8.2 API functions for the GLUT platform

Implementation of API functions on a higher level than those provided by the RFB library, for other platforms than VRT would increase the usability of the library. The most natural choice would probably be for the GLUT platform, since GLUT is available for many operating systems including UNIX, Linux and Windows. The functionality that is missing in the RFB library, and that cannot be implemented on that level is primary the mouse and keyboard filtering described in section 3.4.

8.3 Evaluating the possibilities for other types of usage

The fact that the RFB library provides the client application with raw pixel data from the remote VNC server gives the user many possibilities on how to use the data. It would for example be possible to save the data to file and in this way record

a video of what is happening on the remote computer. If the data is recorded before the decoding, it could also be saved as a compressed video sequence.

Another interesting possibility with VNC would be to add some “real” 3D features to the client, meaning that the data would not only be a 2D projection onto a flat surface in the 3D space, but has some actual depth. Without modifying the VNC server, it would be possible to extract separate windows from the server frame buffer, and arrange them in a more 3D oriented way at the client side. This would make it possible to arrange the windows in a way similar to that presented by the Microsoft Task Gallery presented in the previous work section above.

References

- [1] WEB page. The x windows system. URL www.x.org. 6
- [2] Kenneth R. Wood Tristan Richardson, Quentin Stafford-Fraser and Andy Hopper. Virtual network computing. *IEEE Internet Computing*, 2(1):33–38, January 1998. URL <http://www.uk.research.att.com/pub/docs/att/tr.98.1.pdf>. 6
- [3] Kenneth R. Wood Tristan Richardson. *The RFB Protocol*, 1998. URL <http://www.uk.research.att.com/vnc/rfbproto.pdf>. 8, 10
- [4] Marcus Haupt Steven Feiner, Blair MacIntyre and Eliot Solomon. Windows on the world: 2d windows for 3d augmented reality. November 1993. URL <http://www.cs.columbia.edu/graphics/projects/wotw/>. 11
- [5] George Robertson et al. The task gallery: A 3d window manager. 2000. URL <http://research.microsoft.com/ui/TaskGallery>. 11
- [6] Stefan Seipel. *Virtual Reality Toolkit 1.5 - Programmer's Manual*, 2002. 12, 16
- [7] Björn Andersson. Vase - a functional framework for flexible configuration of virtual and distributed teaching environments. 2002. 12
- [8] Mark J. Kilgard. *OpenGL Utility Toolkit (GLUT) Programming Interface, API Version 3*, 1996. URL <http://www.opengl.org/developers/documentation/glut>. 16, 36
- [9] Brian Hall. *Beej's Guide to Network Programming Using Internet Sockets*, 2001. URL <http://www.ecst.csuchico.edu/~beej/guide/net/bgnet.pdf>. 19
- [10] U.S. Departement of Commerce. *Data Encryption Standard (DES)*, 1999. URL <http://csrc.nist.gov/publications/fips/fips46-3/fips46-3.pdf>. 19

A RFB reference manual

A.1 RFB_state Struct Reference

```
#include <rfb.h>
```

A.1.1 Detailed Description

Struct for state of current connection.

There exists one RFB_state object for each VNC connection. The RFB_state object holds all information about the current state of the connection.

Data Fields

Common state variables

- int [serverSocket](#)
Socket for network communication.
- CARD16 [port](#)
Port number offset.
- CARD16 [RFB_width](#)
Width of remote screen.
- CARD16 [RFB_height](#)
Height of remote screen.
- CARD8 [RFB_bigEndian](#)
Endian of color data from server.
- int [first](#)
A Flag indicating that this is the first frame.

- CARD32 `netbuf` [TEXTURE_SIZE *TEXTURE_SIZE]
Unformatted texture data.
- CARD32 `texbuf` [TEXTURE_SIZE *TEXTURE_SIZE]
Formatted texture data.

Special state variables for non blocking read

- int `non_blocking_priority`
Priority for non blocking read Affects how much data is read at each call to RFB_Update.
- int `request_sent`
Flag indicating that an Update Request message has been sent to the server.
- int `reading`
Flag that is set when a non blocking read is started, and cleared when finished.
- int `reading_subrects`
Flag that is set when not all subrectangles are read, but reading might be false.
- int `current_subrect`
The number of the subrectangle currently being read.
- int `bytes_to_read`
Number of bytes to read with BeginRead and ContinueRead.
- int `read_bytes`
Number of bytes read so far.
- char * `buffer`
A memory buffer where the read data is saved temporarily.
- CARD8 `msgType`
Type of the current Update message.

- rfbFramebufferUpdateMsg [Update](#)
Header for the current Update message.
- rfbFramebufferUpdateRectHeader [RectHeader](#)
Header for the current rectangle being read.

Special state variables for non blocking hextile. Most of them are only interesting locally, but has to be preserved between calls.

- CARD8 [hex_flags](#)
An array of flags used for reading non blocking hextile rectangles.
- rfbRectangle [hex_r](#)
The position and dimensions of the hextile rect currently being read.
- CARD8 [hex_subencoding](#)
The encoding of the hextile rect currently being read.
- CARD8 [hex_nSubrects](#)
The number of subrectangles of the hextile rect currently being read.
- CARD32 [hex_bg](#)
Temporary variable for background color of current hextile.
- CARD32 [hex_fg](#)
Temporary variable for foreground color of current hextile.
- CARD8 * [hex_ptr](#)
Memory pointer in temporary memory buffer.

Variables for benchmarking.

- int [benchmarking](#)
Flag indicating that benchmark mode is active.
- FILE * [outputfile](#)
File for benchmark output.

- double [request_time](#)
Time for request sent.
- double [stop_time](#)
Time for frame recieved.
- int [request_timer_started](#)
Flag indicating that request-reply timer is started.

Number of updated rects and positions.

- rfbRectangle [updated_rects](#) [SUBTEXTURING_MAX_SUBRECTS]
Coordinates for updated rectangles.
- CARD16 [update_xmin](#)
- CARD16 [update_xmax](#)
- CARD16 [update_ymin](#)
- CARD16 [update_ymax](#)

A.2 rfb.h File Reference

A.2.1 Detailed Description

Declarations of constants, structures and functions used by RFB.

Author:

Robin Stridh

Date:

2002-06-14

Data Structures

- struct [RFB_state](#)
Struct for state of current connection.

- struct **RFB_Error**

User functions

- **RFB_state** * **RFB_Init** (char *server, int port, char *password)
Initializes state variable and connects to VNC server.
- void **RFB_Quit** (**RFB_state** *s)
Closes connection with VNC server and frees allocated memory.
- void **RFB_SetPriority** (**RFB_state** *s, int priority)
Sets the priority for the non blocking VNC window updates.
- int **RFB_BenchmarkInit** (**RFB_state** *s, char *filename)
Activates the benchmark mode.
- int **RFB_DebugMode** (int mode)
Changes the mode for debug information.
- int **RFB_Update** (**RFB_state** *s, int blocking, CARD16 x, CARD16 y, CARD16 w, CARD16 h)
Update function.
- int **RFB_UpdateFullScreen** (**RFB_state** *s, int blocking)
Same as RFB_Update, but always requests updates for the whole screen.
- CARD8 * **RFB_GetTexels** (**RFB_state** *s)
Returns a pointer to data updated by RFB_Update.
- int **RFB_GetUpdatedRects** (**RFB_state** *s, rfbRectangle **rects)
Returns information about what data has been changed by RFB_Update.
- int **RFB_HandleMouseEvent** (**RFB_state** *s, CARD16 x, CARD16 y, CARD8 mask)
Notifies the server that the mouse has been moved or clicked.

- int [RFB_HandleKeyEvent](#) ([RFB_state](#) *s, CARD32 key, CARD8 down)
Notifies the server that a key has been pressed or released.
- int [RFB_GetLastError](#) (char *error_message, int size)
Get error code and error message for the last RFB error.
- int [RFB_PrintLastError](#) ()
Prints the last error message to stdout.

Local functions

- int [SetupConnection](#) ([RFB_state](#) *s, char *server)
- int [CloseConnection](#) ([RFB_state](#) *s)
- int [InitMessages](#) ([RFB_state](#) *s, char *passwd)
- int [SetEncodings](#) ([RFB_state](#) *s)
- int [SendUpdateRequest](#) ([RFB_state](#) *s, CARD16 x, CARD16 y, CARD16 w, CARD16 h)
- int [RecieveScreenBlocking](#) ([RFB_state](#) *s)
- int [RecieveScreenNonBlocking](#) ([RFB_state](#) *s)
- void [ShiftColor](#) (CARD32 *pcol)
- void [SetPixels](#) ([RFB_state](#) *s, CARD16 x, CARD16 y, CARD16 w, CARD16 h)
- void [FillRectangle](#) ([RFB_state](#) *s, CARD16 x, CARD16 y, CARD16 w, CARD16 h, CARD32 color)
- void [CopyRectangle](#) ([RFB_state](#) *s, CARD16 dx, CARD16 dy, CARD16 w, CARD16 h, CARD16 sx, CARD16 sy)
- int [ReadRawRect](#) ([RFB_state](#) *s, CARD16 x, CARD16 y, CARD16 w, CARD16 h, int blocking)
- int [ReadCopyRect](#) ([RFB_state](#) *s, CARD16 x, CARD16 y, CARD16 w, CARD16 h, int blocking)
- int [ReadRRERect](#) ([RFB_state](#) *s, CARD16 x, CARD16 y, CARD16 w, CARD16 h)
- int [ReadHextileRect](#) ([RFB_state](#) *s, CARD16 rx, CARD16 ry, CARD16 rw, CARD16 rh)

- int `ReadHextileRectNonBlocking` (`RFB_state` *s, CARD16 rx, CARD16 ry, CARD16 rw, CARD16 rh)
- int `ReadExact` (`RFB_state` *s, char *buf, int len)
- int `WriteExact` (`RFB_state` *s, char *buf, int len)
- int `PeekType` (`RFB_state` *s, CARD8 *msgType)
- void `BeginRead` (`RFB_state` *s, char *buf, int len)
- int `ContinueRead` (`RFB_state` *s)
- void `RFB_Out` (char *fmt,...)
- void `RFB_DebugOut` (char *fmt,...)
- void `RFB_DebugErr` (char *fmt,...)

Typedefs

- typedef unsigned long `CARD32`
- typedef unsigned short `CARD16`
- typedef unsigned char `CARD8`

A.2.2 Function Documentation

A.2.2.1 `RFB_state*` `RFB_Init` (char * *server*, int *port*, char * *password*)

Initializes state variable and connects to VNC server.

This function will create an object of type `RFB_state` that will contain all necessary status variables about the connection, once it has been established, and also information about the current status with the remote desktop. `RFB_Init` will run the functions `RFB_SetupConnection`, `RFB_InitMessages` and `RFB_SetEncodings` in order and then return a pointer to the `RFB_state` variable it creates.

Parameters:

server An ascii string with the server name.

port Port number offset for the connection (actual port number is port + PORTBASE where PORTBASE = 5900 for VNC).

password An ascii string with the server password.

Returns:

A pointer to the state variable to be used in further operations and NULL if connection failed. If NULL is returned, the cause of error can be retrieved by `RFB_GetLastError`.

See also:

[RFB_GetLastError](#)

[RFB_Quit](#)

A.2.2.2 void RFB_Quit (RFB_state * s)

Closes connection with VNC server and frees allocated memory.

Parameters:

s A pointer to the state variable to operate on.

A.2.2.3 void RFB_SetPriority (RFB_state * s, int priority)

Sets the priority for the non blocking VNC window updates.

The priority determines how much data will be read from the server in each call to `RFB_Update`. The priority can range from 1 to infinity, but values higher than about 100 will not give better performance. If higher priority is desired, use the blocking flag in the `RFB_Update` function instead. For applications where high interaction speed with the 3D environment is desired, set the priority value low, and for applications where the high update frequency of and interaction speed with the VNC server is desired, the value should be set high or the blocking flag in the `RFB_Update` function should be set. The priority is set to 20 by default, which will give high interaction speed with the environment and a fast VNC window update frequency for 3D applications with a light CPU load. For heavier processes the priority might have to be raised, at the cost of interaction speed.

Parameters:

s A pointer to the state variable to operate on.

priority The new priority value.

See also:

vRT_VNCWindowUpdate

[RFB_Update](#)

A.2.2.4 int RFB_BenchmarkInit (RFB_state * s, char * filename)

Activates the benchmark mode.

In benchmark mode statistics of update framerate and request-reply time is written to a file.

Parameters:

s A pointer to the state variable to operate on.

filename The name of the file to write data on. Previous contents will be erased.

Returns:

1 if file could be opened, -1 otherwise.

A.2.2.5 int RFB_DebugMode (int mode)

Changes the mode for debug information.

If debug mode is used, all output printed with RFB_Out, RFB_DebugOut and RFB_DebugErr will be written to stdout. If not in debug mode only output from RFB_Out will be printed. Debug mode is turned off as default.

Parameters:

mode 1 turns debug mode on, 0 turns it off.

Returns:

previous mode.

A.2.2.6 int RFB_Update (RFB_state * s, int blocking, CARD16 x, CARD16 y, CARD16 w, CARD16 h)

Update function.

Sends an update request to the server if not already sent and starts receiving an update message. The function can be used in both blocking and non-blocking mode. In blocking mode the program execution will block until the whole update message has been received. In non-blocking mode the control will be returned to the calling program after reading a piece of the update message. The reading will then automatically be resumed when `RFB_Update` is called next time. If used in non-blocking mode the function must be called from inside a loop that ends when 1 is returned. For non blocking update, the amount of data read by each call is determined by the priority set with `RFB_SetPriority`.

Parameters:

s A pointer to the state variable to operate on.

blocking A flag that determines if the function shall be used in blocking (blocking = 1) or non-blocking (blocking = 0) mode.

x X position of upper left corner of requested rectangle.

y Y position of upper left corner of requested rectangle.

w Width of requested rectangle.

h Height of requested rectangle.

Returns:

1 if the whole update message has been received, 0 if the function returned before the whole update message was received and a `RFB_ERR` code if error. If 1 is returned the updated pixel data can be retrieved with `RFB_GetTexels` and the locations of the updated rectangles can be retrieved with `RFB_GetUpdatedRects`.

See also:

[RFB_UpdateFullScreen](#)

[RFB_GetTexels](#)

[RFB_GetUpdatedRects](#)

[RFB_GetLastError](#)

A.2.2.7 int RFB_UpdateFullScreen ([RFB_state](#) * *s*, int *blocking*)

Same as RFB_Update, but always requests updates for the whole screen.

Parameters:

s A pointer to the state variable to operate on.

blocking A flag that determines if the function shall be used in blocking (blocking = 1) or non-blocking (blocking = 0) mode.

Returns:

1 if the whole update message has been received, 0 if the function returned before the whole update message was received and negative if error. If 1 is returned the updated pixel data can be retrieved with RFB_GetTexels and the locations of the updated rectangles can be retrieved with RFB_GetUpdatedRects. The error codes are: RFB_ERR_NO_STATE_POINTER if the state pointer is NULL. RFB_ERR_UNKNOWN on all other errors.

See also:

[RFB_Update](#)
[RFB_GetTexels](#)
[RFB_GetUpdatedRects](#)

A.2.2.8 CARD8* RFB_GetTexels (RFB_state * s)

Returns a pointer to data updated by RFB_Update.

Use RFB_GetUpdatedRects to see what data is actually changed.

Parameters:

s A pointer to the state variable to operate on.

Returns:

A pointer to a data buffer containing the updated look of the remote frame buffer.

See also:

[RFB_Update](#)
[RFB_GetUpdatedRects](#)

A.2.2.9 int RFB_GetUpdatedRects (RFB_state * s, rfbRectangle ** rects)

Returns information about what data has been changed by RFB_Update.

Parameters:

s A pointer to the state variable to operate on.

rects A pointer to an array of rectangle positions and sizes of the updated rectangles.

Returns:

The number of updated rectangles.

See also:

[RFB_Update](#)
[RFB_GetTexels](#)
[rfbRectangle](#)

A.2.2.10 int RFB_HandleMouseEvent (RFB_state * s, CARD16 x, CARD16 y, CARD8 mask)

Notifies the server that the mouse has been moved or clicked.

Parameters:

s A pointer to the state variable to operate on.

x Current x position of the mouse.

y Current y position of the mouse.

mask Bitmask telling which buttons are pressed. 1 = down, 0 = up. LSB = button 1 (left).

Returns:

0 upon success, an RFB_ERR code otherwise.

See also:

[RFB_HandleKeyEvent](#)
[RFB_GetLastError](#)

A.2.2.11 `int RFB_HandleKeyEvent (RFB_state * s, CARD32 key, CARD8 down)`

Notifies the server that a key has been pressed or released.

Parameters:

s A pointer to the state variable to operate on.

key Key code for pressed/released key. VNC uses X-style key codes.

down Flag telling if the button is down (1) or up (0).

Returns:

0 upon success, an RFB_ERR code otherwise.

See also:

[RFB_HandleMouseEvent](#)

[RFB_GetLastError](#)

A.2.2.12 `int RFB_GetLastError (char * error_message, int size)`

Get error code and error message for the last RFB error.

This function is used to retrieve the error code and error message corresponding to the most recently occurred RFB error. It can be used to print error messages to a file.

Parameters:

error_message a string to return the error message in

size size of the string

Returns:

the error code for the last error

A.2.2.13 `int SetupConnection (RFB_state * s, char * server)`

For internal use only.

Sets up connection with the server. This function creates a usual TCP connection to the server using the standard library functions `gethostbyname`, `socket` and `connect`. The socket descriptor is saved in the `RFB_state` variable `*s` that is taken as parameter. Under Windows it is also necessary to call the `WSASocket` function before any Windows socket commands can be run.

Parameters:

- `s` A pointer to the state variable to operate on.
- `server` An ascii string with the server name.

Returns:

0 upon success, -1 otherwise.

A.2.2.14 int CloseConnection (RFB_state * s)**For internal use only.**

Closes the connection with the server.

Parameters:

- `s` A pointer to the state variable to operate on.

Returns:

0 upon success, -1 otherwise.

A.2.2.15 int InitMessages (RFB_state * s, char * passwd)**For internal use only.**

Sends and receives initialisation messages to/from the server. This function takes care of the initial handshaking with the server once a connection exists. It also sends and receives some handshaking messages to/from the VNC server that completely determines the encodings used and the format of the data. First the server protocol version is sent from the server. This has to be received and answered by the protocol version the client can handle, and that will actually be used. (The server is supposed to be backward compatible with older protocol versions, so as long as the client's version is older than the server, the client will decide which version to use.) Next comes the client authentication to the server. The authentication begins with the server sending the authentication method it uses, which is usually "VNC authentication", but can also be "connection failed" or "no authentication". VNC authentication

uses DES encryption with a 16 byte challenge. The challenge is sent by the server and encrypted by the client using the password as key. The resulting, encrypted key is returned to the server for verification. The server then return a status message and if the status is “OK” (or the authentication method is “no authentication”) the client is logged in to the server and the communication is allowed to continue. If the status is “failed” or “too many tries” an error code will be returned in `RFB_last_error_code`. The code for the encryption scheme is taken directly from `vncviewer` by linking the files `d3des.h`, `d3des.c`, `vncauth.h` and `vncauth.c` to the RFB project. Finally the client tells the server if the connection shall be allowed to be shared among several clients and the server responds with information about the remote desktop including size, pixelformat endians and so on.

Parameters:

- s* A pointer to the state variable to operate on.
- password* An ascii string with the server password.

Returns:

- 0 upon success, -1 otherwise.

A.2.2.16 int SetEncodings (RFB_state * s)**For internal use only.**

Tells the server which encodings we can handle. Currently raw encoding, copy rectangle encoding and hextile encoding are supported.

Parameters:

- s* A pointer to the state variable to operate on.

Returns:

- 0 upon success, -1 otherwise.

A.2.2.17 int SendUpdateRequest (RFB_state * s, CARD16 x, CARD16 y, CARD16 w, CARD16 h)**For internal use only.**

Sends update request to server.

Parameters:

- s* A pointer to the state variable to operate on.

x X position of upper left corner of requested rectangle.

y Y position of upper left corner of requested rectangle.

w Width of requested rectangle.

h Height of requested rectangle.

Returns:

0 upon success, -1 otherwise.

A.2.2.18 int RecieveScreenBlocking (RFB_state * s)

For internal use only.

Recieves and interprets a whole update message from the server if one is sent. Returns if not.

Parameters:

s A pointer to the state variable to operate on.

Returns:

1 if a whole update message has been recieved, 0 if no update message was sent (before timeout) and -1 upon errors.

A.2.2.19 int RecieveScreenNonBlocking (RFB_state * s)

For internal use only.

Recieves and interprets parts of an update message from the server if one is sent. For raw encoding the part of the update currently in the buffer will be recieved at most `non_blocking_priority` times. For hextile encoding at most the number of hextiles defined by `non_blocking_priority` will be recieved before return.

Parameters:

s A pointer to the state variable to operate on.

Returns:

1 if a whole update message has been recieved, 0 if no update message was sent (before timeout) or after the specified part of the update message has been recieved and -1 upon errors.

A.2.2.20 void ShiftColor (CARD32 * *pcol*)**For internal use only.**

Shifts the colorvalue from VNC representation to a representation desired by OpenGL and Windows.

Parameters:

pcol A pointer to the colorvalue to be shifted.

A.2.2.21 void SetPixels (RFB_state * *s*, CARD16 *x*, CARD16 *y*, CARD16 *w*, CARD16 *h*)**For internal use only.**

Copies pixelvalues from a temporary network buffer to the specified location in the texture buffer.

Parameters:

s A pointer to the state variable to operate on.

x X position of the upper left corner of the destination rectangle.

y Y position of the upper left corner of the destination rectangle.

w Width of source and destination rectangle.

h Height of source and destination rectangle.

A.2.2.22 void FillRectangle (RFB_state * *s*, CARD16 *x*, CARD16 *y*, CARD16 *w*, CARD16 *h*, CARD32 *color*)**For internal use only.**

Fills the specified rectangle in the texture buffer with the same pixelvalue.

Parameters:

s A pointer to the state variable to operate on.

x X position of the upper left corner of the destination rectangle.

y Y position of the upper left corner of the destination rectangle.

w Width of the destination rectangle.

h Height of the destination rectangle.

color Pixelvalue to set rectangle to.

A.2.2.23 void CopyRectangle (**RFB_state** * *s*, CARD16 *dx*, CARD16 *dy*, CARD16 *w*, CARD16 *h*, CARD16 *sx*, CARD16 *sy*)

For internal use only.

Copies a rectangle of pixelvalues from one position to another in the texture buffer. Works both for forward and backward copying in the case of overlapping memory areas.

Parameters:

- s* A pointer to the state variable to operate on.
- dx* X position of the upper left corner of the destination rectangle.
- dy* Y position of the upper left corner of the destination rectangle.
- w* Width of source and destination rectangle.
- h* Height of source and destination rectangle.
- sx* X position of the upper left corner of the source rectangle.
- sy* Y position of the upper left corner of the source rectangle.

A.2.2.24 int ReadRawRect (**RFB_state** * *s*, CARD16 *x*, CARD16 *y*, CARD16 *w*, CARD16 *h*, int *blocking*)

For internal use only.

Reads a raw encoded rectangle from the server and places the data in the texture buffer.

Parameters:

- s* A pointer to the state variable to operate on.
- x* X position of the upper left corner of the rectangle.
- y* Y position of the upper left corner of the rectangle.
- w* Width of the rectangle.
- h* Height of the rectangle.
- blocking* Flag indicating if the reading shall block until the whole rectangle is read, or if the function shall return earlier.

Returns:

- 1 if the whole rectangle is read, 0 if the reading was interrupted, and -1 on errors.

A.2.2.25 `int ReadCopyRect (RFB_state * s, CARD16 x, CARD16 y, CARD16 w, CARD16 h, int blocking)`

For internal use only.

Reads a copy rectangle encoded rectangle from the server and places the data in the texture buffer.

Parameters:

s A pointer to the state variable to operate on.

x X position of the upper left corner of the rectangle.

y Y position of the upper left corner of the rectangle.

w Width of the rectangle.

h Height of the rectangle.

blocking Flag indicating if the reading shall block until the whole rectangle is read, or if the function shall return earlier.

Returns:

1 if the whole rectangle is read, 0 if the reading was interrupted, and -1 on errors.

A.2.2.26 `int ReadRRERect (RFB_state * s, CARD16 x, CARD16 y, CARD16 w, CARD16 h)`

For internal use only.

Reads a RRE encoded rectangle from the server and places the data in the texture buffer. Only implemented for blocking read.

Parameters:

s A pointer to the state variable to operate on.

x X position of the upper left corner of the rectangle.

y Y position of the upper left corner of the rectangle.

w Width of the rectangle.

h Height of the rectangle.

Returns:

1 if the whole rectangle is read and -1 on errors.

A.2.2.27 `int ReadHextileRect (RFB_state * s, CARD16 rx, CARD16 ry, CARD16 rw, CARD16 rh)`

For internal use only.

Reads a hextile encoded rectangle from the server and places the data in the texture buffer. This function will block the program until the whole rectangle has been recieved.

Parameters:

- s* A pointer to the state variable to operate on.
- rx* X position of the upper left corner of the rectangle.
- ry* Y position of the upper left corner of the rectangle.
- rw* Width of the rectangle.
- rh* Height of the rectangle.

Returns:

1 if the whole rectangle is read and -1 on errors.

A.2.2.28 `int ReadHextileRectNonBlocking (RFB_state * s, CARD16 rx, CARD16 ry, CARD16 rw, CARD16 rh)`

For internal use only.

Reads a hextile encoded rectangle from the server and places the data in the texture buffer. This function will try to read one hextile and then return, to read the next hextile the next time it is called. If a whole hextile cannot be recieved immediately, the function will return anyway and save the current status to resume the reading the next time it is called.

Parameters:

- s* A pointer to the state variable to operate on.
- rx* X position of the upper left corner of the rectangle.
- ry* Y position of the upper left corner of the rectangle.
- rw* Width of the rectangle.
- rh* Height of the rectangle.

Returns:

1 if all hextiles are read, 0 if reading was interrupted or if there are more hextiles to read and -1 on errors.

A.2.2.29 int ReadExact (RFB_state * s, char * buf, int len)**For internal use only.**

Reads exactly the specified number of bytes to the specified buffer and will block the program execution until all has been received.

Parameters:

s A pointer to the state variable to operate on.

buf A pointer to a buffer where the data will be stored. (The user must make sure it can hold at least *len* bytes.)

len The number of bytes to be read.

Returns:

0 upon success, -1 otherwise.

See also:

[WriteExact](#)

[BeginRead](#)

[ContinueRead](#)

A.2.2.30 int WriteExact (RFB_state * s, char * buf, int len)**For internal use only.**

Writes exactly the specified number of bytes from the specified buffer and will block the program execution until all has been transmitted.

Parameters:

s A pointer to the state variable to operate on.

buf A pointer to a buffer from where the data will be sent. (The user must make sure it holds at least *len* bytes of valid data.)

len The number of bytes to be sent.

Returns:

0 upon success, -1 otherwise.

See also:

[ReadExact](#)

[BeginRead](#)

[ContinueRead](#)

A.2.2.31 int PeekType (RFB_state * s, CARD8 * msgType)**For internal use only.**

Reads the first byte from the socket buffer without removing it. If the sent data is a message from the server this will be the type of the message. If no message is recieved before a given timeout the function will return. The timeout is specified by RFB_PEEK_TYPE_TIMEOUT.

Parameters:

s A pointer to the state variable to operate on.

msgType A pointer to a varable where the type will be returned.

Returns:

1 upon succes, 0 upon timeout (*msgType* will not be valid) and -1 otherwise.

A.2.2.32 void BeginRead (RFB_state * s, char * buf, int len)**For internal use only.**

Initializes a non blocking read but does not perform any actual reading. This function, if used together with ContinueRead in a loop (as done in the macro READ_AND_RETURN), can be used as ReadExact to read exactly the specified number of bytes to the specified buffer but will not block the program execution if some of the data is delayed.

Parameters:

s A pointer to the state variable to operate on.

buf A pointer to a buffer where the data will be stored. (The user must make sure it can hold at least *len* bytes.)

len The number of bytes to be read.

See also:

[ContinueRead](#)

[ReadExact](#)

READ_AND_RETURN

A.2.2.33 int ContinueRead (RFB_state * s)**For internal use only.**

Continues a non blocking read initialized by BeginRead. This function will read all data currently in the socket buffer, but at most the remaining part of

what was specified by `BeginRead`. If the buffer is empty the function will return. Used only together with `BeginRead`.

Parameters:

s A pointer to the state variable to operate on.

Returns:

0 upon success, -1 otherwise.

See also:

[BeginRead](#)

[ReadExact](#)

`READ_AND_RETURN`

A.2.2.34 void RFB_Out (char * *fmt*, ...)

For internal use only.

Prints a formatted string to stdout.

Parameters:

fmt The formatted string. Same formatting as `printf`.

Parameters to the string.

A.2.2.35 void RFB_DebugOut (char * *fmt*, ...)

For internal use only.

Prints a formatted string to stdout, only if in debug mode.

Parameters:

fmt The formatted string. Same formatting as `printf`.

Parameters to the string.

A.2.2.36 void RFB_DebugErr (char * *fmt*, ...)

For internal use only.

Prints a formatted string to stdout, appended with the last system error message, only if in debug mode.

Parameters:

fmt The formatted string. Same formatting as `printf`.

Parameters to the string.

B VRT VNC reference manual

B.1 vRT_VNCWindow Struct Reference

```
#include <vrt_vnc.h>
```

B.1.1 Detailed Description

Struct for holding information about the current VNC window connection.

A pointer to a vRT_VNCWindowtag is returned by vRT_VNCWindowNew and must be supplied with all calls to vRT_VNCWindow functions.

Data Fields

- void * [state](#)

*A pointer to a an RFB_state struct containing all information neccesary for the communication with the VNC server (RFB_state *state was changed to void *state to avoid including rbf.h).*

- VRT_Geometry * [g_win](#)

Pointer to a VRT geometry representing the VNC window.

- VRT_Htx [t_win](#)

Handle to a VRT texture with VNC data, mapped onto the polygon.

- VRT_HPlg [p_win](#)

Handle to a VRT polygon onto which the VNC texture is mapped.

- unsigned char [texels](#) [1024 *1024 *4]

The pixeldata from the VNC server.

B.2 vrt_vnc.h File Reference

B.2.1 Detailed Description

A wrapper around the functions in rfb.h / rfb.lib for use in VRT.

Author:

Robin Stridh

Date:

2002-08-20

Data Structures

- struct [vRT_VNCWindow](#)

Struct for holding information about the current VNC window connection.

User functions

- [vRT_VNCWindow](#) * [vRT_VNCWindowNew](#) (char *serverName, int portNo, char *passwd)

Creates a new VNC window object, which consists of a VRT geometry consisting of a quadric polygon with the VNC window texture mapped onto it together with an instance of RFB_state, which contains all necessary information about the VNC server.

- int [vRT_VNCWindowClose](#) ([vRT_VNCWindow](#) *win)

Closes the connection with a VNC server and deallocates the corresponding VRT structures.

- int [vRT_VNCSetPriority](#) ([vRT_VNCWindow](#) *win, int priority)

Sets the priority for the non blocking VNC window updates.

- int `vRT_VNCWindowUpdate` (`vRT_VNCWindow *win`, int blocking)

Sends an update request to the VNC server, if one is not already sent, and receives an update message if something has changed at the server desktop.
- int `vRT_VNCWindowUpdatePartial` (`vRT_VNCWindow *win`, int blocking, int x, int y, int w, int h)

Same as `vRT_VNCWindowUpdate`, but requests only the part of the screen within the specified rectangle.
- int `vRT_VNCHandleMouseEvent` (`VRT_Node *node`, `vRT_VNCWindow *win`, int x, int y, unsigned int winmask)

Sends a message to the VNC server that the mouse has been moved or clicked.
- int `vRT_VNCHandleKeyEvent` (`vRT_VNCWindow *win`, int virtkey, DWORD keyData)

Sends a message to the VNC server that a keyboard key has been pressed or released.
- int `vRT_VNCSendString` (`vRT_VNCWindow *win`, char *str)

Sends an ascii string to the VNC server by repeated calls to `vRT_VNCHandleKeyEvent`.
- int `vRT_VNCSendStringR` (`vRT_VNCWindow *win`, char *str)

Sends an ascii string to the VNC server followed by a line feed (or return), otherwise the same as `vRT_VNCSendString`.
- int `vRT_VNCGetLastError` (char *error_message, int size)

Returns the error code and error message for the last occurred `VRT_VNC` error.

Local functions

- int `RFB2VRT_ERR` (int rfb_error_code)

Defines

- #define `EXTERN` extern

- #define `VRT_VNC_H_VER` "0.5"
Version of the vrt_vnc.h file.
- #define `VNC_TEXTURE_SIZE` 1024
Size of the allocated texture to hold the VNC frame buffer.

Variables

- const char * `vrt_vnc_h_ver`

B.2.2 Function Documentation

B.2.2.1 `vRT_VNCWindow*` `vRT_VNCWindowNew` (`char * serverName`, `int portNo`, `char * passwd`)

Creates a new VNC window object, which consists of a VRT geometry consisting of a quadric polygon with the VNC window texture mapped onto it together with an instance of `RFB_state`, which contains all necessary information about the VNC server.

The default size of the quad is 1.0 by 1.0 VRT units. To retrieve a pointer to the geometry use `vnc->g_win`, if `vnc` is the pointer returned by `vRT_VNCWindowNew`. After the VNC window has been created the function `vRT_VNCWindowUpdate` has to be called as often as possible to update the look of the remote desktop.

Parameters:

- serverName* An ascii string containing the name of the VNC server.
- portNo* Port number offset for the connection (actual port number is `portNo + PORTBASE` where `PORTBASE = 5900` for VNC).
- passwd* An ascii string with the server password.

Returns:

A pointer to a `vRT_VNCWindow` struct to be used in further operations. If the connection fails NULL is returned and an error code can be retrieved with `vRT_VNCGetLastError`.

See also:[vRT_VNCGetLastError](#)**B.2.2.2 int vRT_VNCWindowClose (vRT_VNCWindow * win)**

Closes the connection with a VNC server and deallocates the corresponding VRT structures.

Parameters:

win A pointer to the [vRT_VNCWindow](#) to close.

Returns:

1 on successful shutdown, 0 otherwise.

B.2.2.3 int vRT_VNCSetPriority (vRT_VNCWindow * win, int priority)

Sets the priority for the non blocking VNC window updates.

The priority determines how much data will be read from the server in each call to [vRT_VNCWindowUpdate](#). The priority can range from 1 to infinity, but values higher than about 100 will not give better performance. If higher priority is desired, use the blocking flag in the [vRT_VNCWindowUpdate](#) function instead. For applications where high interaction speed with the VRT environment is desired, set the priority value low, and for applications where the high update frequency of and interaction speed with the VNC server is desired, the value should be set high or the blocking flag in the [vRT_VNCWindowUpdate](#) function should be set. The priority is set to 20 by default, which will give high interaction speed with the environment and a fast VNC window update frequency for VRT applications with a light CPU load. For heavier processes the priority might have to be raised, at the cost of interaction speed.

Parameters:

win A pointer to the [vRT_VNCWindow](#) to be affected.

priority The new priority value.

See also:[vRT_VNCWindowUpdate](#)

Returns:

1 on success, a VRT_VNC_ERR code otherwise.

See also:

[vRT_VNCWindowUpdate](#)

B.2.2.4 int vRT_VNCWindowUpdate (vRT_VNCWindow * win, int blocking)

Sends an update request to the VNC server, if one is not already sent, and receives an update message if something has changed at the server desktop.

The function also performs an update of the texture in the local scene, if something has changed. The function can be run either in blocking or non-blocking mode depending on the variable *blocking*. In blocking mode the function will always receive a whole update message from the server, if one is sent, before returning the control to the calling application. In non-blocking mode the function will be forced to return the control even if the whole update message has not been received. How much data that will be received before returning the control is determined by the priority set by [vRT_VNCSetPriority](#).

Parameters:

win A pointer to the [vRT_VNCWindow](#) to be operated on.

blocking A value of 1 selects blocking mode, 0 selects non-blocking mode.

Returns:

1 if the texture has been updated, 0 if unfinished, non-blocking read and a VRT_VNC_ERR code if VNC server failed.

See also:

[vRT_VNCSetPriority](#)

B.2.2.5 int vRT_VNCWindowUpdatePartial (vRT_VNCWindow * win, int blocking, int x, int y, int w, int h)

Same as [vRT_VNCWindowUpdate](#), but requests only the part of the screen within the specified rectangle.

Parameters:

- win* A pointer to the [vRT_VNCWindow](#) to be operated on.
- blocking* A value of 1 selects blocking mode, 0 selects non-blocking mode.
- x* X position of upper left corner of requested rectangle.
- y* Y position of upper left corner of requested rectangle.
- w* Width of requested rectangle.
- h* Height of requested rectangle.

Returns:

1 if the texture has been updated, 0 if unfinished, non-blocking read and a VRT_VNC_ERR code if VNC server failed.

See also:

[vRT_VNCSetPriority](#)

B.2.2.6 int vRT_VNCHandleMouseEvent (VRT_Node * node, vRT_VNCWindow * win, int x, int y, unsigned int winmask)

Sends a message to the VNC server that the mouse has been moved or clicked.

By supplying the node the VNC window is attached to, with its transformation and projection matrices, it is possible to translate the 2D mouse position into 3D coordinates and then into 2D coordinates for the VNC window, that can have any orientation in the 3D space. All this is taken care of in this function by use of `vRT_PickTexelCoordinate`. To make the updates appear locally as soon as possible, this function should usually be followed by a call to `vRT_VNCWindowUpdate` with the blocking flag set or several calls without the blocking flag.

Parameters:

- node* The VRT node the VNC window is attached to.
- win* A pointer to the [vRT_VNCWindow](#) to be operated on.
- x* X coordinate of pointer in screen coordinates. Usually set to `LOWORD(msg.lParam)`.
- y* Y coordinate of pointer in screen coordinates. Usually set to `HIWORD(msg.lParam)`.

winmask Bitmask defining which mousekeys have been pressed. Usually set to msg.wParam.

Returns:

1 on success, a VRT_VNC_ERR code otherwise.

See also:

[vRT_VNCWindowUpdate](#)
[vRT_VNCHandleKeyEvent](#)

B.2.2.7 int vRT_VNCHandleKeyEvent (vRT_VNCWindow * win, int virtkey, DWORD keyData)

Sends a message to the VNC server that a keyboard key has been pressed or released.

Since Windows and the VNC server use different key encodings, and there is no obvious way to map some windows keys to VNC keycodes, this function becomes quite complicated. All characters consisting of only one keypress are easy to map, since there is a one to one mapping between them. Some characters however are more complicated. The @ symbol for example is represented as a single keycode in Windows, but has to be translated to the following sequence before sending to the VNC server: ctrl-down, alt-down, 2-down, 2-up, alt-up, ctrl-up. This is taken care of in this function by use of keymaps defined in keytab.h. To make the updates appear locally as soon as possible, this function should usually be followed by a call to vRT_VNCWindowUpdate with the blocking flag set or several calls without the blocking flag.

Parameters:

win A pointer to the [vRT_VNCWindow](#) to be operated on.

virtkey Windows virtual key code for the key pressed or released. Usually set to (int)msg.wParam.

keyData Extra Windows information telling for example if the key was pressed or released, if it was pressed before and so on. Usually set to (DWORD)msg.lParam.

Returns:

1 on success, a VRT_VNC_ERR code otherwise.

See also:

[vRT_VNCWindowUpdate](#)
[vRT_VNCHandleMouseEvent](#)

B.2.2.8 int vRT_VNCSendString (vRT_VNCWindow * win, char * str)

Sends an ascii string to the VNC server by repeated calls to vRT_VNCHandleKeyEvent.

For sending of special characters use vRT_VNCHandleKeyEvent directly. For sending a line feed terminated string use vRT_VNCSendStringR instead. To make the updates appear locally as soon as possible, this function should usually be followed by a call to vRT_VNCWindowUpdate with the blocking flag set or several calls without the blocking flag.

Parameters:

win A pointer to the [vRT_VNCWindow](#) to be operated on.
str The ascii string to be printed.

Returns:

1 on success, a VRT_VNC_ERR code otherwise.

See also:

[vRT_VNCSendStringR](#)
[vRT_VNCHandleKeyEvent](#)
[vRT_VNCWindowUpdate](#)

B.2.2.9 int vRT_VNCSendStringR (vRT_VNCWindow * win, char * str)

Sends an ascii string to the VNC server followed by a line feed (or return), otherwise the same as vRT_VNCSendString.

Parameters:

win A pointer to the [vRT_VNCWindow](#) to be operated on.
str The ascii string to be printed.

Returns:

1 on success, a VRT_VNC_ERR code otherwise.

See also:

[vRT_VNC_SendString](#)
[vRT_VNC_HandleKeyEvent](#)
[vRT_VNC_WindowUpdate](#)

B.2.2.10 int vRT_VNC_GetLastError (char * *error_message*, int *size*)

Returns the error code and error message for the last occurred VRT_VNC error.

The error message consists of a null terminated ascii string of at most *size* characters, written to *error_message*.

Parameters:

error_message VRT_VNC error message.

size Maximum size of error message.

Returns:

VRT_VNC error code.

C An example of a VRT VNC application

```

// #####
// ## vrt_vnc_test.cpp - A test application to show the use of a
// ## VNC client in VRT.
// ## By: Robin Stridh 2002
// #####

#include <windows.h>
#include <stdio.h>
#include <string.h>
#include <math.h>
#include <time.h>
#include <stdlib.h>
#include "vrt.h"
#include "vrt'util.h"
#include "vrt'vnc.h"

// Camera parameters
float fov = 45.0;
float camera_x = 0.0;
float camera_y = 0.0;
float camera_z = 10.0;
float camera_step = 0.5;

VRT_Node *screen;
vRT_VNCWindow *vnc;

void build_scene() {
    #define shading_model VRT_SM_FLAT
    VRT_Geometry *geom;
    // Server initialization takes server name, port number (minus 5900) and password.
    vnc = vRT_VNCWindowNew("hydra.hci.uu.se", 1, "hejhopp");
    geom = vnc->g_win;
    screen = VRT_NodeNew(vrtctx->root, "screen main node");
    VRT_SetTextureModulationMode(VRT_TEXTURE_MODULATION_DECAL);
    VRT_GeometrySetShadingModel(geom, VRT_SM_TEXTURE);
    VRT_NodeSetGeometry(screen, geom);
    VRT_NodeScale(screen, (float)(3 * 0.8), (float)(3 * 0.6), 1);
}

```

```
static VRT_HookPtr SimulationLoop(MSG msg) {
    int camera_moved = 0;

    // The update function is always run. If second parameter is 1 the function
    // blocks until the whole screen is recieved. If it is 0 it will only read
    // a small piece of the update and return. When the whole update message has
    // been recieved the screen will be updated automatically.
    vRT_VNCWindowUpdate(vnc, 0);

    switch (msg.message) {

        // Mouse actions are sent directly to the server.
        case WM_LBUTTONDOWN:
        case WM_LBUTTONUP:
        case WM_MBUTTONDOWN:
        case WM_MBUTTONUP:
        case WM_RBUTTONDOWN:
        case WM_RBUTTONUP:
        case WM_MOUSEMOVE:
            vRT_VNCHandleMouseEvent(screen, vnc, LOWORD(msg.lParam),
                                   HIWORD(msg.lParam), msg.wParam);

            break;

        // Keyboard actions are checked for arrows and page up/down, which will
        // move the camera. All recognized keyboard actions are sent to the server.
        case WM_CHAR:
        case WM_KEYDOWN:
            camera_moved = 1; // assume camera has moved
            switch (msg.wParam) {
                case VK_UP:
                    camera_y += camera_step; break;
                case VK_DOWN:
                    camera_y -= camera_step; break;
                case VK_LEFT:
                    camera_x -= camera_step; break;
                case VK_RIGHT:
                    camera_x += camera_step; break;
                case VK_PRIOR:
                    camera_z -= camera_step; break;
                case VK_NEXT:
                    camera_z += camera_step; break;
                default:
                    camera_moved = 0;
            }
    }
}
```

```
    if (camera_moved) {
        VRT_SetDefaultCamera(camera_x, camera_y, camera_z,
                            0,0,0, 0,1,0);
    }
    vRT_VNCHandleKeyEvent(vnc, msg.message, msg.wParam);
    default:
        break;
}
return 0;
}

int WINAPI WinMain (HINSTANCE hInstance,
                   HINSTANCE hPrevInstance,
                   LPSTR lpCmdLine, int nCmdShow)
{
    if (hPrevInstance) return 0;
    VRT_Init(hInstance);
    VRT_SetDisplay(hInstance, VRT_DEFAULT);
    build_scene();
    VRT_SetCallback((VRT_HookPtr)SimulationLoop);
    VRT_SetClearColor(0.7f,0.7f,8.0f,0.5f);
    VRT_SetDefaultCamera(camera_x, camera_y, camera_z,
                        0.0,0.0,0.0,0.0,1.0,0.0);
    VRT_SimulationLoop();
    VRT_Close();
    return TRUE;
}
```

D Pixel formats

The VNC server uses a different pixel format than for example OpenGL and Windows as can be seen in table 1. The pixel data therefore has to be converted to the appropriate format before it can be used in a 3D application. The VNC server also use different colour encodings dependent of the server settings.

Colour format:	lowest address ... highest address
vncserver Windows:	<code>0x00rrggbb</code>
vncserver UNIX:	<code>0xbbggrr00</code>
Windows COLORREF:	<code>0xααbbggrr</code>
OpenGL RGBA (and VRT):	<code>0xααbbggrr</code>

Table 1: *Some different pixel formats.*

Since the alpha channel is not used by the VNC server, it is set to 255, which makes the screen fully opaque. The red, green and blue values are reorganized by a simple macro to be useful in OpenGL and Windows.

E VNC Server Usage

The recommended use of vncserver on X systems is:

```
vncserver -depth 32 -pixelformat RGB888 -geometry 1024x768
```

Typeset in L^AT_EX.