NTNU
Innovation and Creativity

Realtime capture and streaming of gameplay experiences

Kristian Amlie

Master of Science in Computer Science
Submission date: July 2007
Supervisor: Richard E. Blake, IDI
Co-supervisor: Ole-Ivar Holthe, ITEM
Ole-Ivar Holthe, Gridmedia Technologies AS

Norwegian University of Science and Technology
Department of Computer and Information Science

Problem Description

Geelix is a new online service for sharing of gaming experiences. With Geelix, one can navigate and play large amounts of game media content, like trailers, gameplay and more. The goal of the project is to extend Geelix with advanced methods for realtime capture and streaming of video and data in games. The main focus will be on functional extensions of Geelix for realtime compression of video, a streaming system, and analysis of performance. The project includes both design and implementation, and requires high proficiency in both C++ and Java.

Assignment given: 20. January 2007
Supervisor: Richard E. Blake, IDI

Abstract

Today's games are social on a level that could only be imagined before. With modern games putting a stronger emphasis on social networking than ever before, the identity in the game often becomes on par with the one in real life. Yet many games lack the really strong networking tools, especially when networking regarding players of different games is concerned.

Geelix is a project which tries to enhance the social gaming aspect by providing sharing of what has been aptly named: gaming experiences. The motivation for this goal is to enable stronger support for letting friends take part in your online, or even offline, experiences. It is the belief that sharing gaming experiences is a key element in building strong social networks in games. This master thesis was written in relation to the Geelix project, where the focus was on enhancing the Geelix live sharing experience with advanced methods for video compression and streaming.

Contents

Chapter 1

Introduction

1.1 Background

The author of this master thesis, Kristian Amlie, is a Computer Science student at the Norwegian University of Science and Technology (NTNU). The thesis was written as part of a student exchange at the University of California, Berkeley, in collaboration with Gridmedia Technologies AS. The thesis was written using the LaTeX typesetting system, and the figures were made using OpenOffice.org.

Gridmedia Technologies AS is a software company started in Norway in 1996, and has since acquired offices in San Francisco in California, USA, currently with two employees in addition to the students. The company is led by PhD. student Ole-Ivar Holthe, who has also been the author's supervisor during the project.

The project which the thesis is based on has been a part of the bigger project which is the whole of Geelix. Besides working with Gridmedia, the author also collaborated with Richard Tingstad, a fellow student from NTNU. His project is called "Recording and publishing of gameplay experiences", which touches upon some of the same academic fields.

1.2 Due thanks

The author wishes to thank his supervisor, Ole-Ivar Holthe for allowing us to come to the US and work on such a unique project, as well as Richard Tingstad for his cooperation and discussion input. A thanks should also be directed to professor John Canny for his help with everything related to UC Berkeley.

I also wish to thank my family for supporting me in my decision to spend a whole year so far away. I miss you all!

1.3 Document structure

This document is divided into nine chapters:

Chapter 1 contains the introduction to the author and the involved parties.

Chapter 2 talks about the motivation for setting the project goals of Geelix, and gives an overview over Geelix's already existing software and the weaknesses that inspired the initiation of this project.

Chapter 3 will go through the preliminary study, where related work and technologies will be discussed.

Chapter 4 lists all the requirements for the new Geelix system.

Chapter 5 describes the specification of the new system design, as well as an explanation of how it solves the requirements.

Chapter 6 documents the implemented system and focuses on how it differs from the design.

Chapter 7 shows the results of the system and performance tests that were performed on the implemented system, and discusses their implication for Geelix.

Chapter 8 talks about possible future directions for Geelix.

Chapter 9 gives an evaluation of the different experiences with the Geelix project.

Chapter 2

Motivation

This chapter will give an overview of the background that is the basis for this project, as well as the overall goals for the system, called Geelix. It will then move on to describing the old system, how it works and how it tries to address the goals of the system. Afterwards it goes into addressing the weaknesses of the old system, and how the desired system should function in order to avoid or reduce these symptoms, and to improve the fulfillment of its goals.

2.1 Background and goal

The overall goal of Geelix is to be a system for sharing gaming experiences. The motivation for setting this goal is to enable players to take part in other players' experiences, whether the purpose is for learning, entertainment or teamplay.

To understand why this goal is important, it is crucial to realize that social networks form a powerful component of the gaming experience [1], especially in multiplayer games, and that the offline ties between players also serve an important part in the game enjoyment. Common to any community revolving around a common interest, is the desire to show off achieved feats or artifacts to other members of the community. An example can be to kill a very difficult monster using a cleverly devised method, and then showing off the deed to your friends. Some games do contain measures for recording such events and replaying them later, but in general, the support for this type of action is low.

Jakobsson and Taylor also argue that players of massively multiplayer online games only achieve their greatest individual results by working together. Working together requires a good deal of communication, and language based

communication is the most common way of achieving this [2]. Even though most multiplayer games today feature some sort of language based communication, it is often based on textual messages, which is a slow method of exchanging words compared to the spoken word. According to [3], certain games such as war games can benefit greatly from having users communicating by using spoken language and voice over IP. In other games, they claim that communication has no direct impact on game performance, but still adds to the overall enjoyment of the game.

In addition, there are several other types of communication that do not involve language: For instance, spatial behavior as well as posture communicates a lot about the intended action of a player, and often much quicker than any language based communication can achieve [4]. The degree to which games provide these cues is variable, but seems to improve with newer games. Still, we think that most games leave much to be desired in this area, even if cues are well supported on avatars within the game. For instance, a lot of strategy and tactics depend on the situation of other teammates. When your teammates are out of direct view, it is often hard to tell what their situation is, no matter how well their avatar is animated when looking at him/her directly. Some games try to solve this by using automated audio to report status about teammates, as well as radar maps that inform you of their position. However all of these means are restricted to the rules and mechanics of the individual games, and may vary greatly.

Looking at the discussed concepts together (event recording, efficient communication and situation awareness), they can be very useful to players learning how to play the game, both novice and advanced. Language communication is an important part of the process of learning the game [3], as is learning by watching others perform the action in question [2]. By also adding event recording, players have the ability to study the experience also after it has happened, giving further opportunities for learning.

Geelix's goal is to try to enhance these concepts in games by enabling players to share their gaming experiences in a variety of ways. The goal is to improve communication and situation awareness by using methods to take part in another player's experiences in real time, as well as providing means for recording and archiving experiences for later consumption, whether it be for learning, personal enjoyment or boosting community status.

2.2 Old system

In the old Geelix system, sharing a gaming experience is defined primarily as sharing a gameplay video. The system offers several ways of sharing these videos, which will be described shortly.

The system consists of three main components: An in-game Heads Up Display (Geelix HUD), a Web portal and a desktop application (Geelix Desktop GX). Geelix HUD is an application that shows up inside the users' games and gives them the opportunity to communicate and share their experiences. The Web portal provides users with the means to search for gameplay videos uploaded by other users. And lastly, Geelix Desktop GX acts as an intermediary between the portal and the HUD, giving users the ability to register their games for use with the HUD, as well as upload their own videos to the portal.

2.2.1 Geelix Desktop GX

A screenshot of Geelix Desktop GX can be seen in figure 2.1. The "Add Game" button allows you to add a game to the list of games seen in the figure. As long as the game remains in the list, Geelix HUD will show up inside the game when playing. The game can be removed using the menu, if desired.

The "Add Game" window that can be seen, is used to search for game titles in order to make the Geelix system aware of exactly which game you have.

The "Record" button simply gives some guidelines on how to record videos; the actual recording takes place within Geelix HUD. When a video has been recorded there, it will show up under the "My Recordings" tab, similar to the games in the "My Games" tab. From here one can decide to upload the video using the "Upload" button, in which case a form will show up where the user can fill out comments on what is displayed in the video, such as which game stage is being displayed, difficulty of the performed action, etc. Because the video recorded by the HUD is not compressed at all, the video is reencoded into a compressed format (WMV) before being uploaded. The progress of the encoding and upload can be seen under the "My Uploads" tab.

2.2.2 Geelix HUD

The Geelix HUD is an application that does not run on its own, but hooks into the game and provides the user with an interface inside the game. There

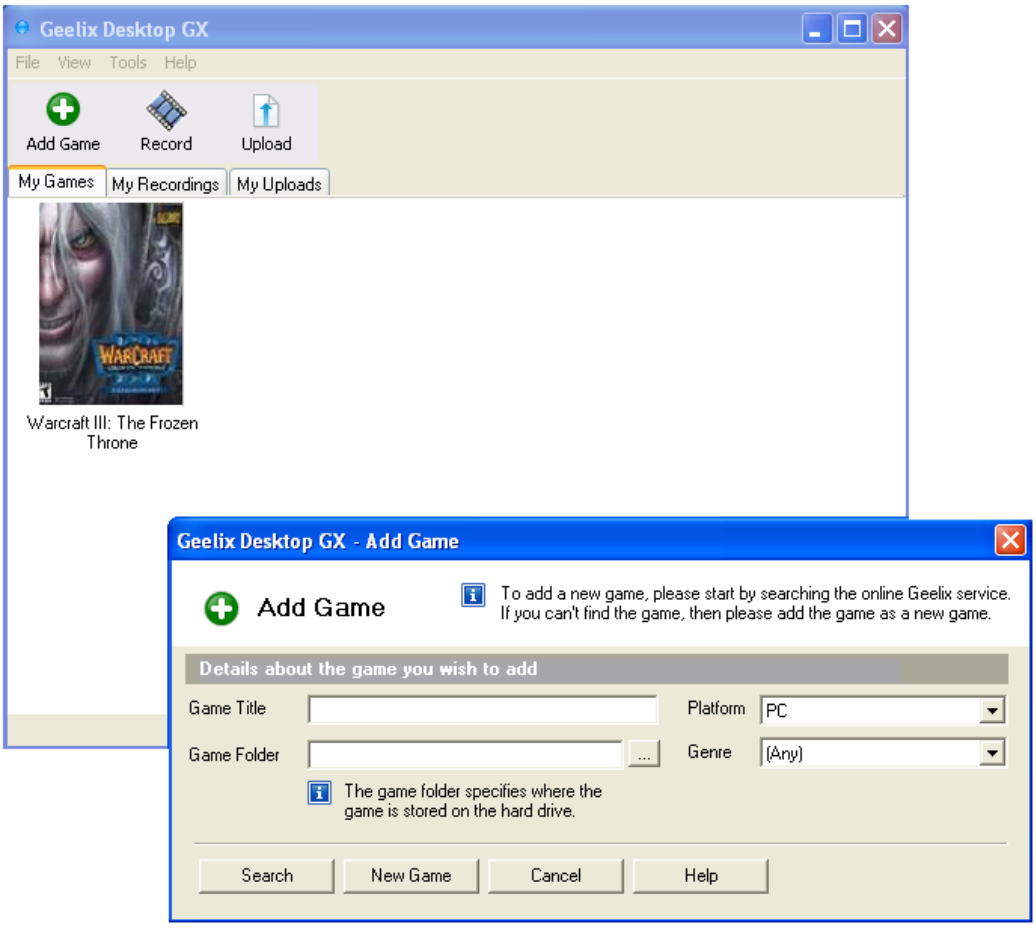


Figure 2.1: Geelix Desktop GX

are three main services being offered by the HUD: Chatting, recording and live video sharing.

Chatting is realized by having a list of contacts in the interface, and choosing people to talk to. It highly resembles instant messaging services, such as MSN [5], with a buddy list, chat windows, and the ability to talk to more than one person at a time.

Recording means to capture video from the current gameplay. The resulting video file is stored on disk, will show up in Geelix Desktop GX and can be uploaded to the portal.

Live video sharing is the most interesting feature, and it allows you to record video from the game and share with another user of Geelix, in real time. This is realized in the same way as chat. For instance, if Jack wants to view Jill, he clicks on her name in his buddy list and then selects the view

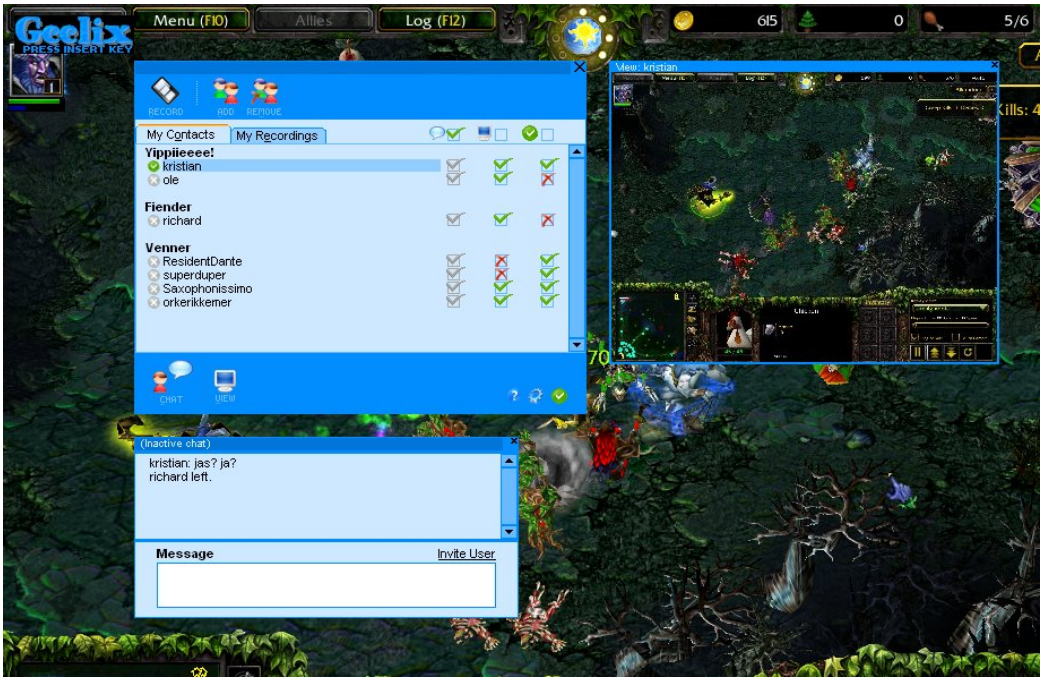


Figure 2.2: Geelix HUD used in Warcraft III

button, which will start a video stream from Jill to Jack. If Jill doesn't want Jack to view her gameplay, she can indicate this by clicking the permission tick boxes to the right of Jack's name.

The live video sharing is generally of much lower quality than recording to disk.

Figure 2.2 shows Geelix HUD being used in the game Warcraft III. The largest blue window is the main interface and has the contact list. From here, a user can initiate all the mentioned actions. The smaller blue window is a chatting window, which works in the same way as other instant messaging services. The last window contains the live view of another user's screen, in real time.

When the HUD is not needed, it can be hidden instantaneously by a single keystroke. It is also possible to hide individual windows, so as to only keep chat open, for example.

When this document talks about "users", "players" or "gamers" it generally refers to a person that is logged into the Geelix HUD.

Figure 2.3: Web portal

2.2.3 Web portal

The Web portal is a central repository of user submitted gameplay videos, and gives visitors the opportunity to search, browse and play these videos. A screenshot of the portal can be seen in figure 2.3. The portal is not unlike YouTube [6] in design, but is focused on games. For instance, you can go to the "Games" section of the portal to browse or search for specific games. When a game has been chosen, the portal displays videos that are exclusively related to that particular game.

The portal offers many ways to browse through video material. Searches can be performed by using standard search terms (like Google) or by category, or both together. In figure 2.4, the user searched for the word "intro", and then afterwards narrowed down the search further by choosing gameplay videos (as opposed to trailers) and the adventure genre from the menu on the left.

Figure 2.4: Video search using both terms and categories

The portal is also where users can create and manage their profile. The management tools offer these main services:

- Adding and removing contacts from your profile. Contacts represent other users of Geelix, and having them as a contact allows you to chat with them and see their gameplay.
- Sending asynchronous messages to other users. When users are not available for direct chatting, the portal can be used to leave an asynchronous message that can be read at a later time, similar to an email service.
- Manage uploaded media. Videos can be published for others to see, or removed from public view, and one can also change the name and characteristics of the video, such as which game it displays.
- Manage profile details. The portal allows you to change your profile details, such as email address, description, gender and birth date.

The last main use of the portal is seeing live gameplay of other users. The portal offers a list of users that are currently sharing their HUD, and by clicking on one of the names, the user is able to see the live gameplay of that user without being logged into the HUD him-/herself.

2.3 Problems

Geelix has chosen video as the definition of a gaming experience, and it uses this to enhance communication and situation awareness between players. Video is also used as the medium for storing gaming experiences. In this project, the focus will be on the live exchange of gaming experiences.

The old live video solution has many problems. It has bad performance, meaning that it has a very low framerate and therefore is not very visually pleasing. The system also does not scale well, which is important if Geelix is ever to support a high number of users. These problems stem mostly from the technical design of the streaming solution, which was meant as a proof-of-concept rather than an industry strong, scalable technology.

The old solution also has no way of transferring audio between clients. Audio would be very useful both for voice communication as well as hearing what is happening in another user's game.

2.4 Desired system

The desired system will have a live screen sharing service that is reliable, scalable and visually pleasing to the user. It should also be possible to tailor the application to specific user needs, such as users with low net bandwidth or low computer processing power. In addition it should be possible to tailor the service for each game. For instance, a fast paced shooter game will require a fast, smooth, and low delay video stream, where detail may not be so important. A strategy game however, requires a lot of detail in order to see the numerous screen elements, whereas the framerate may be slightly less important.

The solution should also include voice support, both for communication between users using a microphone, and for in-game sounds. The solution should focus on achieving low latency and low CPU processing cost.

The system should also support the transmission of user input data from one user to another, such as keyboard, mouse and joystick input. Dyck *et al.* [7] argue that difficult input sequences can be hard for the observer to catch unless the demonstrator is very explicit and deliberate in his actions. We believe that this can aid in the learning of game skills, and more closely resemble the situation of "looking over someone's shoulder" to learn game tricks.

Chapter 3

Preliminary study

This chapter will present the results of the work and analysis that was performed prior to starting the design of the new system. It starts by going through some of the related work in the field, and then moves on to more general platforms that deal with the concept of video streaming. Afterwards, we discuss technology that is relevant to the project, and then end the chapter with an analysis and a conclusion on the decision of which platform and technology to use.

3.1 Related work

At the time of writing, the author is not aware of any research that deals explicitly with video sharing in games. The research that discusses how sharing of experiences can improve learning and strategic aspects of gaming ([2, 4], discussed in section 2.1 on page 7) deal primarily with the design of new games, while we try to attack the issue from a more general perspective, covering all games, existing and new.

The commercial sector does have some interesting applications that overlap with our work, but they are not trying to solve the same problem as we are:

Playlinc is a multifaceted gaming tool that heavily focuses on the integration of communication, gameplay and community [8]. Like Geelix, they have taken a more general approach, focusing on gaming as a whole instead of any specific subgame or subgenre. They provide an in-game interface and include support for inviting your friends to games and conversations, as well as voice over IP. However, they provide no means to share experiences directly, neither through video nor any other medium.

Xfire is an application quite similar to Playlinc that provides an in-game interface and means to chat with your friends using text and voice over IP [9]. They also provide game server browsing and seeing which servers your friends are playing on.

Game Overlay is an application that tries to regain some of the lost multitasking capabilities when playing games [10]. The way it works is by rendering regular application windows on top of the game, enabling users to take advantage of any application they normally use outside of the gaming environment. Sharing gaming experiences thus becomes possible by using third party tools, for example VNC (see section 3.2), but since they are not trying to solve the same problem as Geelix, this solution is suboptimal and quite slow.

At the time of writing, none of these applications provide the kind of experience sharing that Geelix wants to support.

3.2 Existing platforms

In this section, some existing work will be discussed that is relevant to us in the design of the system. From a technical viewpoint, the live screen sharing service is simply an audio/video stream going from one user to another. This is not a new scenario and many solutions already exist, each with their own advantages and disadvantages.

At first, even though no cameras are involved, the service might resemble that of videoconferencing, where audio and video are sent between users to form a communication link between them. However, Geelix can also be used to stream from only one user to a large number of viewers (for example in a tournament), which makes the system more similar to a live broadcast streaming service. Because the Geelix system is a sort of hybrid between these technologies, many existing solutions will be examined.

Windows Media Services is a streaming server for the Windows Server 2003 platform [11]. It focuses mainly on streaming the Windows Media formats, WMV and WMA. To use the server, it is required to purchase a license.

Darwin Streaming Server is an open source streaming server, based on the code from Apple's QuickTime Streaming Server [12]. The server uses the RTSP protocol to deliver QuickTime, MPEG-4 or 3GPP streams to clients. The server is cross platform.

Helix is a streaming server from Real Networks. It has a wide range of supported streaming formats, such as RealAudio, RealVideo, Windows Media, QuickTime, MP3, MPEG-4, 3GPP (H.263 and H.264) [13]. The server requires purchasing a license to use.

VideoLAN is actually a media player [14], but supports streaming of many formats to clients [15]. It is also cross platform.

VNC stands for Virtual Network Computing, and is a cross platform application for viewing and interacting with computer desktops across the Internet [16]. It does not use video streaming, but instead its own protocol which is based on sending only the parts of the screen buffer that are updated.

Asterisk is an open source, cross platform VoIP server [17]. It is licensed under GPL (GNU General Public License) and can be downloaded and used free of charge. It provides both audio and video conferencing using the SIP (Session Initiation Protocol) and H.323 protocol.

3.3 Technology

In this section we go through some of the technologies that are relevant for the implementation of live screen sharing. First we go through some of the streaming technologies available and then move on to talking about compression codecs.

3.3.1 Streaming

Streaming is the way by which multimedia content is transported from one endpoint to another, and the protocols are generally found in the application layer of the OSI network model. There are several protocols available for streaming both video and sound.

HTTP

HTTP stands for HyperText Transfer Protocol and was originally designed to transfer web pages consisting of text over the Internet. In time, however, it evolved into an advanced data transfer protocol capable of handling any data type. When a client wants to stream media, it requests it using a URL and then receives the stream. Traditionally, HTTP has been used as a "download first, then watch" protocol, but many new media players start playing the media before it has been fully downloaded.

RTP / RTCP

RTP stands for Real-time Transport Protocol and RTCP stands for RTP Control Protocol. RTP is a protocol designed to transport real-time data, such as voice and video conference calls [18]. It is based on UDP, which is more desirable for real-time protocols than TCP. UDP guarantees packet integrity, but not stream integrity [19]. This means that an arriving packet is guaranteed to be intact, but if a packet is lost or duplicated, no attempt will be made to recover from the anomaly. This is good for real-time applications, where a late packet typically is just as bad as a lost packet.

RTP is augmented by the RTCP protocol, which is a TCP-based protocol running alongside RTP. It carries no application data, but instead reports statistics about the quality of service (QoS) of the RTP transport. This can be used by the application to adjust the bitrate of the stream, for example if many packet losses are observed.

RTSP

RTSP stands for Real Time Streaming Protocol, and is not a streaming protocol by itself. It is a TCP-based protocol meant to be used alongside RTP / RTCP as a control protocol, providing stream control operations such as pausing, seeking and fast forward [20]. The RTP stream reacts to messages sent through the RTSP channel.

MMS

MMS stands for Microsoft Media Services and is a proprietary streaming protocol that can use both UDP and TCP, depending on the conditions. MMS is deprecated by Microsoft in new versions of their Media Services, in favor of RTP / RTSP [21], and will not be considered further in this project.

SIP / H.323

SIP (Session Initiation Protocol) and H.323 are both network protocols for audio and video communication over packet based networks, such as the Internet [22, 23]. SIP is used as a negotiation protocol to establish, configure and tear down communication sessions. A transport protocol, such as RTP, is needed to to transport audio and video between participants.

H.323 is a fully featured multimedia communications protocol, providing session initiation and negotiation, as well as multimedia transport. The protocol supports using so called gateways and gatekeepers, which are intermediary servers, in order to bridge networks together.

3.3.2 Codecs

Codecs (COmpressor / DECompressor) are used to compress data before it is transmitted in order to conserve network bandwidth. This section looks at some of the available compression codecs available, for both audio and video. We will only consider lossy codecs in this section (codecs that distort the original image slightly), because of their superior compression ratio compared to lossless codecs.

H.263 / H.263v2

H.263 is a video codec standard defined by the Video Coding Experts Group (VCEG) [24]. It is built upon the H.261 standard, which was defined by the same group, and is designed to improve its performance at all bitrates. The codec has the ability to operate in one of many modes, where each mode has encoding parameters optimized for a given scenario, such as low latency for conversations or high latency for efficient throughput. Each coding parameter, which are called annexes, can also be specified explicitly for special applications.

H.263v2 is sometimes called H.263+ and is the second version of the H.263 codec, which adds both new modes and annexes. Encoders and decoders are not required to support all annexes, but must support the baseline encoding, without annexes.

H.264

The H.264 codec was developed as the next major version of the H.263 standard, focusing on improving the compression efficiency of the previous codec [25]. The functional elements of the compression scheme are very similar to that of H.263, but the main differences lie in the details of their operation.

FLV / Sorenson Spark / On2 VP6

FLV (Flash Video) is a proprietary codec and file format used by Adobe's Flash Player (formerly owned by Macromedia). The codec is sometimes called Sorenson Spark and is based on the H.263 codec, but is not compatible with it. Newer versions of the FLV format (available from Flash Player 8 and onward) use the On2 VP6 codec, which is a big quality improvement over Sorenson Spark [26].

MPEG4

MPEG4 is a codec developed by MPEG (Moving Picture Experts Group). It aims to provide a flexible codec for use in many different situations, including low bitrate networks [27].

WMV

WMV (Windows Media Video) is Microsoft's proprietary video codec, and is based on MPEG4. It comes in three different versions, version 7, 8 and 9, where version 9 is the latest one. The short names for these codecs are (somewhat confusingly) WMV1, WMV2 and WMV3, respectively. WMV3 has been formally published by the Society of Motion Picture and Television Engineers (SMPTE), and is called VC-1 [28].

MP3

MP3 is a lossy compression codec for audio, developed by scientists at the Fraunhofer Institute, and specified by MPEG in 1993 [29]. It is widely used in both the commercial and private sector.

3.4 Analysis

This section goes through the analysis of the different existing platforms, as well as the technologies involved with the chosen platform. A few project requirements will be brought up, which are described in detail in chapter 4.

3.4.1 Existing platforms

A lot of time was spent on doing research on the different streaming platforms available. Even though live testing of servers would have been ideal for determining their properties, this was deemed a too time consuming task to be viable, especially considering the number of servers we had to test.

For this reason, we chose to rely on doing research on the topic on the Internet, and look for other people's experiences with the various solutions. The analysis presented here is therefore not based on cold, hard facts, but rather on the various impressions we got from exploring people's experiences with the software. We chose to disregard all information coming directly from the companies that produce the software (or affiliated companies), since they tend to be biased and downplay any disadvantages the systems may have.

Because Windows Media Services and VNC was already installed on one of our computers, we did a few informal tests on them to back up our readings, but it was not an authoritative test.

The three first servers, Windows Media Services, Darwin, and Helix all seem to have one problem in common, namely latency. Because they are streaming servers and not conferencing servers, they are optimized for high throughput and reliable, uninterrupted playback. Thus, they typically include a lot of buffers in many of the stages, which gives a very noticeable delay from the video frame is transmitted until it is received by the viewer, often several seconds. Even live streams typically have a few seconds of delay, which is acceptable for a football game transmission, but not for sharing game screens, where up-to-date information is very important. Testing Windows Media Services only confirmed our readings, sometimes with latencies of up to fifteen seconds.

In addition to latency problems, Windows Media Services and Helix are commercial solutions, and are therefore not free, which acts as further discouragement for their use.

VideoLAN has the same problems regarding latency as the rest of the streaming servers, and in addition, it is a desktop application, which makes it unsuitable for scalability and server deployment.

VNC does not have the latency problems that the other servers have, as it is optimized for responsiveness. It is, however, also optimized for desktop use, meaning that it performs best on slow moving graphics surfaces, where typically only small areas change at a time. The application is designed to continuously poll the screen the buffer and only send the changes to the receiver. The compression algorithm only uses intraframe encoding (it doesn't use previous frames to reconstruct the current one), as well as a polling mechanism that seems to be called only at specific intervals. Testing the server with movie clips resulted in graphics updates where each frame did not even have time to render completely before a new frame started to appear, leading to several pieces of different frames on the viewing screen. Because of the inefficient polling and compression, VNC was deemed not suitable for this project.

Asterisk is probably the server that is best at fulfilling the needs of our application. It is an audio and video communications server (the core server is audio only, but plugins exist to add video capabilities) optimized for low delay transmission. It has a large set of features, ranging from call specific functions (call waiting, call ID), voicemail and SMS messaging, to support for several protocols, such as H.323 and SIP. It also provides programming APIs for many of these. While most of the call specific features are not important to the Geelix project, its core functionality, which is to let users

communicate using audio and video with low latency, is exactly what Geelix needs.

However, it was decided to not use Asterisk, even though functionally it seems to fit quite well for the purpose. There are several reasons for this, but the main reason is its license, the GPL. The GPL is an open source license that requires all applications that utilize the library to also be placed under the GPL license [30], although it is not required that you release the application to the public. This means that if Gridmedia uses the library, they do not have to release the application, but if they do, they must do so according to the GPL license. The Gridmedia management is not interested in making their projects open source at this time, and for this reason, the GPL license, which is outside their control, is best avoided.

In addition it is not clear how well Asterisk will support Geelix's goal of transmitting user generated input data between users, nor how well it will integrate with the existing Geelix system. Because of these reasons, Asterisk was deemed not suitable for this project.

Please note that there are several other open source alternatives to Asterisk (OpenSER, Yate, etc.), but they are also licensed under GPL.

For the reasons described above, it was decided to implement our own streaming platform, focused entirely on Geelix's needs.

3.4.2 Technologies

For the custom platform we needed a codec and a network streaming protocol. HTTP was quickly ruled out because it is based on TCP which doesn't have as good realtime qualities as UDP. SIP and H.323 both need one of the big server libraries that were deemed inappropriate, so RTP and UDP were considered instead. UDP is a low level packet system in the operating system and has good realtime qualities. RTP is based on UDP, and has QoS monitoring built in. It therefore stands out as a good network protocol for realtime applications. It also has a framework available, called ccRTP.

It was discovered during early testing however, that compiling ccRTP with Visual C++ 6.0 was troublesome. Even though QoS monitoring would have been a useful feature for the live screen sharing, it is not crucial. We therefore decided to drop RTP support in order to save time, and instead base our protocol entirely on UDP. By making out own protocol based on UDP, we also have the added advantage of ensuring that extra data, such as user input data, can be easily embedded in the protocol.

When choosing a video codec for the application, we wanted to do a benchmark of many different codecs in order to determine their various advantages and disadvantages. The best option was therefore to find an encoder

library that is capable of handling many different codecs. The library chosen for this purpose was FFmpeg, which is an open source video compression library [31]. The reasons for choosing this library was:

- The author already had some experience with the library
- It has a wide range of supported codecs
- It is highly configurable and tunable
- The library can be recompiled with only the chosen codec and all the others stripped, making the library very compact and well suited for implementation (this helps to fulfill nonfunctional requirement 3, described on page 29)
- As long as some specific GPL features are not compiled in, the library is licensed under LGPL, which permits use by proprietary programs [32]

A lot of FFmpeg's codecs were tested (more than 20, most of which are variations of the codecs described in section 3.3.2 on page 19), and many were completely unsuited for the task. A strong emphasis was put on the relationship between CPU workload and file size, where both should be kept low, while picture quality was considered less important. This helps to fulfill nonfunctional requirement 1, described on page 29.

Typically, when encoding in a production environment, most attention is given to the file size. It does not matter if the CPU has to work a little extra if the video is not being distributed in real time, or if the encoding process has an entire dedicated CPU. For this reason, we believed that many of the codecs had been optimized for file size and compression quality, and not so much for CPU workload. This however, was not generally the case, as the factors seemed to be relatively well balanced already. Any tweaking we did on any of the many encoding parameters almost always resulted in either higher workload or lower picture quality. The only exception was the motion vector search radius, which reduced workload, but the effects on picture quality were so severe that the result was unusable. After much tweaking of parameters, we decided to leave them all at their defaults for the benchmarking test.

The test was performed by reencoding a 60 second movie clip of the game "The Shield", into the given format, with the same bitrate for all codecs. Because each run consisted of both decoding the original movie clip and encoding the new file, the original file was decoded on its own without doing any encoding, and the time spent was subtracted from all the decoding

/ encoding results. This way we produce a number that is based on the encoding time only.

Table 3.1 lists the benchmark results of the most important codecs. The codecs that were left out had too large values, but MJPEG (codec that just stores JPEG images in sequence) was included because it closely represents the old situation (see section 5.1 on page 30 for a discussion of the old architecture). The percentages are the relative percentage compared to the smallest factor for that category (e.g. every size percentage is relative to WMV2, because it has the smallest file size).

Codec	**Time**	**Time%**	**Size**	**Size%**
FLV	5.09	7.8%	2374352	18.1%
H.263	4.97	5.3%	2374698	18.1%
MJPEG	4.72	0.0%	12882926	540.9%
MPEG4	5.77	22.3%	2028790	0.9%
WMV2	5.76	22.1%	2010218	0.0%

Table 3.1: Codec benchmarks

There is a clear tradeoff between time and file size among the top two and bottom two codecs, respectively. In the end, it was decided to prefer size over time and use the WMV2 codec. The argument (although not empirically proven) was that gamers typically have high end computers available because the games require it, while network connection is often based more on your location (people living outside a city may not be able to get broadband Internet at all). We thus wanted to provide a codec usable by as broad an audience as possible. The decision to use WMV2 was also backed up by the choice of the Web portal player, described in the next paragraphs.

The old Geelix system allows users to view other people's gameplay using the HUD tab on the web portal (see section 2.2.3 on page 12). The implementation uses a very simple solution where JPEG images are updated on a periodic basis to simulate a framerate. This solution has several drawbacks, such as low scalability and low visual quality (see section 5.1 on page 30 for details). In the new implementation, this will not be good enough, and a movie plugin is required. Windows Media Player and Flash Player were the candidates considered, and there was a considerable debate over which player to use. While there is no definite answer to this question, here are some of the important arguments for and against each player:

- Flash Player is used by many of the recent successful video services

(Google Video and YouTube, for instance), which proves its strength in the market.

- Flash Player gives you the opportunity to overlay objects over the movie clips, which could be important for Geelix's visualization of input data between users, and could also be used for watermarking movies to prevent plagiarism.
- The Flash Player can be customized so that its player theme matches that of the site.
- Windows Media (WMV) format files are smaller than their FLV counterparts, used by Flash.
- Gridmedia had more experience with the Windows Media format and was already using the format for their existing movie clips.

The author argued that choosing Flash was a choice more focused on the future, and easier to adapt using themes and object overlays. In the end the Gridmedia management decided on Windows Media Player, mostly because of the last argument. Because this player uses the WMV2 codec, it also confirms our choice to use that very codec.

Because audio makes up a very small portion of the bitstream, the MP3 codec, which is a well known and stable codec, was chosen for audio compression without doing any benchmark testing. The MP3 codec is also supported by both FFmpeg and Windows Media Player.

3.5 Conclusion

In this chapter, we have gone through the work that already exists in the fields that the project touches, as well as the streaming and codec technologies that the project involves. We have also looked at the analysis of the different existing solutions and technologies, and presented the libraries that implement the technologies chosen for the project.

All of the existing platforms were chosen not to be used in the project, either because of unsuitability or license problems, and it was decided to build our own application from the ground up. Among the technologies, plain UDP was chosen as the transport mechanism, with our own protocol on top, and WMV2 and MP3 were chosen as compression codecs after a benchmarking test. FFmpeg was picked as the compression library.

Incorporating extra services such as input data is possible because the platform uses a custom transport protocol.

Chapter 4

Requirement specification

This chapter will go through the formal requirement specification for the project. Each requirement will have a priority associated with it, which is either high (H), medium (M) or low (L).

Throughout the specification, and also the remainder of the report, a user in general refers to a person that is logged on to the Geelix network using the HUD application. A viewing user refers to the one who is consuming a stream, and a sharing user refers to the one who produces it. Any one user can have both roles.

4.1 Functional requirements

This section goes through the functional requirements for the application, and they can be seen in tables 4.1 to 4.5 on pages 27–28.

Note that not all of the requirements are directly dependent on the author to implement. For example, functional requirement 4.4 (FREQ4.4) depends on the Geelix Desktop application to set the various tunings for different games, but it is not the author who is responsible for implementing Geelix Desktop. The fulfillment of such demands therefore depends on cooperating with the other developers, and making sure their modules work together with the author's application. In addition, no GUI elements shall be made by the author, with the exception of possible modifications to the viewing window[1]. Section 5.4 on page 51 describes how the author plans to fulfill each requirement.

Note that there is a difference between functional requirement 3.2 and 6.3.

[1]Gridmedia wanted to remain in control of the appearance of the HUD, so the author would ask them for a new element if it was needed. Since the author does not consider himself a good GUI designer, this was a natural work distribution.

Number	Description	Pri
FREQ1	A user must be able to observe the gameplay of another user in real time. See table 4.2 for subrequirements.	H
FREQ2	It must be possible for several users to be viewing the same user.	H
FREQ3	A shared video must be viewable on the Web portal in real time. See table 4.3 for subrequirements.	M
FREQ4	It must be possible to tune streaming parameters in order to adapt to different processing environments. See table 4.4 for subrequirements.	M
FREQ5	It must be possible for the viewer to see the key and mouse presses of the sharing user in real time.	L
FREQ6	It must be possible to allow/disallow requests to view a user's screen by setting permissions. See table 4.5 for subrequirements.	M
FREQ7	It must be possible for the sharing user to see in the GUI that he/she is sharing.	M
FREQ8	It must be possible for the sharing user to see who is watching him/her.	L

Table 4.1: Main functional requirements

Number	Description	Pri
FREQ1.1	It must be possible for a user to request viewing of another user.	H
FREQ1.2	The sharing user's HUD must generate the video stream by capturing screen contents.	H
FREQ1.3	The sharing user's HUD must generate the audio stream by capturing audio from a microphone or from the game sounds.	M
FREQ1.4	The viewer must be able to see the shared video in a HUD window inside the game.	H
FREQ1.5	It must be possible to stream audio only.	L
FREQ1.6	It must be possible to stream video only.	L

Table 4.2: Gameplay streaming requirements

Number	Description	Pri
FREQ3.1	When a HUD user is viewing another HUD user, it must be possible to view the shared video on the Web portal in real time.	M
FREQ3.2	It must be possible for a user to share his screen to the Web portal in real time even when no other HUD users are viewing.	L

Table 4.3: Web portal requirements

Number	Description	Pri
FREQ4.1	It must be possible to tune the resolution of the video.	M
FREQ4.2	It must be possible to tune frames per second of the video.	M
FREQ4.3	It must be possible to tune the bitrate of the video.	M
FREQ4.4	It must be possible to tune streaming parameters individually for each game, so that one game can run with different parameters than another without user intervention.	L

Table 4.4: Parameter tuning requirements

Number	Description	Pri
FREQ6.1	The sharing user must be able to allow or disallow specific users from viewing his/her screen.	M
FREQ6.2	The sharing user must be able to allow or disallow specific users from viewing his/her input data.	L
FREQ6.3	The sharing user must be able to turn viewing on the Web portal on and off.	M

Table 4.5: Permissions requirements

The first requirement refers to the ability to share your screen on the Web portal with no prior viewers. The second requirement refers to the ability to deny viewing of the user's screen, whether there are other HUD viewers or not. This makes sense for instance if teammates are watching each other in the HUD, they do not want to give the enemy an opportunity to cheat by spying on them on the Web portal.

Number	Description	Pri
NREQ1	The application must achieve a better compression ratio for the video stream than the old implementation (when using the same resolution and frames per second).	H
NREQ2	The application must be scalable to a large number of users.	H
NREQ3	The Geelix HUD compiled code size must be kept as low as possible.	H
NREQ4	A change in the permissions must be reflected immediately, by terminating access to viewers that no longer have permission.	L

Table 4.6: Nonfunctional requirements

4.2 Nonfunctional requirements

This section goes through the nonfunctional requirements for the application, and they can be seen in table 4.6.

NREQ3 deserves some explaining: Gridmedia wants to keep the size as low as possible because large file sizes discourage downloads by the users. What this means for the development, is that libraries and coding techniques that favor small compiled code size should be chosen over other methods, as long as it does not carry any other significant drawbacks. As an example, the C++ Standard Template Library (STL) is being specifically avoided in the HUD because of its potential for "code bloat" (excessively large compiled code size).

Chapter 5

Design

This chapter will describe the design of the Geelix real time live streaming application. It will begin by going through the old design, and then explaining what changes are required for the new one. Finally, a description will be given of how the new design fulfills the functional requirements.

5.1 Old architecture

This section goes through the old architecture of the Geelix system. This is important because the new design has to be based on this. Focus will be on the live sharing part of Geelix. The section will also describe the old code modules in the client.

5.1.1 Overview

The old implementation is based on the client/server model. The client is the HUD application, while the server is divided into two parts, a web server and a custom server called Geelix Live Server (GXLS). Both servers are backed by a common database. An illustration of the architecture can be found in figure 5.1.

As can be seen, the Geelix HUD communicates with both of these servers. GXLS functions as chat server for the HUD, and it also delivers other types of messages. As long as the HUD is logged on, it maintains a network socket connected to GXLS. The web server is used for any other request the HUD has, such as looking up information or storing something in the database. The connections to the web server are regular HTTP requests and are severed as soon as they are completed (as is common in HTTP).

The old live streaming implementation uses a very simple protocol, where

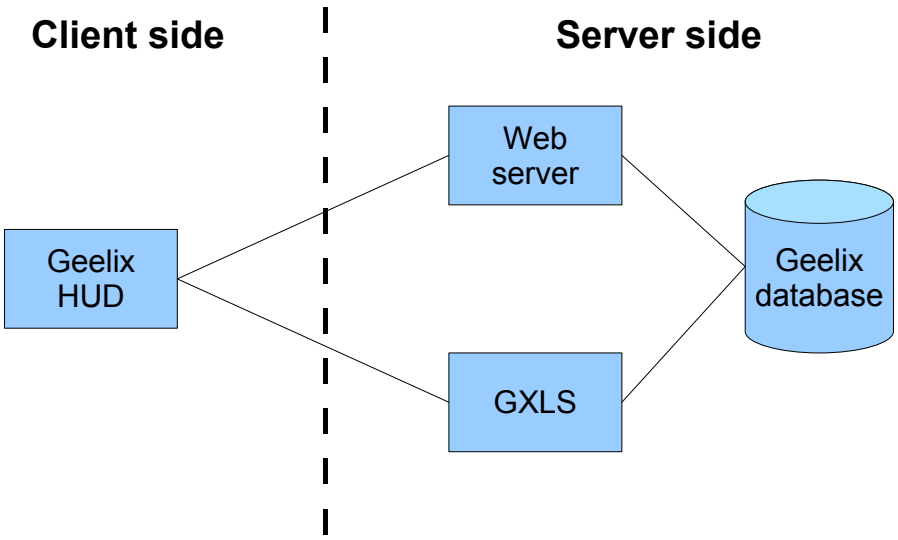


Figure 5.1: Old architecture

JPEG images are uploaded at given intervals to simulate a video stream. Below is a description of what happens when someone requests viewing someone else. We will call the viewer's HUD A, and the sharing user's HUD B. A sequence diagram for the operations can be seen in figure 5.2.

1. The user of A highlights user B's entry in his buddy list, and then selects "View"
2. A sends a special view message to GXLS containing B's username
3. GXLS checks that A is allowed to view B (by using the database) and then sends a special message to B, asking the HUD to start uploading screen contents to the web server
4. B contacts the web server and starts uploading JPEG images
5. The web server sends a session ID back to B
6. B sends the session ID to GXLS
7. GXLS sends the same session ID to A
8. A contacts the web server, using the session ID acquired from GXLS and starts downloading B's JPEG images

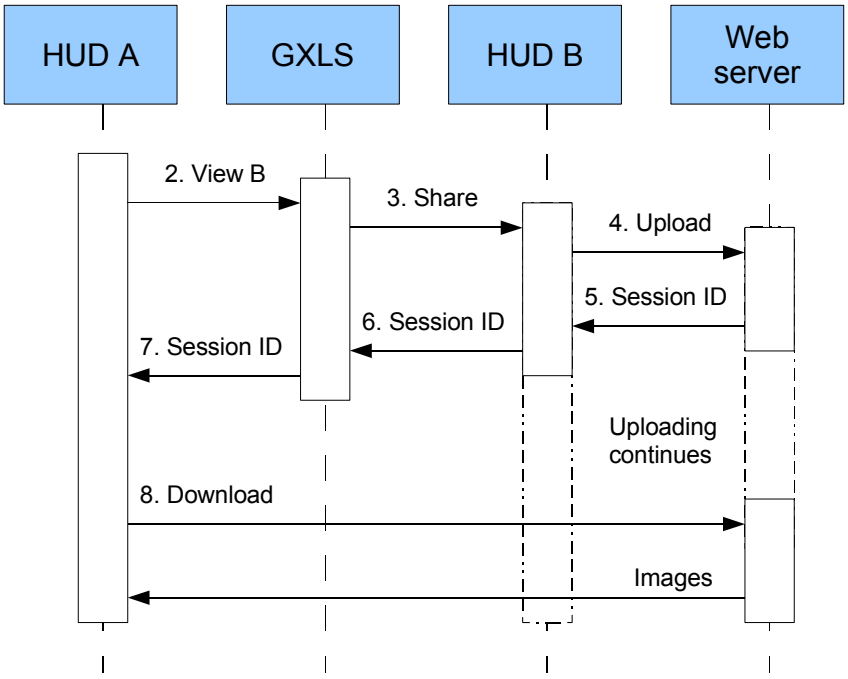


Figure 5.2: Sequence of events when viewing someone

Most of the steps in the sequence represent very little overhead, both in terms of processing power and network bandwidth. It is the 4th and the 8th step that represent the bulk of the workload in both respects, and for this reason, these are the ones we seek to improve.

5.1.2 Client modules

The Geelix HUD application is coded in C++ in its entirety, and is divided into several modules, each containing a number of C++ classes. The number of classes is quite high and it is unnecessary for this document to list them all, but the modules can be seen in figure 5.3, and the classes of the sharing module later in figure 5.4 on page 35.

The program works by inserting itself as a layer between DirectX and the game. This is done by placing a Windows DLL-file in the game folder, which provides the game with an alternative DirectX object. The calls from the game can then be intercepted and extra operations can be performed, such as adding graphics to the screen buffer before it is shown. This is what the HUD does to display its interface. In addition, input can be intercepted, and

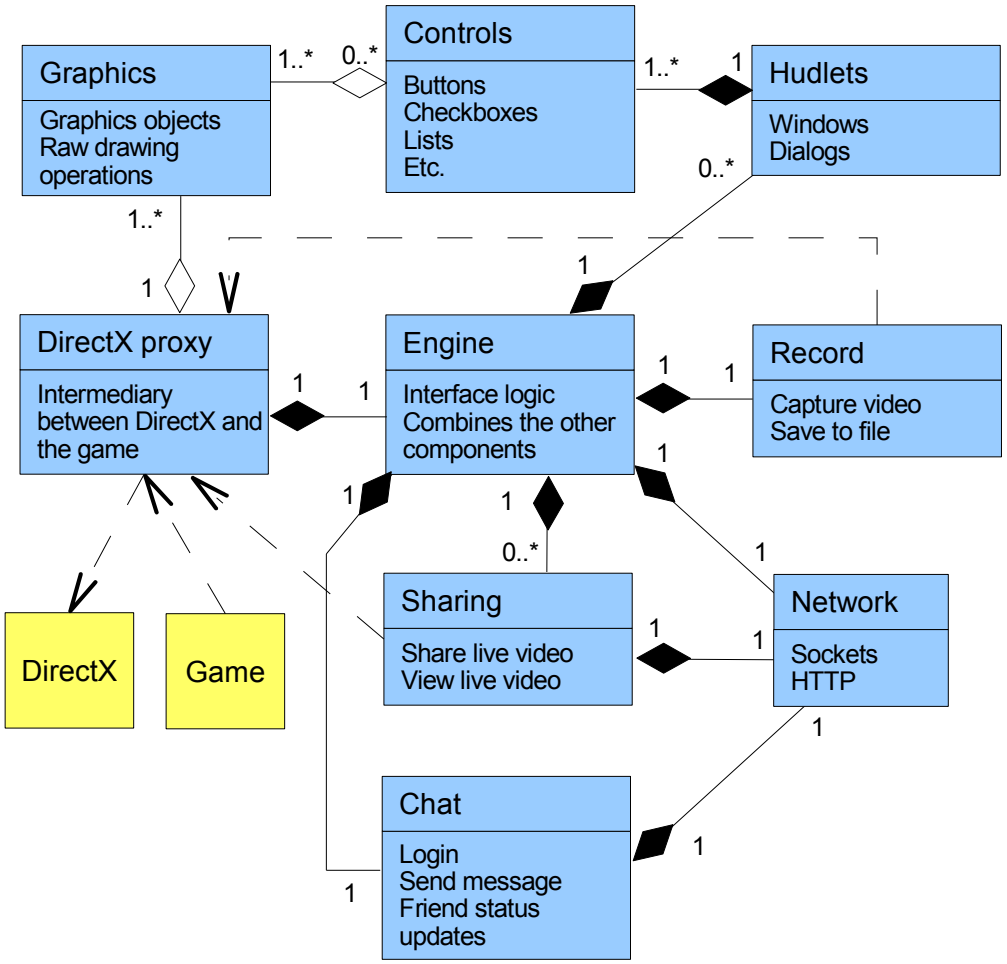


Figure 5.3: Modules in the old Geelix HUD

be used as input to the HUD or just be passed on to the game. The DirectX layer object is the class group called `DirectX proxy` in the diagram.

The interesting input from `DirectX proxy` is passed on to `Engine`, which is the main class in the application, responsible for binding the other classes together, and routing messages where they should go. `Engine` contains a number of `Hudlets`, which are window and dialog classes. The dashboard and the chat window are examples of windows that are modeled by classes in this group. `Hudlets` contain `Controls`, which are window elements, such as buttons, tick boxes, text fields, window frames, etc. These in turn use `Graphics` objects to draw their visual representation onto. `DirectX proxy`

uses these `Graphics` objects directly to draw on the screen surface (the direct link is primarily for performance reasons).

When an input reaches `Engine` from `DirectX proxy`, it is passed down the chain of classes (including any nested `Controls`) until it reaches an element that is interested in the input. For instance, when a mouse key press arrives, `Engine` passes it on to the correct `Hudlet` (the window that received the click) which in turn routes it to the proper `Control`, in this case a button that was pressed. Similarly, when the button knows it has been pressed, it sends a message back up through the chain to signal any interested parties that it was pressed and an action should be taken.

The `Chat` classes implement the chat protocol, and makes it easy for the `Engine` object to log in and send and receive messages. This class uses the `Network` selection of classes for low level network communication with the Geelix servers.

The `Sharing` classes are responsible for streaming live video in both directions, and also utilize the low level `Network` classes to do so. The frame that is being sent, is compressed entirely within one frame of game time (that is, the game is not allowed to continue executing and producing new frames before the compression is complete).

When a chat message is received by `Chat` or a new frame downloaded by `Sharing`, `Engine` is notified so that it can update the corresponding `Hudlets`, either by inserting a message in a window, or direct a viewing window to draw a new video frame.

The `Network` classes are also used directly by `Engine` to use other PHP services than screen sharing, such as getting and setting of permissions and checking of available updates.

The `Record` classes are used to capture high definition video and sound, and saving it to a file. One can see that both `Record` and `Sharing` use methods from `DirectX proxy` directly. This is to capture the user's screen and is also done directly for performance reasons.

The Geelix application is completely single threaded (the game can create its own threads though), so the processor will only be running inside one of the modules at a time. Because of this, every operation that could potentially cause the processor to wait, for example when sending a network packet when the network is congested, is implemented asynchronously, so that the processor will save the task for later and continue running if it cannot complete it immediately.

The way the HUD is run is by hooking into the `Direct3D::Present()` call. This is the function that the game calls to display its rendered graphics to the screen [33], and hence we can guarantee that it is called relatively often.

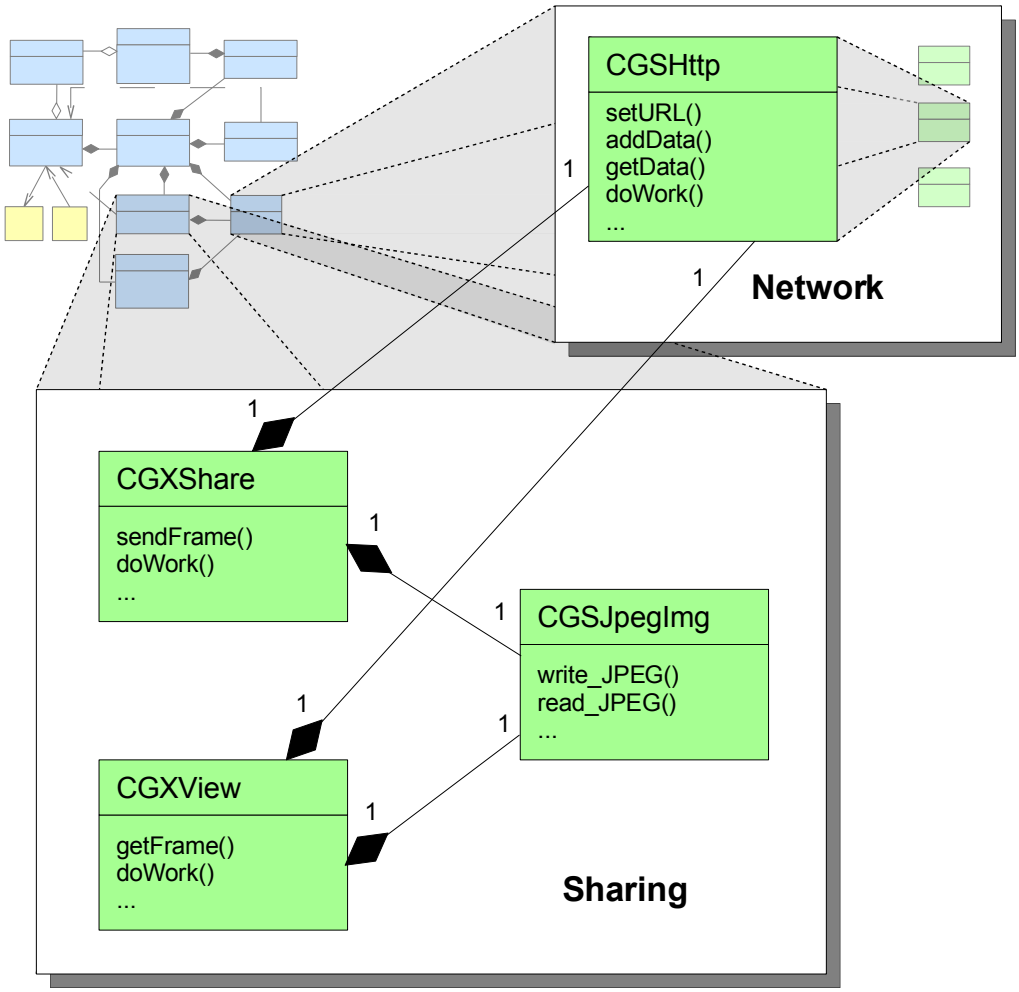


Figure 5.4: Classes in the sharing module

5.1.3 Sharing module

We now go into a little bit more detail of the `Sharing` module. The classes in this module can be seen in figure 5.4 along with a subset of the `Network` module.

Not all class methods are listed (a lot of getters and setters), but the most important ones are the initiative methods as well as the `doWork()` methods. An illustration of a sharing session is presented in the sequence diagram in figure 5.5. The `Engine` module initiates a sharing session by starting to capture the screen buffer at a given interval and calling `CGXShare::sendFrame()`, giving in the captured screen buffer. Then it calls

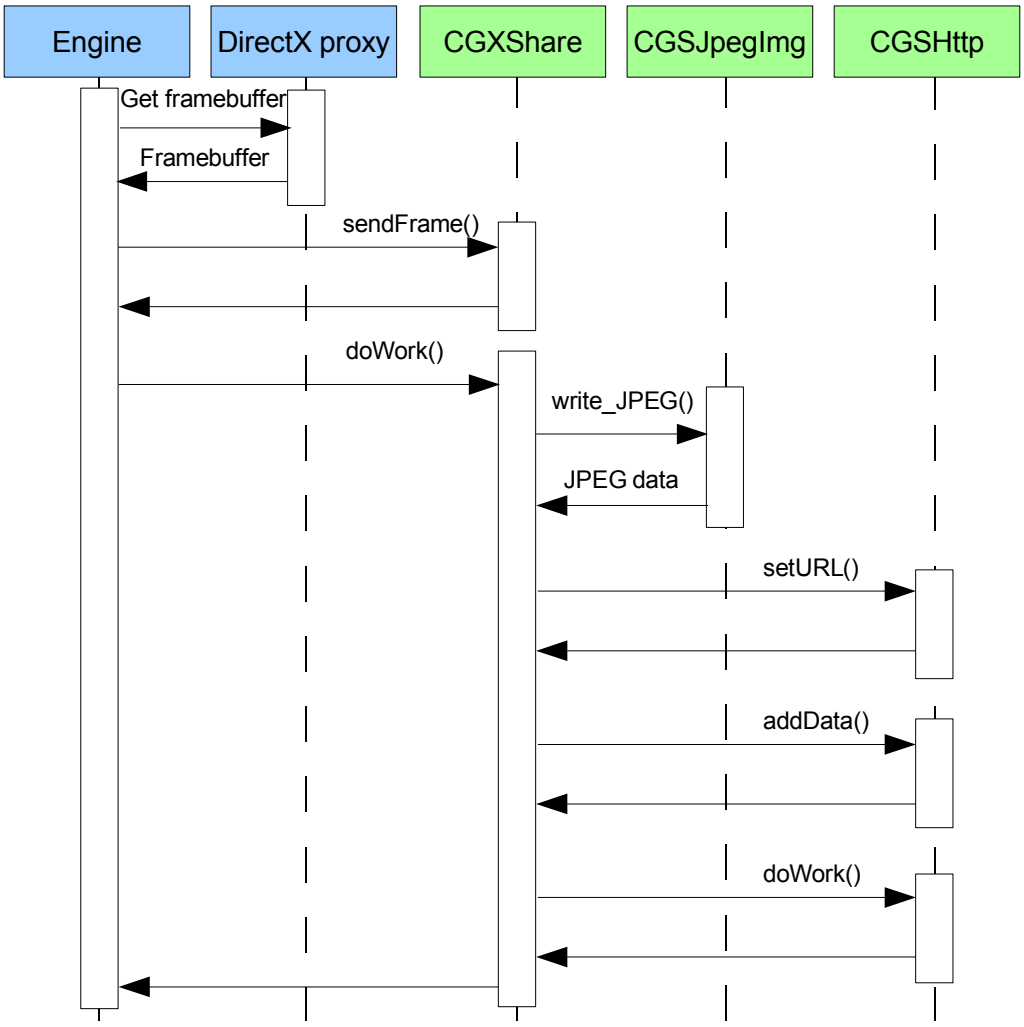


Figure 5.5: Sequence of function calls when sharing

`CGXShare::doWork()`, which is where the actual work is done. This method in turn calls `CGSJpegImg::write_JPEG()` to produce a buffer containing JPEG data, and then submits this data to `CGSHttp` using its `addData()` method. The uploading URL is set using `setURL()`, after which `doWork()` is called to start the HTTP transaction with the web server.

As mentioned earlier, all the `DoWork()` methods are meant to return quickly if their work cannot be executed immediately. This does not mean that the calls execute quickly though. For instance, the `CGSJpegImg::write_JPEG()` call can use quite a bit of time, but that is

because there is a lot of work to be done, not because the processor has to wait.

A viewing session works in much the same way, only the process is more or less reversed, with `CGSHttp` being called before decompression in `CGSJpegImg`.

5.2 New server architecture

In the new server architecture we sought to do away completely with the old protocol being used and design our own from the bottom up (as described in chapter 3). This means that the new server architecture needed a brand new server application as well as a protocol. This section will go through the design of them. The programming language was chosen to be C++, a natural choice since most of Gridmedia's software is developed using it, and the new server was named GXVS (GeeliX Video Server).

5.2.1 Goals

Before the design of the server architecture started, some goals were set up to act as design guidelines. The first goal was make the server a lightweight application. This goal was set up both to help satisfy nonfunctional requirement 2 on page 29, and to keep the codebase as small as possible. By keeping the code base small, bugs are less likely to occur and is usually easier to maintain. Avoiding bugs is very important in a server application due to its position in the software chain. If the server crashes, it will take every connected user with it.

The second goal was to make the server distributed and portable, again to satisfy nonfunctional requirement 2. Making the server distributed means that the server can have any number of instances, and no instance depends on any particular other instance. This means that the scalability problem is solved by allowing the server to be replicated as many times as needed to accommodate the user load. By portable, it means that the server should recompile on a different platform with little or no changes to the code. The reason for this is that when the server is replicated, the new platform may not be Microsoft® Windows®, which is the development platform.

Avoiding any encoding was the third goal, and the reason is simple: Encoding is a very time consuming process, and doing it on a server would greatly limit the number of simultaneous users.

The fourth goal was to make a network protocol that is as simple as possible. This goal was set in order to make implementation and debugging

of the application easier, and also to incur as little network overhead as possible.

The fifth and final goal was to make the server resistant to packet loss. As mentioned in chapter 3, the server will use UDP which is not guarded against this type of error. If no effort was put into making it resistant, errors could occur when packet loss happens, and clients could be disconnected prematurely.

5.2.2 Protocol

Before starting to make a protocol, it is important to decide whether or not it will be used in a peer2peer network model or a client/server network model. Peer2peer means that the clients make network connections between themselves instead of going through a central server. The advantage of this model is that if the participants are close to each other geographically, the network signal has to travel a shorter distance, and is likely to get both lower latency and higher throughput. There is a good reason why we chose not to use peer2peer, however: Functional requirement 3 on page 27 becomes almost impossible to solve with it, because there is no central server for the web server to contact. FREQ2 is also likely to be more difficult, since in a peer2peer network, if four people are watching you, you need to send data to four people, hence four times the bandwidth requirements. In a central server model, you only send it once to the server and it sends it to the four people for you. This conservative bandwidth usage is important to encourage adoption of Geelix, and therefore, the client/server model was chosen for GXVS.

In section 5.1.1 we went through how the GXLS server is used for one HUD user to signal his interest in viewing another. We decided to keep this system more or less exactly as it is in the new implementation, so nothing was changed in that protocol. This means that we can assume in this section that the viewer is always able to obtain the ID of someone's sharing session (assuming the correct permissions are set).

In order to fulfill the fourth goal, the protocol was decided to be text based. This makes inspecting network traffic much easier, and thus is good for debugging. Making the protocol text based does however count against the same goal in a certain sense, since text based protocols are usually a less space efficient use of bytes than binary data. It was decided that this sacrifice would be small compared to the added benefit of easier debugging.

We clearly did not want the video data itself to be transmitted in a text based encoding, so care was taken to allow for binary data in the stream context. More about that in the protocol description below.

The protocol was developed to be a bidirectional request/response protocol, where either party may send a message which may or may not produce a response from the other. The client must initiate first contact, but after that, the server sends packets periodically to keep the connection alive. The server distinguishes between sharing users and viewing users, and the video streaming happens by forwarding all video stream data from the sharing user to all the viewing users with the same session ID.

A request consists of any number of lines, separated by newlines. Two consecutive newlines signals the end of a request. A request may look like the following:

```
LOGIN
VERSION 1
<username>
<password>
VIEW
12345
67890
```

This request is the first request sent to the server by a client, and signals that it wants to log in. `VERSION 1` is a version identifier designed to throw off clients that have an older version than the server. `<username>` and `<password>` should be self-explanatory. `VIEW` signals that the client is requesting a viewing session, and the session ID is `67890`. `12345` simply acts as a login identifier to make sure duplicate packets don't cause any confusion.

Assuming the username and password was correct, the response would be as follows:

```
LOGIN OK
12345
67890
```

The first line informs the client that the login was successful, and the two lines after it are simply repetitions of the IDs in the login.

Because the protocol is based on UDP, the stream must be divided into packets. There is no limit on how many requests can be put into one, but the protocol requires that each request fits entirely within that one packet. If either peer has accumulated enough requests to exceed the maximum size of a UDP packet, it must split the list into several parts and send them in different packets.

Request	Meaning
`LOGIN`	Signals the login action of the client and is followed by a `LOGIN OK` or `LOGIN FAILED` from the server. This is where the client tells the server if it is a sharer or a viewer.
`SESSION`	Because the protocol is based on UDP, each packet is an individual unit with no relation to any other packets. This request should be at the beginning of any packet (except the `LOGIN` packet) to signal that it is part of an ongoing session. Its only parameter is the session ID.
`ECHO`	This packet serves to keep sessions alive by pinging the peers periodically to check their online status. Those who do not reply with `ECHO REPLY` after several attempts are considered timed out. Both client and server may send this.
`FRAME`	Signals a frame of audio/video stream data or user input data. It contains a size number, `x`, followed by `x` bytes of binary video stream data, and is thus the only request which is not entirely made out of text. It also includes an offset parameter which allows the data to be split across several `FRAME` requests if the length exceeds the packet size, making `FRAME` the only request capable of splitting its arguments as well. Sharing clients will send this to the server, and the server will send it to viewers.
`REFRESH SESSION`	This is sent by the sharing client to generate a new session ID, which is sent back to the client by the server using `NEW SESSION`. See the explanation in the text for its purpose.
`REQUEST VIEWERS`	This is sent by the sharing user to get a list of the current viewers of the stream. Again, see the text for the explanation.

Table 5.1: GXVS protocol

Table 5.1 lists the available requests in the protocol, along with the reason for their inclusion in the protocol. To save space, the parameters of the requests are not included.

The `REFRESH SESSION` request deserves some explanation. It is present in order to deal with nonfunctional requirement 4 on page 29. The way it works is that when a user changes permissions, the HUD simultaneously sends this request to the GXVS server. When a new session ID arrives, the sharing

client switches to this new ID, and also sends it through GXLS to all the current viewers (which can be retrieved using `REQUEST VIEWERS`), *except* the viewers that are now denied access. Those viewers will be disconnected after the server invalidates the old session ID and thus can not watch anymore.

`REQUEST VIEWERS` is present for the previous request to work, but other than that, it works as a timeout mechanism for the client to stop sharing when there are no viewers. The client sends this to the server periodically, and when it discovers that there are nobody watching, it stops sharing (unless sharing has been explicitly enabled by the user, as per functional requirement 3.2). It also helps to fulfill functional requirement 8 on page 27.

To satisfy the fifth goal of being resistant to packet loss, we will see that none of the requests nor responses cause erroneous behavior if individual packets are lost. Massive packet loss will of course result in disconnection eventually.

If either `LOGIN` and `LOGIN OK` messages are lost, the client status is still not logged in. Therefore it will continue to send `LOGIN` until it succeeds. For the server this may mean that extra streams are created that are empty, but they will quickly time out using `ECHO`. `SESSION` messages are never lost unless their remaining payload is also lost, so the loss will cause no errors by itself. `ECHO` messages are meant to be tried several times before a host is considered dead, so individual losses are OK. If `FRAME` messages are lost, it means that the viewer(s) will most likely experience some video and/or audio distortion, but it has no other ill effects. The `REFRESH SESSION` can cause some problems if the reply, `NEW SESSION` with the new session ID is lost. This is solved by having the server cache the old value for some time, and letting the client rerequest a session ID if it has not received one within a given timeframe. `REQUEST VIEWERS` is purely informational and thus causes no errors if it is lost. It *is* possible for a refreshed session ID not to be sent to every viewer if the response to `REQUEST VIEWERS` has not been received for a while, but this might just as well happen if the viewer logged in exactly at the same time as a session ID change. It will never happen to viewers who have been logged in long enough for their presence to be detected by `REQUEST VIEWERS`.

The complete GXVS protocol as documented in the Geelix system can be seen in appendix A on page 106. Note that the appendix represents the protocol at the end of the project, which differs slightly from the designed protocol. More about this in chapter 6.

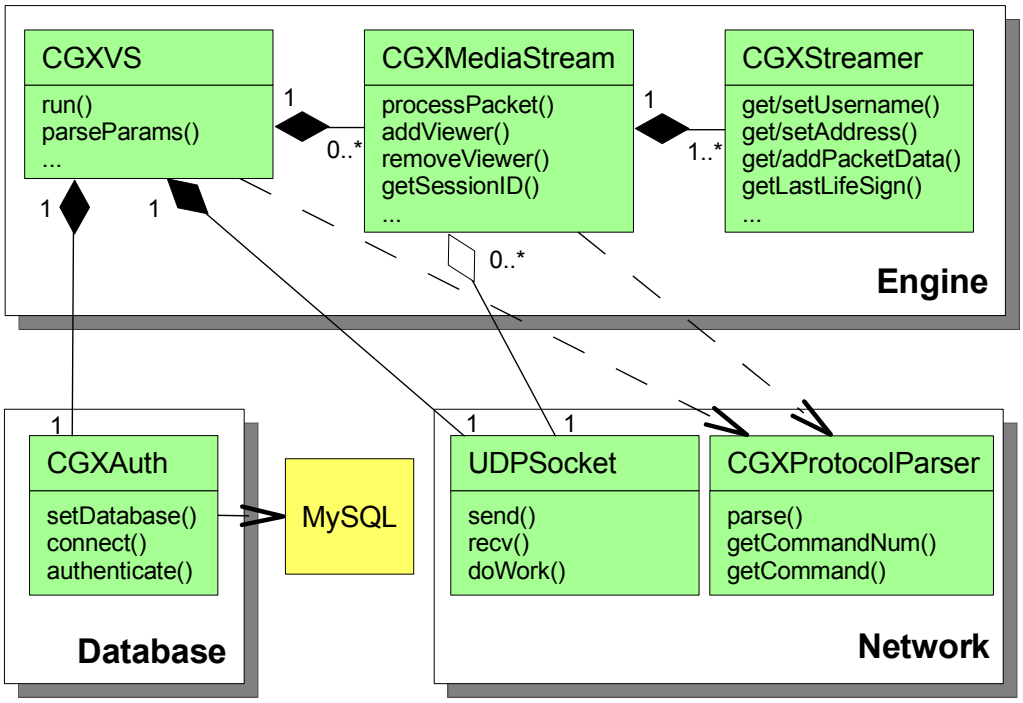


Figure 5.6: GXVS class diagram

5.2.3 Server classes

With the server being a lightweight application, we decided to implement it as a single threaded program. Single threaded programs are usually easier to implement correctly, and the author already had some experience with writing single threaded servers. We use asynchronous I/O operations to counter I/O delays in the single threaded application.

The program does not have many modules, so we display them together with the classes in the same diagram. They can be seen in figure 5.6. To save space, only the most important methods are listed.

`CGXVS` is the main class which is used to initiate and run the server. It is responsible for parsing command line parameters, initializing network and database connections and for running "the big server loop". Command line parameters are used to change for instance network port to listen to, and database login credentials.

`CGXMediaStream` represents one sharing session, and when `CGXVS` sees a packet that is part of an ongoing session (all non-login packets must start with a `SESSION` request), it will forward the packet to it. `CGXMediaStream` contains `CGXStreamer` objects that correspond to one logged in user. One

`CGXMediaStream` always contains exactly one sharing user and any number of viewers (including zero). If the sharing user decides to log out, every viewer of the same stream will be logged out as well, and the stream removed. Most protocol requests by clients are handled by `CGXMediaStream` except for logins (for which there is no stream yet).

The `CGXStreamer` class is simply a storage class and has no internal logic.

The `CGXAuth` class is responsible for maintaining a connection to the Geelix database, and authenticate users upon request from `CGXVS`.

`UDPSocket` is a class that implements an asynchronous UDP socket. It has methods to send and receive packets by the application and a `doWork()` method to actually perform the underlying socket operation. Due to being asynchronous, requests that cannot be handled immediately will be buffered in this class. The reason for making the interface asynchronous is to avoid delays in the server process as a result of waiting for I/O. `CGXMediaStream` objects share the same socket as `CGXVS`.

The `CGXProtocolParser` is a class that parses incoming packets by dividing lines into groups of requests (requests are called "commands" in the class interface), as well as carefully extracting binary data from `FRAME` requests. It is used by `CGXVS` to parse `LOGIN` and `SESSION` requests, and `CGXMediaStream` uses it to parse everything else.

There is no separate class for constructing requests and responses. Since each request/response is just a text string, there was not really any need for a separate class. When one of the existing classes want to send a request or response, they simply concatenate their arguments together in the correct order in a string, and send it to the socket.

5.2.4 Web viewing

To fulfill functional requirement 3 on page 27, the original plan was to have GXVS use the FFmpeg library to decode the stream (only the container, not the contained video), and send it as a WMV file to the web server so that people could view the stream with a browser. After testing different approaches, this turned out to be exceedingly difficult. Because GXVS uses a protocol prone to packet loss, frames may be lost, and when faced with this scenario, GXVS only has three choices: It could reencode the previous frame into the stream so that the framerate is kept. However, this breaks the third design goal of the server, which is to do no reencoding. Alternatively, it could insert the previous frame without reencoding it, but this will cause a lot of visual distortion because one frame may depend on previous frames. Finally, the server could adjust the timestamps in the video file so that the skipped frame leaves a "hole" in the file, but still plays at the correct framerate.

The two last options were the only really viable ones, but unfortunately, Windows® Media Player® was not willing to play either resulting file, and thus this functionality was dropped.

5.3 New client architecture

In the new client architecture, we wanted to solve a number of problems with the old one. The technological changes have already been described in chapter 3, so in this section we will focus on the problems that relate to the implementation of that technology, what changes it requires to the old code, and what new modules and classes have to be added.

Since the old HUD was using C++ in all its code modules, it was natural to use this programming language for the new client modules as well.

5.3.1 Goals

Since the live sharing project was a part of a bigger project (see section 1.1 on page 5), we worked on the overall design goals together and we tried to make the design beneficial to all parts. The other project that overlapped with this one was Richard Tingstad's "Recording and publishing of gameplay experiences", which tries to implement high definition recording to disk. That project also needed to do screen capture and encoding, although with entirely different distribution and performance goals. Even though figure 5.5 does not directly illustrate it, the old implementation captures the framebuffer once for every service that needs it, so that if sharing and high definition recording were both running at the same time, it would be captured twice. Our first design goal was therefore to redesign the HUD to have a central unit for doing screen capture. This would improve both efficiency and maintainability, since the work is only done once, and in a single place.

Our second design goal, which correlates strongly with the first, was to have a common interface for encoding of the framebuffer, so that the rest of the HUD does not need to concern itself too much with what happens to the framebuffer after it is sent to the encoder. The rationale for this is that the recording process is a separate process from the rest of the HUD, and apart from setting some initialization values, the process can operate on its own. This improves modularization of the code, which often improves maintainability and is considered good practice in projects with more than one developer [34].

One can see that the `CGXShare` class already does all the work necessary without much involvement from the `Engine`, but there is no common

interface, so the `Engine` does have to duplicate the same procedure for the high definition recording with only some minor changes. The two goals put together would therefore call for a class design where the HUD only needs to concern itself with one thing: Run recording modules.

The third design goal again correlates with the second one, and was to separate the recording process from the remaining HUD process by running it in its own thread. We believed (and still do) that this improves performance for several reasons: First, since the HUD performs all its functions within one frame of game time, this can produce quite noticeable hiccups in the framerate. By only initializing the work in the main thread, and actually doing it a different one, the HUD can return control to the game earlier and hopefully give a smoother, if not higher, framerate. Second, if the game ever sleeps in its loop, for whatever reason, in most cases the other thread will continue running, thereby using the processor more efficiently. And third, it enables the HUD to exploit dual core processors, since the operating system normally tries to balance thread execution among the available cores.

Running the recording process in a separate thread is made much easier because of our second goal of making the interface to the recorder as small and common as possible. This reduces the points where the HUD needs to do thread synchronization, which is both an expensive process for the CPU and a common source of bugs for the programmer.

5.3.2 Modules

Our new module design aimed at the given goals resulted in the modules seen in figure 5.7. There are three groups of modules in the diagram: The modules that run in the main thread, the ones that run in the new thread, and one that operates as a synchronizer object between the two. From now on we will call the new thread the "encoder thread".

As can be seen, the `Engine` module controls the lifetime of the `Encoders` and `Decoders` modules. However, the `Controller` module acts as a pipe interface to the `Engine`, and all of the encoders and decoders must be accessed through it. This is to ensure thread safe operation. When `Engine` wants access to an encoder or decoder (which should ideally be as seldom as possible), it asks the `Controller` module for it, which makes sure that the threads are synchronized before allowing access to the module. `Engine` then queries the objects in the `Encoders` module to find out if they need a screen capture and starts the capture if they do. If more than one encoder needs a frame, `Engine` will only capture it once and share it among the encoders. In addition, it collects any frames that the objects in the `Decoders` module have decoded.

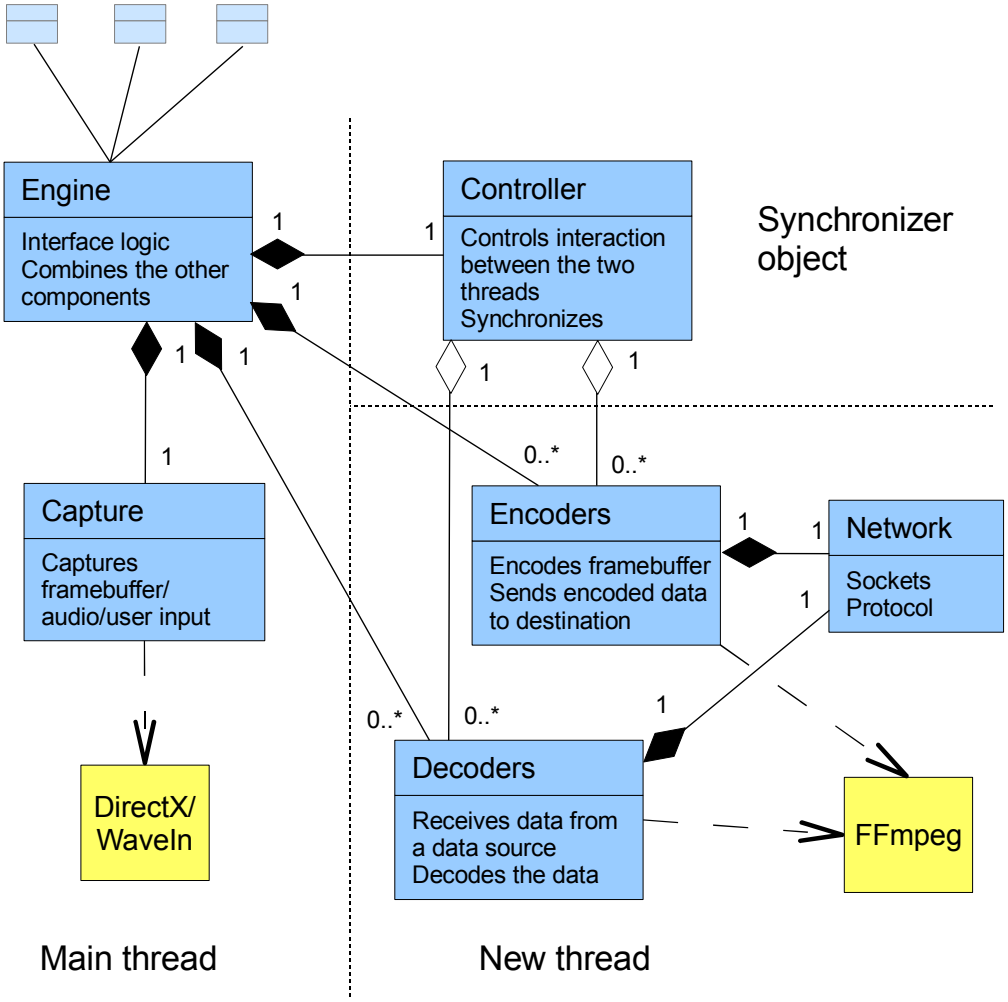


Figure 5.7: Modules in the new design

Other than to give access to `Engine`, the `Controller` module's work consists of executing the work of the `Encoders` and `Decoders` modules.

The `Capture` module is responsible for capturing screen data, audio from the sound card and user input events from the mouse and keyboard. When the document talks about "capturing a frame" it normally refers to capturing all these three, unless stated otherwise. The reader may be puzzled as to why the module is in the main thread. It performs an expensive operation, after all. The reason for this is simple: The call to `DirectX` which captures the screen contents is not known to be thread safe, and synchronizing on the object before calling will not work in general, because the game is free to call

any `DirectX` function behind our back. Even though it might be possible to intercept every function call the game makes and check for synchronization, this would add more complexity to the program than we would like, so we decided to leave it in the main thread. The same arguments apply to both audio and user input capturing.

The `Encoders` module is responsible for encoding data it receives from `Engine` into the WMV2 format, and outputting the result to the network using the `Network` module. For reference, in the high definition recording project, the network module will instead be a disk writing module, but we will not go into that in this document.

Similarly, the `Decoders` module is responsible for receiving data packets from the network using the `Network` module, decoding it, and making the result available in one of its `get` methods. The result will later be collected by `Engine`, which uses it to render a frame on the screen. Both the `Encoders` and the `Decoders` module use the `FFmpeg` library in their operation.

Note that the `Network` module does not operate exclusively in the encoder thread. For example, the `Chat` module from figure 5.3 on page 33 also uses the `Network` module, but from the main thread. However, as long as they maintain distinct instances of the module objects, thread safety is not an issue.

The work flow in the new set of modules can be seen in figure 5.8. The thick dotted line in the middle signals the boundary between modules that run in different threads. The main thread is on the left (thread 1), and the encoder thread is on the right (thread 2). The process loops, so the last "Get encoders/decoders" call in `Engine` is a repetition of the first one.

Whenever the two `Controller` executions synchronize their threads (remember: they are really just one object running in two threads), some operation is performed on an object that crosses the thread boundary.

The tasks being done in the sequence are:

1. Capture a frame (thread 1).
2. Send frame to `Encoders` (synchronized).
3. Encode audio/video (thread 2).
4. Send encoded data to network (thread 2).
5. Fetch incoming data from network (thread 2).
6. Decode incoming data (thread 2).
7. Hand decoded data over to `Engine` (synchronized).

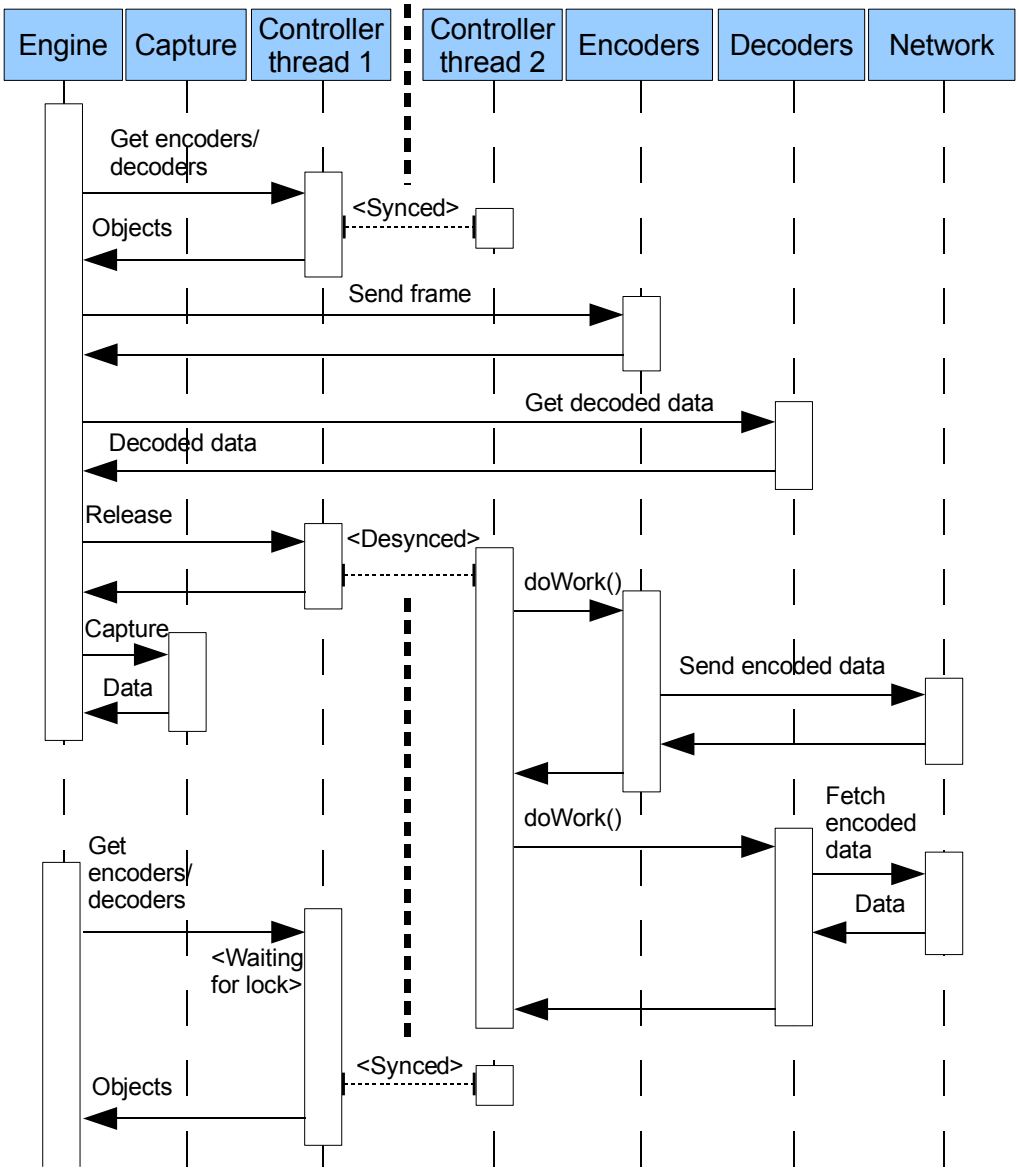


Figure 5.8: Execution sequence in the new design

One can see from the figure that the tasks are executed in a slightly modified order. This is to maximize parallelism between the processes. Passing pointers between `Encoders` and `Decoders` is done first, which implies that there has been a previous run where the results were buffered. This is done while the threads are synchronized, but each of these operations is very fast, both of which are simply passing a pointer to data. This leaves the big operations to be performed while the threads are desynchronized.

After the pointers have been exchanged, steps 1 and 3–6 are performed in parallel in each thread. This should be an efficient use of resources, since capturing is a memory transfer and therefore mostly system bus intensive, while encoding/decoding is CPU intensive.

Note that the timescale in the figure is not at all correct. First of all, the "Get decoded data" operation is much quicker than for example "Capture", since it only assigns a pointer. In addition, the pause between the two executions of `Engine` is likely to be much longer, since this is where the game gets to run.

Whether Geelix HUD has to wait long for the lock in the last synchronization remains to be seen in practice, and is likely to vary greatly both between different games and different computers.

5.3.3 Classes

Figure 5.9 shows the classes in the new design (only the most important methods are listed). One can see that the class layout follows the module layout quite closely. One of the differences is the splitting of the `Encoders` and `Decoders` modules into class hierarchies. The top classes support the basic operations like `doWork()` and `getFrame()`, but it is up to the subclasses to implement them. This was done to make the implementation of a different type of encoder easier. For instance, the high definition recording project was experimenting with a non-FFmpeg based codec at the time, and in addition, Gridmedia may want to add the native Windows Media encoder in the future, both of which should fit nicely into the hierarchy.

The `CGXShareEncoder` and `CGXViewDecoder` classes were also separated from the base `FFmpeg` classes, so that their purposes could be separated in the code. `CGXFFmpegEncoder` is strictly a stream encoder using FFmpeg, while `CGXShareEncoder` is a specialization that also includes streaming the encoded data to a server. After `Engine` has submitted an encoder (which has its initialization done by that module) to `Controller`, `Controller` does not need to know about the exact kind of encoder it is handling, as long as it performs the standard `CGXEncoder` calls. The same arguments apply to the `Decoders` hierarchy.

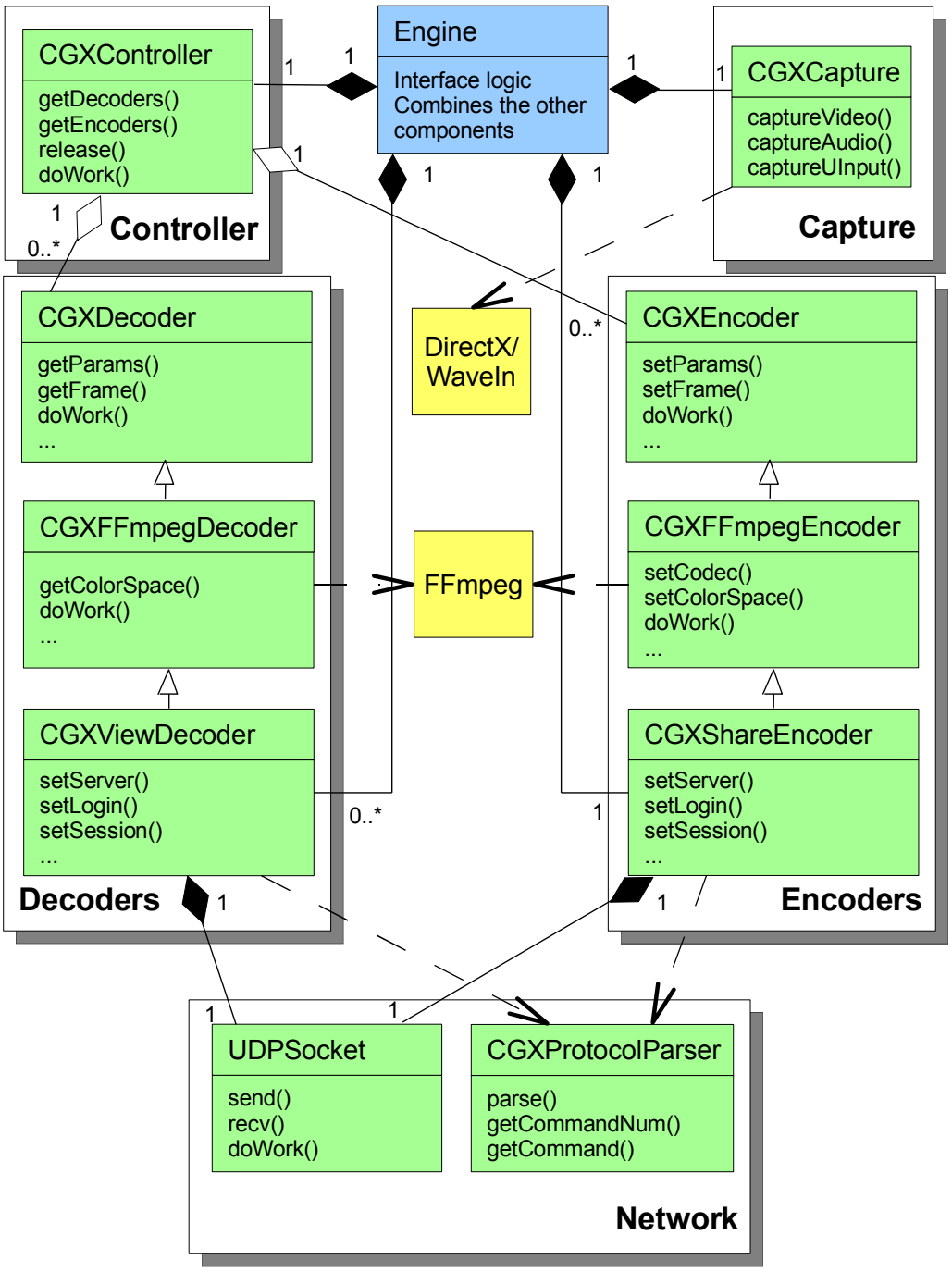


Figure 5.9: Classes in the new design

The observant reader may have noticed that the composition relationship between `Engine` and `CGXShareEncoder` has changed from 0..* to 1 as compared to figure 5.7 on page 46. This is because the module design included the possibility of more encoders, such as the high definition recorder, but this diagram is focused on the sharing service only, where just one encoder will be active at a time.

Both modules use the two classes in the `Network` module, `UDPSocket` and `ProtocolParser`. The two classes are the same as the classes in GXVS, and their operation is described in section 5.2.3 on page 42.

5.4 Fulfillment of requirements

This section will go through all the functional requirements (FREQs) from chapter 4 and determine how the planned design will fulfill them. In the cases where a requirement was deemed too difficult to fulfill, an explanation as to why will be given. Keep in mind that the author is not responsible for the GUI elements, only the "backend", but he *is* responsible for making sure the element gets made by the appropriate person. Hence, a requirement is not considered fulfilled if it is not possible to activate it from the GUI.

FREQ1 is realized by fulfilling each of its subrequirements.

FREQ1.1 is fulfilled by using the same system as in the old HUD, using GXLS.

FREQ1.2 is fulfilled by using the `CGXCapture` class to grab the framebuffer and have `Engine` feed it to the encoder.

FREQ1.3 is fulfilled by using the `CGXCapture` class to record audio from the sound card and have `Engine` feed it to the encoder.

FREQ1.4 is realized by using the same viewing window as in the original HUD, but feeding it frames from the `Decoders` module instead of the original `CGXJpegImg` class. In addition, the window will be changed slightly to allow playback of audio and a keyboard overlay for the user input stream.

FREQ1.5 is fulfilled by simply not calling the framebuffer capture in `CGXCapture`, leaving only the audio capture call. The function will have a separate button in the interface, and will be called voice.

FREQ1.6 is fulfilled by the implementation in the same way as the previous one, but there is no planned GUI for enabling the feature yet.

FREQ2 is fulfilled because Geelix will use a central server for streaming video, which will forward streams from sharing users to any number of viewers.

FREQ3 will not be fulfilled because of difficulty with Windows® Media Player® and packet loss (see section 5.2.4 on page 43).

FREQ3.1 will not be fulfilled (see above).

FREQ3.2 will not be fulfilled (see above).

FREQ4 will be fulfilled because the `Capture` and `Encoders` modules will get their operating parameters from the Windows Registry. Geelix Desktop GX will be responsible for setting the parameters.

FREQ4.1 will be fulfilled because the encoder scales its input according to given parameters before encoding.

FREQ4.2 is realized by having `Engine` only call the capture function as many times as is necessary to maintain the given framerate.

FREQ4.3 will be realized by giving the expected bitrate to the encoder before it starts its work. The bitrate output probably will not be 100% correct.

FREQ4.4 has not been specifically mapped in the design. At the time of the design, there was some uncertainty as to how this would be realized in the GUI, hence the low priority of this requirement. The parameters should be quite easy to change according to the path from which the HUD is run, so even if this requirement is not entirely fulfilled, it is very nearly so.

FREQ5 is fulfilled because the `Capture` module will capture user input events from the mouse and keyboard. The `FRAME` request in the GXVS protocol allows for transmission of these events, and FREQ1.4 specifies how they should be displayed.

FREQ6 will be fulfilled by using the same system as in the old HUD. GXLS will be responsible for only granting sharing IDs to authorized users.

FREQ6.1 is realized in the same way as above.

FREQ6.2 has not been mapped in the design because it was not clear at the time how this permission should be presented to the user.

FREQ6.3 is irrelevant because FREQ3 is not fulfilled.

FREQ7 is fulfilled by having the `Engine` module query `Controller` for whether the `CGXShareEncoder` class is active.

FREQ8 is realized by using the `REQUEST VIEWERS` request in the GXVS protocol to fetch the current viewers.

Chapter 6

Implementation

In this chapter we will go through the actual implementation of the system. The focus will be on the differences between the design and the implemented system.

6.1 Server implementation

The server implementation followed the design specification quite closely, and only a few changes were made, which will be discussed in this section.

6.1.1 Protocol changes

As the implementation progressed, a problem arose in FFmpeg's handling of video streams. The problem was that FFmpeg uses a buffer while reading stream data. While this poses no problem in itself, it adds a significant delay to the video stream, which is exactly what we did not want in our application. Some work was put into patching FFmpeg to make the buffer smaller, but the process was nontrivial. As was described in section 5.2.4 on page 43, fulfilling functional requirement 3 was difficult because of problems with playing the stream in Windows® Media Player®. With that player out of the picture, the only component which would play the video stream was the FFmpeg library in the HUD. For this reason it was decided to not change the buffer size in FFmpeg, but instead switch the stream over to a containerless stream, consisting of raw encoded video frames and audio samples instead of a WMV stream. The FFmpeg library would have no problems decoding such frames without a container format.

For this reason, some minor changes were needed to the GXVS network protocol. Things like video resolution, audio sample rate and framerate could

no longer be carried by the stream container, since there would not be one. The changes therefore involved adding this extra data at the corresponding places in the protocol. First of all, the `LOGIN` request was extended to include video resolution and audio parameters, as was its accompanying response, `LOGIN OK` (so that viewers would know how big frames to expect from the sharer). In addition, the `FRAME` request was extended to include some extra data about the frames, namely timestamp, type of data (audio/video), and whether the frame is an I-frame or not.

In addition, two new information fields were added to the `LOGIN` request, in order to provide the server with more information about the client. The first was a game ID number, which informs the server of which game the user is playing. Geelix HUD acquires this number by using Geelix Desktop GX. The second was a variable length array of custom fields where future information can be put. The array is represented in the protocol by first listing a number and then following up with as many fields as the number said. This was added to make the addition of new information easier without having to break compatibility between versions. A typical field might look like "`CPU type: Intel`" which would inform the server that the client is running on an Intel® processor. Both new information fields (game ID and the array) are also sent back to new clients with `LOGIN OK`.

The latest protocol specification at the time of writing can be seen in appendix A on page 106.

6.1.2 Implementation changes

Because of the failure to fulfill functional requirement 3, another, less ambitious feature was invented to replace it. The solution would consist of GXVS saving the packets it received to disk, and then running a separate encoding tool over this data (preferably from another computer) to produce valid WMV files playable by Windows® Media Player®. The file would then be presented on the portal, not in real time as planned, but instead after the video stream has completed.

This resulted in no structural changes to the server architecture, but some new server arguments were added to specify the directory to store packets in, and the server was modified to save packets to disk whenever it is specified. Disk writing can be a time consuming operation depending on the sizes involved, so this change could make the server less scalable. Gridmedia was willing to accept this, but in case it causes problems, packet saving can be turned off both by using command line arguments to the server, as well as logging in to the server with a special flag in the custom fields which disables saving.

The actual reencoding and publishing of the saved streams was not the author's responsibility and this document will not go into further detail on it.

The `CGXAuth` class was completely removed from GXVS. This was done because Gridmedia wanted to downprioritize authentication on the GXVS server and focus on the other remaining issues. The author warned the management about the potential security holes this would open[1], but Gridmedia maintained their stand on keeping it a low priority.

Ultimately, there was insufficient time to implement this feature. The consequence of this is that the GXVS server will accept any user credentials as login, including invalid usernames and valid usernames with invalid passwords. The implemented class diagram is therefore identical to figure 5.6 on page 42, except that `CGXAuth` is removed.

6.2 Client implementation

The implementation of the client changed quite substantially from the original design, for various reasons. Some changes were introduced because weaknesses were discovered in the design; others were introduced because the Gridmedia management made modifications to the design that had to be followed.

Figure 6.1 lists the classes in the final implementation. Again, only the most important methods are listed.

The figure looks quite different from figure 5.9 on page 50. We will look at how these changes came to be by taking each module from the design figure and explain what was the motivation to change it, and how it was changed.

The `Network` module contains the only change that was made by the author alone, which was the decision to create the `CGXVSClient`, `CGXViewVS` (VS stands for Video Server) and `CGXShareVS` classes. This was done because the protocol code which was designed to talk to GXVS turned out to be bigger than anticipated. It also turned out that the protocol code was very similar in `CGXViewDecoder` and `CGXShareEncoder`. In order to avoid having to maintain two distinct, but nearly identical codebases, this motivated the creation of `CGXVSClient`, where most of the protocol code would be put. This

[1]Without an authentication mechanism, the server is open to impersonation attacks, meaning that a client can pose as any user, and stream videos under their name, and GXVS has no way of detecting this. In addition, if the server protocol is reverse engineered (which should not be hard), any client can abuse the server to stream videos completely unrelated to Geelix, and GXVS will have no way of knowing which videos are legitimate or not. Even with authentication, this problem may persist, but at least the malicious action will be tied to a user account that can be suspended.

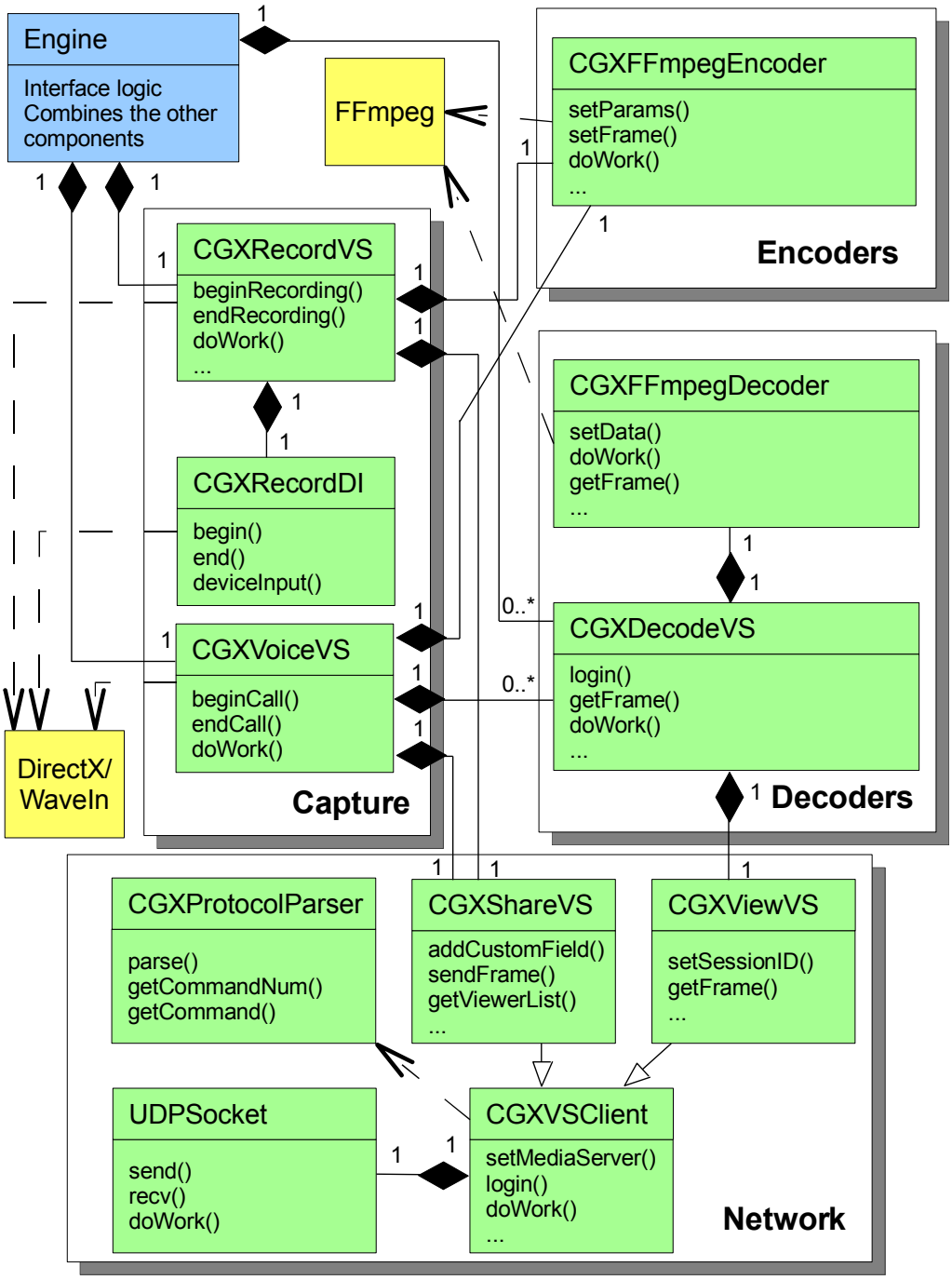


Figure 6.1: Implemented class diagram

common code consisted of tasks like pinging after given intervals to maintain the connection, and parsing and replying to server requests. The code that was specific for each service (sharing and viewing) was put into `CGXShareVS` and `CGXViewVS`.

The rest of the modules were changed because the Gridmedia management made a complete overhaul of the design. There was significant debating about whether most of these changes were necessary, and the discussion will not be reproduced in this section. Instead, the discussion can be read in section 9.1.2 on page 97, while this section focuses on the implementation consequences of the changes only.

The `Decoders` and `Encoders` modules had their class hierarchies basically collapsed into a single class in each module. The internal design was also changed slightly to allow the compression and decompression of raw frames instead of video streams, as was described in section 6.1.1. Their only difference in usage is that they handle raw encoded frames instead of frames encapsulated in a container format.

The new `CGXDecodeVS` basically fills the same role as the `CGXViewDecoder` in the design. Its purpose is to take stream data from the network using `CGXViewVS` and decode it using `CGXFFmpegDecoder`, and then hand the result to the caller of `getFrame()`.

The changes to the `Capture` and `Controller` modules are somewhat related. The `Capture` module was split into classes corresponding to the intended recipient of the captured frame, rather than corresponding to the action itself (capturing a frame). This resulted in the two new classes, `CGXRecordVS` and `CGXVoiceVS` (as well as a third class, `CGXRecordSE`, belonging to the high definition recording project), all doing frame capturing. The first class is responsible for recording frames for use with the sharing function, while the second is used by the voice function (and thus captures audio only). This means that the classes in `Capture` now capture frames separately for each purpose, possibly recapturing the same frame several times. In addition to doing capturing, the two classes fill the same role as the `CGXShareEncoder` from the design, which is to encode and stream frames to the server. They do this by using `CGXFFmpegEncoder` for encoding, and `CGXShareVS` for communicating with the server.

At the same time, the `Controller` module was eliminated from the design and its class' function, which was thread management, was duplicated and moved into the three classes just mentioned in the `Capture` module. This means that the classes in `Capture` each run in their own thread, and do their own thread management and synchronization. Using threads for decoding was removed completely.

The last class in `Capture`, called `CGXRecordDI`, was made in order to have a separate class capturing user input.

When the design was changed, the responsibilities also changed somewhat, and thus the classes `CGXRecordVS` and `CGXRecordDI` were mainly implemented by Gridmedia, and not the author. `CGXVoiceVS` was implemented by the author by copying `CGXRecordVS` and making the necessary changes to make it work for voice service.

Figure 6.2 shows the new work flow in the HUD when sharing. Again, the thick dotted line represents the boundary between objects running in separate threads. The `CGXRecordVS` objects are really just one instance running in two threads.

As can be seen, the work flow has changed somewhat. The most apparent change is that `CGXRecordVS` now synchronizes the two threads before calling capture, meaning that no other work is performed while that call is running. In addition, the lock, capturing and encoding is skipped entirely if the previous encoding is not complete, as can be seen by the "Busy" response in the diagram.

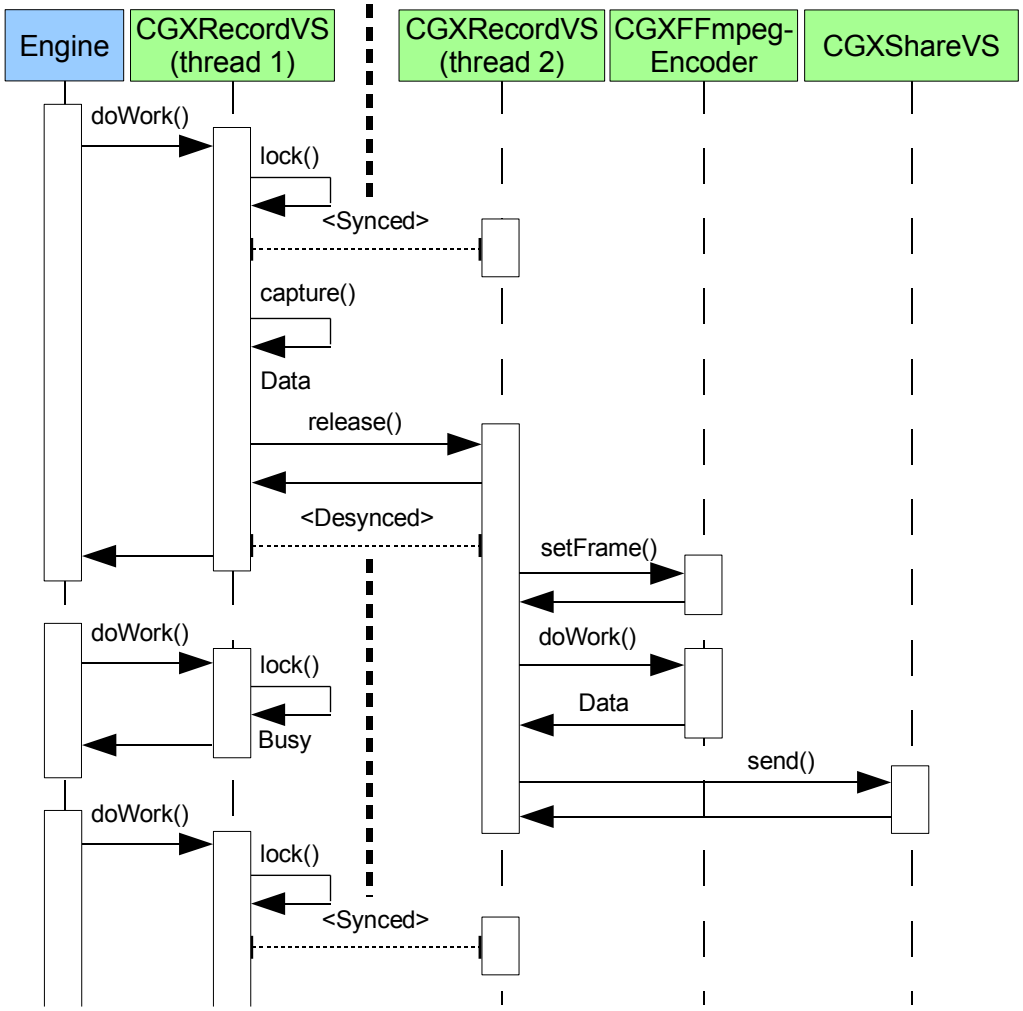


Figure 6.2: Implemented work flow when sharing

Chapter 7

Testing

This chapter will deal with the testing of the implemented Geelix system. It will first go through the tests that determine fulfillment of requirements, and then move on to performance testing and an analysis on their impact on the Geelix system.

7.1 Fulfillment of requirements

This section goes through the system tests performed on Geelix. The tests determine whether or not the individual functional requirements are fulfilled. The tests will be executed by using the new HUD GUI (Graphical User Interface), which can be seen in figure 7.1. As was explained in section 4.1, the author was not responsible for making GUI elements. However, there still was a responsibility to make sure that the elements were made by appropriate person, and therefore a requirement is not considered fulfilled if it cannot be accessed from the GUI, even if the technical prerequisites are in place.

Afterwards the section looks at the fulfillment of nonfunctional requirements.

7.1.1 System tests and results

The system tests and their results can be seen in tables 7.1 to 7.7 on pages 63–69. Not all of the functional requirements were tested, because some were known not to work before the testing started.

In the first table, table 7.1 on page 63, functional requirement 5 is not fulfilled. This was a feature that was intended to be in the HUD, and most of the framework is indeed in place, but the visual display was not finished in time. The data is however recorded, transmitted using the GXVS protocol

Figure 7.1: New Geelix HUD interface

Test name	Live viewing
Tests requirements	FREQ1.1, FREQ1.2, FREQ1.3, FREQ1.4, FREQ2 and FREQ5
Preconditions	• Three users, A, B and C are all on each other's contact lists. • A and B are allowed to view C. • C's recording device is set to microphone in Windows audio settings.
Procedure	1. Log into the HUD using three different computers for A, B and C. 2. Click on C on A's computer. 3. Click on "View" on A's computer. 4. Click on C on B's computer. 5. Click on "View" on B's computer.
Expected result	A should see C's gameplay in a window in his HUD, hear audio from his microphone, and see his keypresses on a keyboard overlay in the window. B should see and hear the same in his HUD.
Fulfilled requirements	FREQ1.1, FREQ1.2, FREQ1.3, FREQ1.4 and FREQ2
Failed requirements	FREQ5
Errors	The keyboard overlay did not appear, so no user input could be examined.

Table 7.1: Live viewing

Test name	Voice
Tests requirements	FREQ1.5
Preconditions	• Two users, A and B are both on each other's contact lists. • A is allowed to view B. • Both users' recording devices are set to microphone in Windows audio settings.
Procedure	1. Log into the HUD using two different computers for A and B. 2. Click on B on A's computer. 3. Click on "Voice" on A's computer. 4. Click on "OK" in the popup on B's computer.
Expected result	A should be able to talk to B using his microphone, and vice versa.
Fulfilled requirements	FREQ1.5
Failed requirements	None
Errors	None

Table 7.2: Voice

Test name	Web viewing
Tests requirements	FREQ3.1 with an extra modification: not in real time
Preconditions	• The user is allowed to view himself.
Procedure	1. Log into the HUD. 2. Click on your own username. 3. Click "View". 4. Wait 10 seconds. 5. Exit the HUD. 6. Go to the portal and select HUDs.
Expected result	Somewhere at step four, a viewing window should appear. The same contents should appear on the portal at step 6 within 24 hours (usually much faster).
Fulfilled requirements	FREQ3.1 with an extra modification: not in real time
Failed requirements	None
Errors	None

Table 7.3: Web viewing

Test name	Web sharing
Tests requirements	FREQ3.2 with an extra modification: not in real time
Preconditions	None
Procedure	1. Log into the HUD. 2. Click on "Share". 3. Wait 10 seconds. 4. Exit the HUD. 5. Go to the portal and select HUDs.
Expected result	The gameplay video should appear on the portal at step five within 24 hours (usually much faster).
Fulfilled requirements	FREQ3.2 with an extra modification: not in real time
Failed requirements	None
Errors	None

Table 7.4: Web sharing

Test name	Video tuning
Tests requirements	FREQ4.1, FREQ4.2 and FREQ4.3
Preconditions	• The user is allowed to view himself. • The Geelix Desktop GX's options are set to Reduce Factor 2, FPS 5 and Video Bitrate 512000.
Procedure	1. Log into the HUD. 2. Click on your own username. 3. Click "View". 4. Study the video. 5. Exit the HUD. 6. Start Geelix Desktop GX. 7. Enter Options in the Tools menu. 8. Change the Reduce Factor to 4. 9. Repeat steps 1–3. 10. Study the video. 11. Repeat steps 5–7. 12. Change the Reduce Factor back to 2 and change FPS to 10. 13. Repeat steps 1–3. 14. Study the video. 15. Repeat steps 5–7. 16. Change FPS to back to 5 and change Video Bitrate to 128000. 17. Repeat steps 1–3. 18. Study the video.
Expected result	In step 10, the video should be much smaller than in step 4. In step 14, the video should have lower framerate than in step 4. In step 18, the video quality should be worse than in step 4.
Fulfilled requirements	FREQ4.1, FREQ4.2 and FREQ4.3
Failed requirements	None
Errors	None

Table 7.5: Video tuning

Test name	Viewing permissions
Tests requirements	FREQ6.1
Preconditions	• Two users, A and B are both on each other's contact lists. • A is allowed to view B
Procedure	1. Log into the HUD using two different computers for A and B. 2. Forbid that A can view B by clicking on the sharing permission next to A on B's computer. 3. Click on B on A's computer. 4. Click on "View" on A's computer.
Expected result	A viewing session should not begin.
Fulfilled requirements	FREQ6.1
Failed requirements	None
Errors	None, but there was no message about the failed viewing, which is a little confusing.

Table 7.6: Viewing permissions

Test name	Sharing awareness
Tests requirements	FREQ7 and FREQ8
Preconditions	• Two users, A and B are both on each other's contact lists. • A is allowed to view B
Procedure	1. Log into the HUD using two different computers for A and B. 2. Click on B on A's computer. 3. Click on "View" on A's computer.
Expected result	B should see in the GUI that he is sharing. B should see that it is A who is watching.
Fulfilled requirements	None
Failed requirements	FREQ7 and FREQ8
Errors	The GUI gives no indication at all about either status.

Table 7.7: Sharing awareness

and streamed to clients, even though they are not able to see it. Fulfilling this requirement should thus be quite easy by simply adding a visual display and feeding the data to it. The user input can indeed be seen on the web portal HUD videos.

As mentioned in section 5.2.4 and 6.1.2, functional requirement 3 (and its two subrequirements FREQ3.1 and FREQ3.2) was replaced by requirements without the real time demand, because of problems with playing the stream in Windows® Media Player®. The new requirements were then fulfilled, according to the results in table 7.4 on page 66.

The requirements tested in table 7.7 did not pass because the GUI is lacking the ability to display the information. The technical framework is in place and the information is available through existing function calls, so these requirements should be easily fixable by adding the supporting GUI elements.

The requirements that were not tested were: FREQ1.6, FREQ4.4, FREQ6.2 and FREQ6.3. The two first requirements were not implemented due to time constraints and the requirements' low priority. FREQ6.2 was not implemented because FREQ5 was not implemented. And finally, FREQ6.3 was not implemented because the original feature was changed. With the new feature, this requirement became obsolete, since the decision of whether or not people can see your HUD on the web portal is now made there.

7.1.2 Nonfunctional requirements

Nonfunctional requirement 1 was tested by taking five screenshots from Unreal Tournament and saving the images as JPEG, which is the compression used by the old HUD. The five images corresponds to five FPS. The total size of the images was 615 KB. As can be seen from figure 7.4 on page 80 in the performance testing, the total size of the images from the new stream is far lower, averaging around 70–80 KB/s. Even with this crude and possibly inaccurate method, the improvement of the new application can clearly be seen.

NREQ2 was not tested, due to difficulty in reproducing an authentic workload. However, theoretically the application should scale to any number of users due to its distributed nature.

NREQ3 also could not be tested, but care was taken to follow it whenever library and coding decisions were made.

NREQ4 was the only nonfunctional requirement that clearly was not fulfilled. Even though the GXVS protocol contains the necessary framework to support the operation (see section 5.2.2 on page 38), there was no time

Fulfilled:	FREQ1.1
	FREQ1.2
	FREQ1.3
	FREQ1.4
	FREQ1.5
	FREQ2
	FREQ4.1
	FREQ4.2
	FREQ4.3
	FREQ6.1
Almost fulfilled:	FREQ1 (5/6)
	FREQ3 (with modifications)
	FREQ3.1 (with modifications)
	FREQ3.2 (with modifications)
	FREQ4 (3/4)
	FREQ5 (needs GUI)
	FREQ6
	FREQ7 (needs GUI)
	FREQ8 (needs GUI)
Not fulfilled:	FREQ1.6
	FREQ4.4
	FREQ6.2
	FREQ6.3 (obsolete)

Table 7.8: Fulfillment of functional requirements

to implement the required HUD code to switch session IDs when changing permissions. Implementing this would take some coding on the HUD, but the server already has everything that is necessary for it to work.

7.1.3 Conclusion

In this section we looked at the functional requirements of the project and determined their level of fulfillment by using system tests. Most of the requirements were either fulfilled or nearly fulfilled, but there were some that failed.

To summarize, table 7.8 lists an overview over the status of the requirements at the end of the project:

FREQ6 was included in the almost fulfilled list because the two require-

ments that failed in its subrequirements both became obsolete after other requirements failed to be fulfilled.

Nonfunctional requirement 4 was the only nonfunctional requirement that was not fulfilled. Its inclusion was planned and implemented in GXVS, but there was insufficient time to implement the complementing code in the HUD.

7.2 Performance tests

The Geelix system performs a type of action that has been implemented successfully many times before. What sets Geelix apart is the environment under which the operations are performed. Even on high-end computers Geelix must be careful in its consumption of resources in order to not interfere with the gameplay, which typically drains a large portion of resources on its own. This section will try to shed some light on how well Geelix has succeeded in this area. To do this, a description of the test metrics will be presented, what tests were executed, what their results were, why the results are the way they are, and what can be done to improve weak areas.

Only the Geelix HUD client was tested in the performance test, not the GXVS server. Even though testing the server could definitely be useful to determine how well it scales, it was decided to focus on testing the HUD client. The reason for this is that the performance bottlenecks are primarily present in the user's system. If that system overloads, Geelix cannot acquire new resources elsewhere. The GXVS server however, is designed to be distributed, and Gridmedia can simply add servers to their farm in order to give it more resources. In addition, Geelix HUD is the application that is most uncertain when it comes to performance, since it accomplishes a task that has not been attempted before.

7.2.1 Metrics

When doing the performance testing it was chosen to rely on metrics that fulfill a certain set of criteria. They must:

- be quantifiable and comparable measurements.
- be as objective as possible.
- be able to be reliably captured using profiling techniques on the program execution code.

These criteria were chosen in order to make testing of many different settings a tractable problem. If other metrics were to be chosen that did not have these properties, testing would be either too time consuming or inaccurate.

The first metric chosen was the number of bytes transferred over the network as a result of using HUD functions. Such data can be captured by using network sniffers such as Wireshark [35].

The second metric chosen was execution time spent on various Geelix internal functions. Such data can be captured in one of two ways: One can use specialized profiling software that requires little or no modification of the original program. It works by adding extra debug code to the compiled output that records entry and exit from function calls inside the program. The other method is to trace function call timings by adding program statements that record timings at strategic places in the program, for example right before and right after a call to the render function call. This is the method that was chosen, due to its relative simplicity.

It was chosen not to measure memory usage, because it violates the third criterion. Because the game and Geelix runs in the same memory address space, it is difficult to measure exactly how much memory is being used by each of the parties.

A metric that was specifically avoided is that of image quality. While measuring image quality may seem straightforward at first glance, it contains many pitfalls. A simple bit-for-bit comparison is not suitable because the video codec is lossy, and thus will introduce many bit errors even though perceptible image quality does not suffer much. Another possible metric is PSNR (Peak Signal to Noise Ratio), which measures the ratio between the power of a signal and the power of noise that affects the fidelity of its representation. This metric is in fact used frequently when measuring the quality of an image after compression and decompression. However, this metric has been criticized for not correlating well with the actual human perception of image quality, as described by Wang [36]. Wang also states that "It has been reported that none of the complicated objective image quality metrics in the [paper's references] has shown any clear advantage over simple mathematical measures such as PSNR under strict testing conditions and different image distortion environments". Because of this difficulty in measuring image quality without the use of human test subjects, it was decided not to measure this explicitly.

Notes and examples will still be given when significant quality changes occur in the results, as observed by the author.

It was chosen not to measure audio quality as well, both because it suffers from some of the same problems as measuring image quality, but also because

the audio codec used in Geelix cannot be tuned below its default value, and therefore has no potential for losing quality given different settings.

7.2.2 Environment

The tests were all carried out on the same pair of computers, in order to maintain comparability between the obtained results. One computer acted as the client (running the HUD) and the other as server (running GXVS).

The computer configurations were as follows:

- Client computer:
  - CPU: Intel® Pentium® M 1.6 GHz
  - RAM: 512 MB
  - Graphics: ATI® Mobility™ Radeon™ 9700 64 MB
- Server computer:
  - CPU: AMD Athlon™ XP 2800+
  - RAM: 256 MB
  - Graphics: NVIDIA® GeForce4® 420 Go 32 MB

The client computer recorded the function call timings of its running HUD. The HUD was modified to store a log of every call to specific functions, most notably entry and exit of HUD code, rendering code, as well as subcalls in the HUD code such as encoding, decoding and screenbuffer capture. The log file was written using an intermediary buffer in order to minimize system overhead due to file I/O. The timings recorded were real time clock values, not CPU time, but this should not be an issue because the game and Geelix were the only applications running on the system. Two games were used in the testing: Warcraft III and Unreal Tournament 2004.

The server computer recorded network bandwidth used by capturing packets arriving to and originating from its GXVS server. The bandwidth used by communication to GXLS and the web server was not included, because they are not used by the live sharing protocol. The capturing software used was Wireshark. The server also ran an instance of the HUD in order to provide a second user for the first HUD to communicate with, but no results were recorded from it.

The network link between the two computers was a 100Mbps LAN (Local Area Network), with no other traffic running on the same wire, so packet loss should be negligible.

7.2.3 Tests

The tests were designed to get an overview of where Geelix has performance bottlenecks, and what effects different operating parameters have on these bottlenecks. These tests are important because they provide insight into how well the chosen technologies apply to the problem being solved. Since only a single test computer was used, little weight will be put on the actual result values; it is the correlation between different values that is considered important.

Because we want to focus on the applied technology and its properties, the tests were not designed to prove Geelix's success in providing a good user experience. Such a conclusion is also very hard to reach without doing live user testing, and because of time constraints, such testing is beyond the scope of this project.

Threads were disabled in the HUD before the tests were run, in order to get reliable timings on the different operations. The effect on the HUD is that all functions execute in serial order instead of in parallel.

One basic test consisted of running a game and using HUD functions according to a given timetable. Only one major HUD function was employed in each test so that its performance effects could be easily determined. Table 7.9 shows the timetable that was used for the tests. It assumes that GXVS is already running on the server.

Time (mm:ss)	**Action**
-00:10	Start network packet capturing with Wireshark
00:00	Start the game. The function call log will start at this point in time.
02:00	Start the intended HUD action.
02:10	Timestamp 1.
03:10	Timestamp 2.
04:10	End game and function call log.

Table 7.9: Performance tests timetable

The long period between the second and third step is intended to give the game time to load all of its resources and enter the normal game cycle. The next period is there to allow the HUD action to start and be fully operative by the time the clock reaches timestamp 1. No explicit action is performed here, but this is the point where data is to be extracted from the log. The extraction continues for one minute until timestamp 2. The remaining period

is intended to give a buffer zone for the extraction data, in case the stopwatch and the timestamps in the log file do not match up 100%.

The run was repeated for each HUD action, which are:

- HUD running in background with no windows or functions active.
- HUD window displayed on screen, but no functions active.
- Viewing another user's screen (main HUD and view window displayed).
- Letting another user view the screen (sharing, no window displayed).
- Having a voice conversation (no window displayed).

The session in table 7.9 captures the performance characteristics of one particular set of parameters for each HUD action, and thus were repeated many times with different parameters. The parameters that were adjusted in each run are as follows:

- Video frames per second: 5, 10 and 15
- Expected bitrate: 512 Kbps, 256 Kbps and 128 Kbps
- Video resolution (fraction of game dimensions): 1/2, 1/3 and 1/4
- Game resolution: 1024x768, 800x600 and 640x480
- Game color depth: 32-bit and 16-bit

The base configuration was 5 FPS, 512 Kbps bitrate, 1/2 video resolution, 1024x768 game resolution and 32-bit color depth, and all the other variations were compared to this one. Only one of the given parameters were changed in each run, and the same game level was always loaded.

7.2.4 Results & analysis

The result section will start by going through the overall results for time spent on running different HUD functions, and then it goes into bandwidth usage of the functions that utilize the network[1]. Afterwards, a more detailed breakdown of function call timings will be given, followed by the results obtained when changing key parameters in the HUD. Finally, an analysis on the impact on the game will be given, followed by a conclusion in section 7.2.5.

[1]The raw test data can be acquired by interested parties by emailing the supervisor for the project, Ole I. Holthe at `ole@gridmedia.com`. The amount of data captured from the game and the network is quite substantial, so if it were to be added to the appendix, the total length of the document would be more than 15 000 pages!

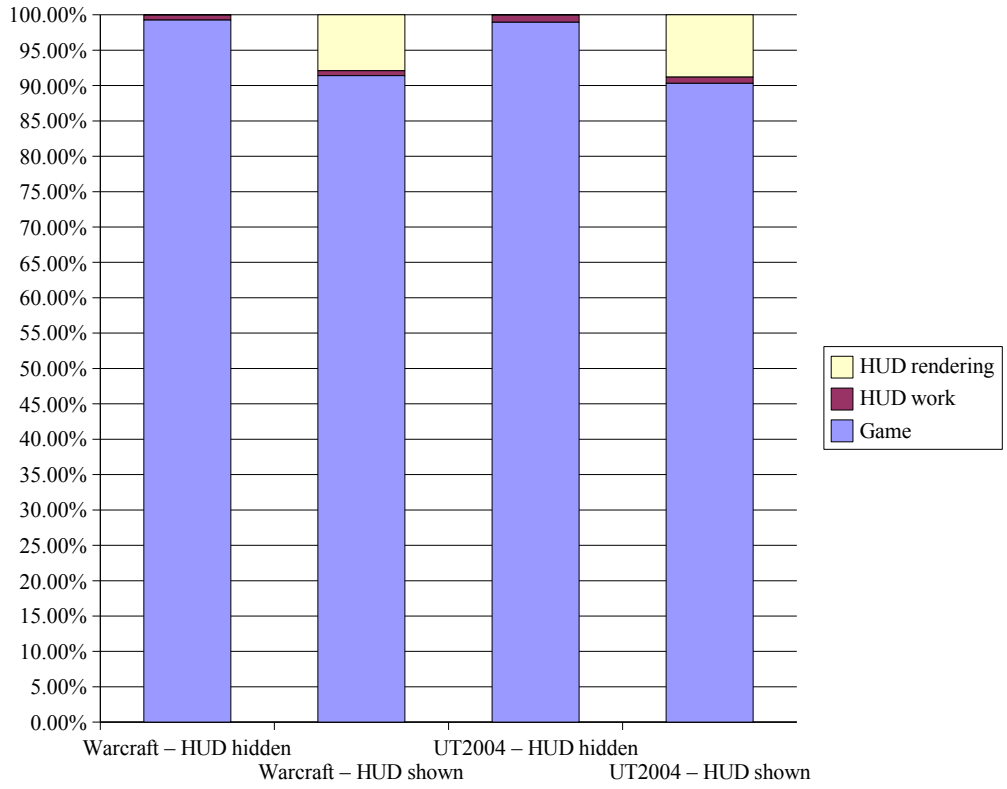


Figure 7.2: HUD running with no actions

Different HUD functions

Figure 7.2 shows the test results from the HUD when no action was performed. The results reflect the base parameter configuration, but most of the parameters should have no impact on these results, since they are not related to the video stream. To make it easier to interpret the values, the chart uses percentage of the total workload of the game and Geelix HUD combined. Obviously, no network traffic was generated during these particular tests.

One can clearly see that Geelix HUD's impact on the game when hidden is negligible. In fact, the HUD consumes less than one percent of the CPU resources when completely idle. As one would expect, the workload rises somewhat when the HUD display is brought into view, using 7–8% of the resources for rendering purposes. The rendering call consists of drawing the HUD components into an internal buffer if they are dirty (have changed since last rendering), and then transferring the buffer to graphics memory. The

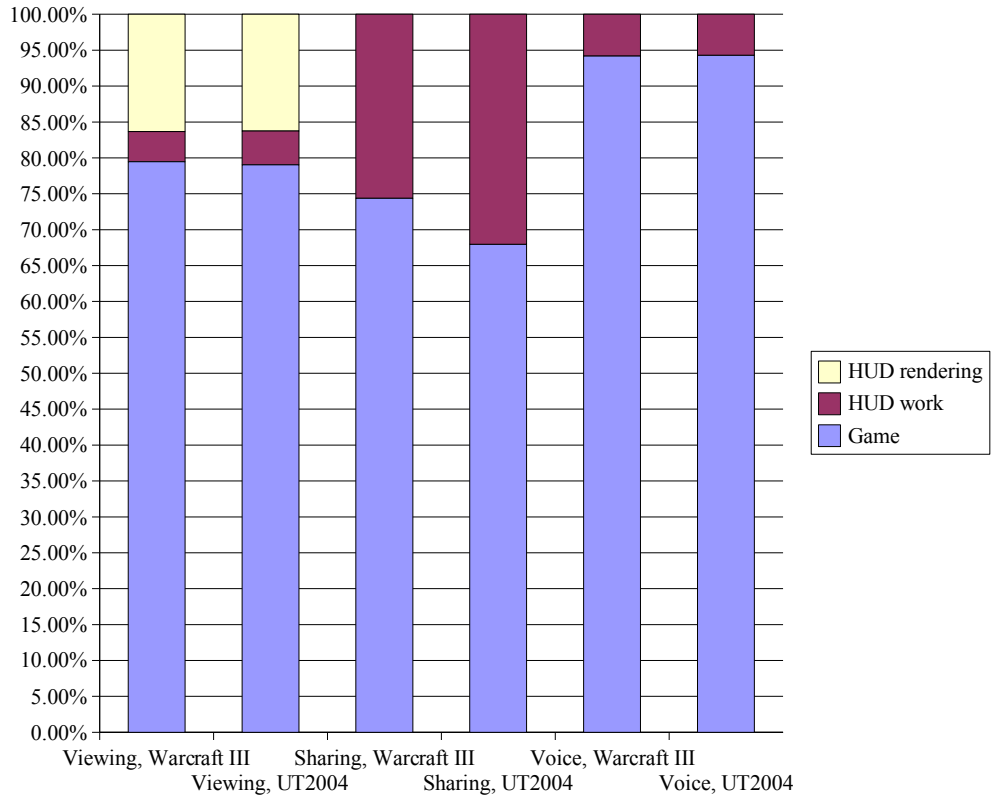


Figure 7.3: HUD running with various actions

last transfer has to be done every cycle because the game overwrites Geelix's graphics every time it updates its display.

There does not appear to be a big difference between the performance in the two games, with Unreal Tournament 2004 losing only about 1% extra CPU cycles to Geelix HUD, compared to Warcraft III. This can probably be attributed to the fact that UT2004 had a slightly higher framerate than Warcraft during the tests (UT2004 had approximately 59.4 FPS average while Warcraft clocked in at about 57.7). Thus the Geelix render code will be called more often and hence use more resources.

Figure 7.3 shows the performance results when the Geelix HUD functions were employed.

When using the view function, one can see that both the HUD render and HUD work portion have increased significantly. The HUD work has undoubtedly been increased by the need for decoding of both audio and video. In addition, the render code has more windows to draw and also has

to draw them more often, because the video frame will mark the window dirty every time a new frame is extracted from the video stream. The HUD also has to submit audio to the sound card, as well as query the network card for new video stream packets.

During sharing, the total amount of work goes up even more, but this time, rendering is not to blame. The HUD now has to capture the entire screen buffer, scale and encode it, capture audio and encode it as well, and output the video packet to the network card (see figure 6.2 on page 60). Encoding generally takes longer than decoding when using WMV2, so this has surely been a factor in the increase of the total workload. We will go into the details of this workload a little later in this section.

There is also a rather significant difference between the two games in the amount of workload. While one might think that this is because of the same reason as in the previous comparison (framerate causing code to be called more often), we will see later why this is not the case.

Using voice functionality has the same effect on both games, and remains a relatively CPU-friendly activity, using only about 6% of total capacity. Most of this is due to the encoding of audio samples.

Bandwidth

The bandwidth usage for the sharing function can be seen in figure 7.4. The rate of the stream is measured in bytes and sampled once every second. The average curves are calculated by using a weighted, linear average with seven steps. The exact formula is:

$$\begin{aligned} \bar{x}_i &= ax_{i-3} + 2ax_{i-2} + 3ax_{i-1} + 4ax_i + 3ax_{i+1} + 2ax_{i+2} + ax_{i+3} \\ a &= \frac{1}{4^2} \end{aligned} \tag{7.1}$$

As the figure shows, the average bandwidth consumption lies roughly between the 70–90 KB/s marks for both games. However, the sampled curves fluctuate periodically, and the maximum consumption is much higher, sometimes reaching beyond 150 KB/s. The reason for this repetitive fluctuation is the way the video compression works. It is set to generate an I-frame every 20th frame, and since I-frames are normally bigger than P-frames, the bandwidth consumption naturally rises as a consequence of this. The framerate for the video is 5 FPS, so a new I-frame should be generated every four seconds, a figure that seems to match with the chart.

Even though the average is the same, Warcraft seems to fluctuate more violently than Unreal Tournament, but the exact reason for this has not been pinpointed.

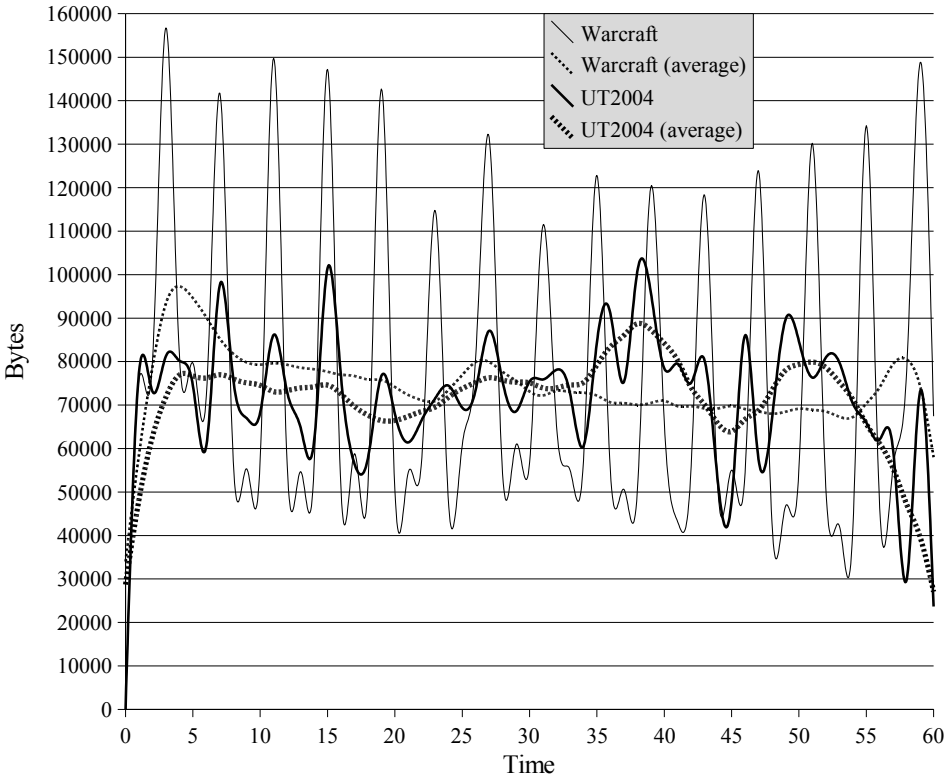


Figure 7.4: Bandwidth usage during sharing

Bandwidth data for the viewing function was not tested explicitly, because it is not necessary. The upstream bandwidth consumption of the sharing user will always be the same as the downstream consumption of the viewer.

In figure 7.5 one can see the bandwidth consumption when using the voice service, together with a similar sample taken when using the voice service available in Teamspeak [37]. It shows that Geelix still has a long way to go in implementing an efficient voice service. As mentioned in section 7.2.1, the audio codec cannot be tuned below its default value, which is 64 Kbps. Teamspeak uses codecs designed for speech such as Speex and GSM, and operates at bitrates around 10 Kbps (which is the default, it can be tuned even lower), so switching out Geelix's general purpose WMA2 codec with one more suited for voice communication would be an important first step.

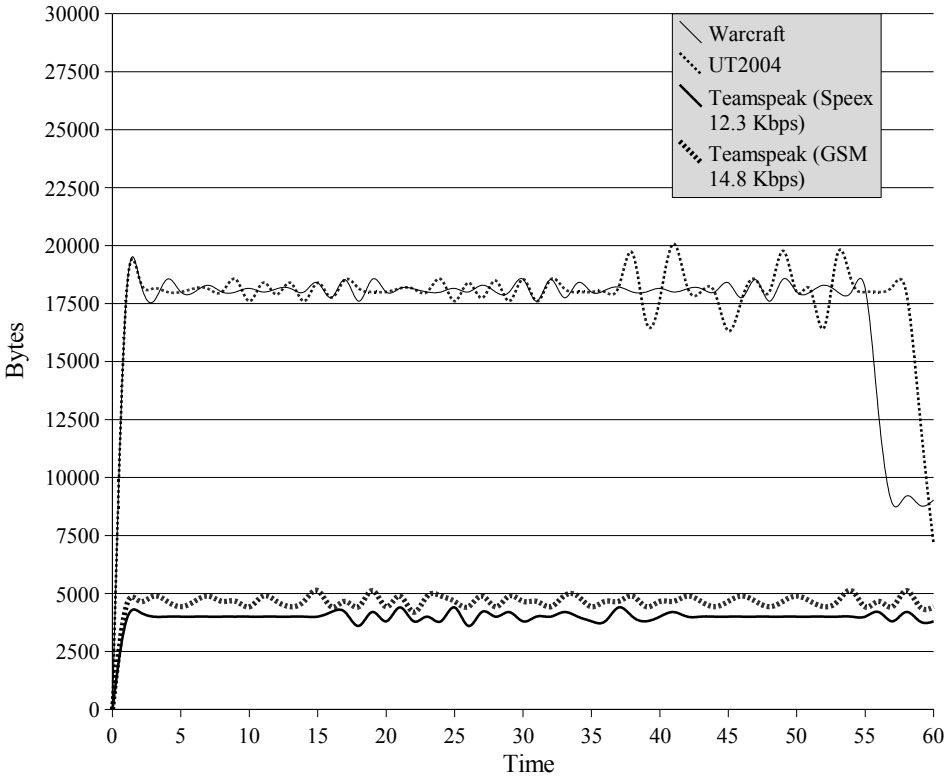


Figure 7.5: Bandwidth usage during voice communication

Workload breakdown

From this point on we will keep a focus on the sharing part, because the results clearly show that it has the biggest performance hit, both in terms of CPU and bandwidth consumption.

We will start by looking at a more detailed breakdown of Geelix's workload. Figure 7.6 shows how the HUD spends its time while sharing is running. The most time consuming piece of work being done (apart from the game) is capturing the screen buffer. This call consists of instructing the graphics card to dump its backbuffer (not the frontbuffer, since it has Geelix HUD drawn on top of it) to system memory, where it can be accessed by the CPU.

The CPU remains idle while waiting for the transfer to complete, which, given the timings in the results, wastes a quite significant portion of CPU time. It could be advantageous to explore different approaches to capturing the backbuffer, so as to keep the CPU occupied with other tasks while the

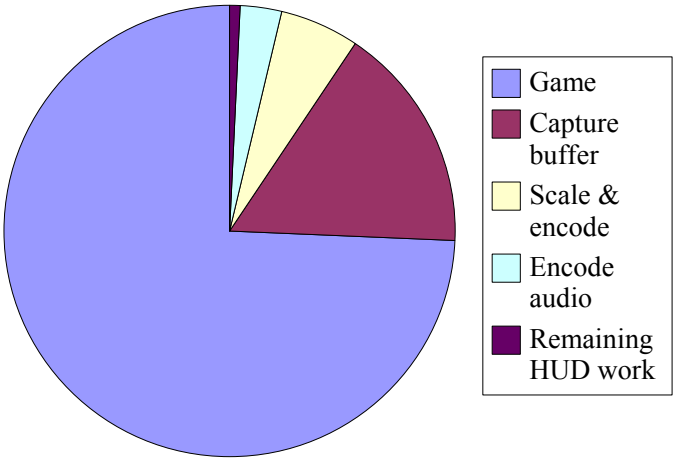


Figure 7.6: Breakdown of HUD workload while sharing

transfer completes. Note that even though threads were turned off for this test, the result in regard to capturing probably would not have changed much. This is because the implementation never runs capturing in parallel with any other operation, as can be seen from figure 6.2 on page 60.

Another approach is to try to rearrange the sequence of calls when initiating the transfer. As described in section 5.1.2, all of the HUD's work takes place within the game's call to `Direct3D::Present()`. However, the driver starts rendering the current scene before that, namely when the game calls `Direct3D::EndScene()` [33]. The Direct3D documentation encourages programmers to try to let as much time pass as possible between the two calls, in order to let the CPU and GPU work in parallel while the rendering takes place. For this reason, it is quite possible that the AGP bus has already been idle for a while by the time Geelix gets to start the transfer, if the rendering finished early. By queuing up the transfer right after the call to `EndScene()`, the graphics card would be given a chance to start the transfer immediately after rendering, hopefully shortening the time the CPU has to wait when reaching `Present()`. Whether or not it is a likely scenario that the GPU finishes before the CPU remains to be investigated, of course.

The second heaviest workload is the scaling and encoding of the video buffer, which is not a surprise. Most lossy codecs are well known to take

significantly longer to encode than to decode, and this could also be seen when doing the codec benchmark testing in the preliminary study. If capturing were allowed to run in parallel with encoding, there is a possibility that these workloads would overlap and use less total time. This is because capturing is a memory transfer, and therefore uses the bus intensively, while the encoding is more intensive on the CPU. Since these are different resources, even a single core CPU could benefit from running them simultaneously. This is what motivated the original thread design, but unfortunately, it was changed in the final implementation. The same arguments apply to the third heaviest workload, encoding audio.

The final workload, which consists of remaining work that the HUD has to do, consists of various tasks, which may use many different resources. Because of this, this workload might not benefit as much from running in a separate thread as the previous operations. In addition, it would be hard to make the program thread safe, since the work may touch many parts of the HUD memory space. It should not be a big problem running it in the main thread however, since the workload is the smallest of all.

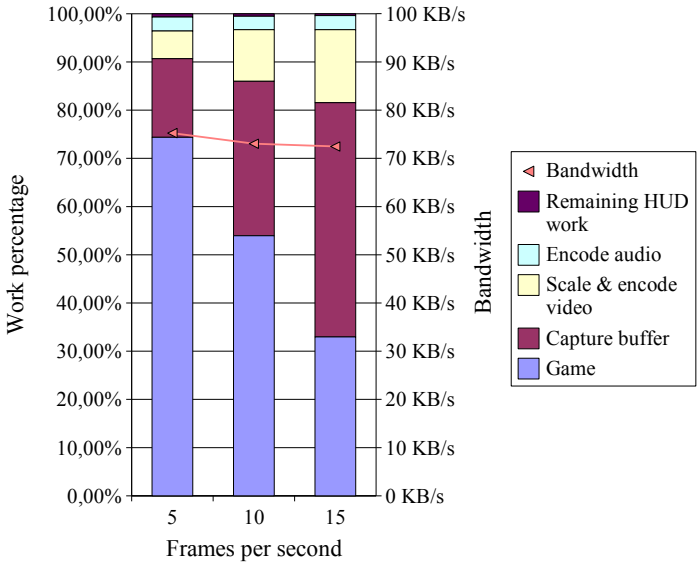
Different parameters

We now turn to the testing of various parameters. We will analyze their influence on performance when using the sharing functions, since sharing has the highest performance cost in all respects. Figures 7.7 to 7.9 on pages 84–86 display the test results, with the columns representing workload percentage and the curves representing bandwidth usage in KB/s.

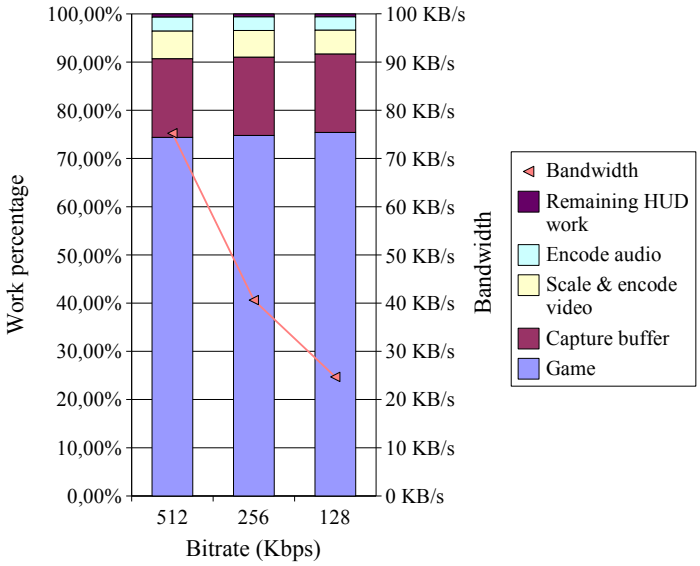
Figure 7.7(a) shows what happens if the number of frames per second is increased. As one would expect, the workload increases dramatically. Since every frame that is encoded must go through all the phases of being captured, scaled, encoded and sent, it is logical that the workload increases linearly with the number of frames per second. On the test computer, 65% of the CPU resources is a completely unacceptable use of resources. The number would obviously change in a different environment, but it remains a fact that increasing framerate of the video has a very expensive resource cost.

What is more surprising is the relative lack of change in bandwidth usage. In fact, it drops slightly when increasing the number of frames. It is not clear why this happens, but if one were to speculate, it could be that the motion vectors in the WMV2 codec (which have a limited search range), are more efficiently used when processing smaller image changes at a time.

Changing the bitrate gives the results displayed in figure 7.7(b). The workload remains almost unaffected, but the bandwidth decreases quite rapidly with decreasing bitrate. None of this is particularly surprising, but

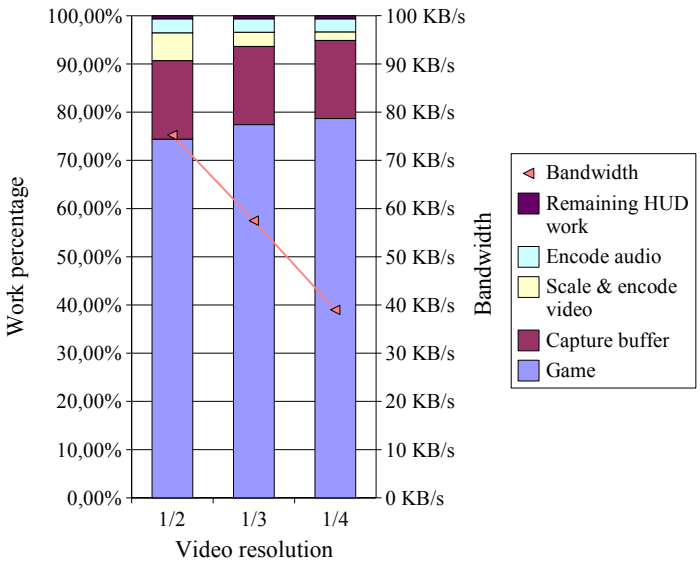


(a) Change of number of video frames per second

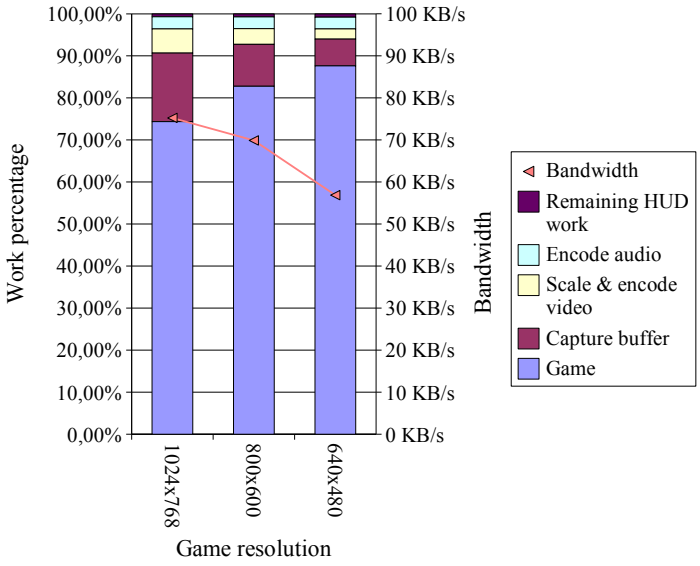


(b) Change of bitrate

Figure 7.7: Testing different parameters

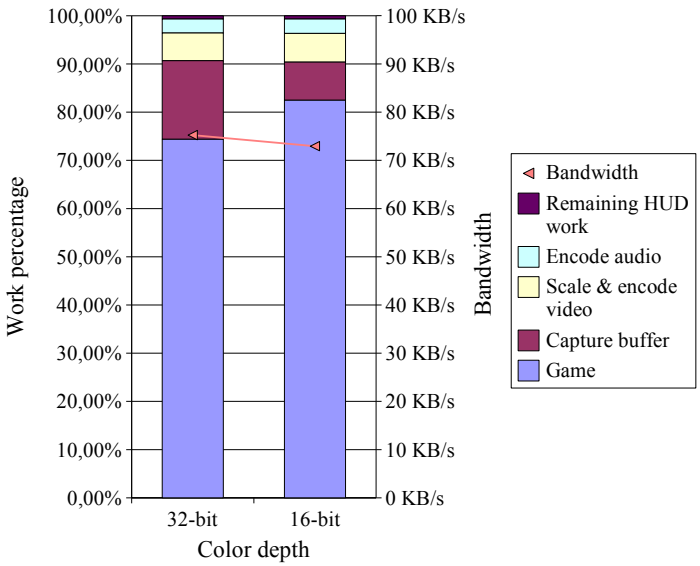


(a) Change of video resolution

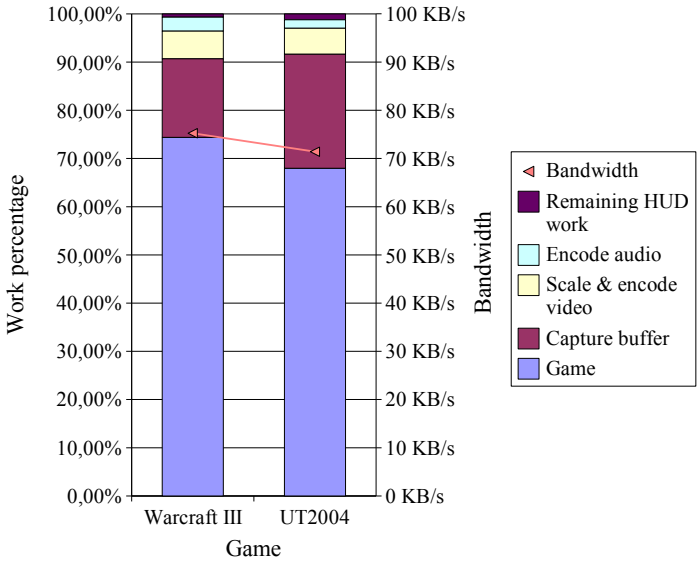


(b) Change of game resolution

Figure 7.8: Testing different parameters



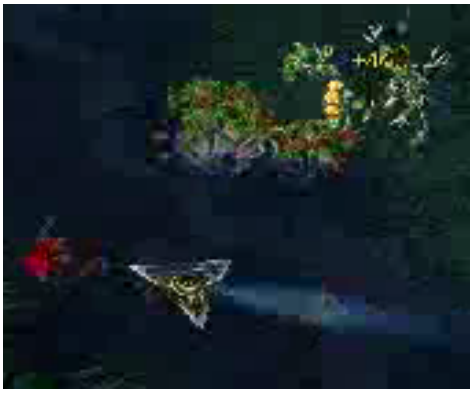
(a) Change of color depth



(b) Performance in different games

Figure 7.9: Testing different parameters

(a) Basic



(b) 128 Kbps bitrate

(c) 1/4 video resolution

Figure 7.10: Quality under different circumstances

at least it confirms that the bitrate setting is very effective at reducing bandwidth usage, even if encoding time is not noteworthy reduced. The reduction of bitrate has quite a significant impact on the visual quality however, especially when there is a lot of movement in the image. Figure 7.10(b) shows how the quality deteriorates. Once can see how the image has become more blocky, and the ground has lost a lot of its texture.

When changing the resolution of the encoded video, we got the results shown in figure 7.8(a) on page 85, with resolution expressed as a fraction of the game resolution. Capturing remains unchanged because the full buffer still has to be transferred from the graphics card, but video scaling and

encoding seems to have decreased in an apparently linear fashion, as has the bandwidth usage. Reduction of video resolution also has a significant effect on visual quality. Assuming that the image is scaled up to the same size as before the reduction (which is possible for the viewer using HUD controls), the impact on quality can clearly be seen in figure 7.10(c).

Changing game resolution has a similar effect, but has one advantage and one disadvantage, the former being that capturing now also takes less time, and the latter being that the game appearance is affected. The results can be seen in figure 7.8(b) on page 85.

Figure 7.9(a) on page 86 displays the results obtained when changing color depth. Capturing is cut in half because each sample is cut in half, which gives a significant performance increase, but could also affect the game appearance. How visible this would be depends on the game's use of colors. Bandwidth usage is almost unaffected, and is because WMV2 converts the image to the YUV color space before encoding, no matter what the original format is.

The last parameter change consisted of switching out the game, which in reality changes quite a lot of parameters. Figure 7.9(b) on page 86 shows the effects on the HUD. As could be seen also in figure 7.3 on page 78, Unreal Tournament 2004 causes a significantly higher workload than Warcraft III. What is puzzling is that this happens during capturing. Geelix is designed to capture only as many frames as is necessary to produce a video with the given framerate (5 FPS in this case). Because both games use the same resolution and color depth, one would think that the same number of bytes would be captured in both, hence taking the same amount of time. The results could either suggest that there is a bug in Geelix and it really captures more frames than necessary, or it could be a result of the discussion of `Direct3D` calls earlier in this section, where Geelix has to wait longer if the game's rendering is not yet finished. If this is the case, it strengthens the argument for trying to keep Geelix occupied instead of waiting while the memory transfer finishes.

On a side note, when comparing the two games by sitting in front of the system and trying to play, Unreal actually does *feel* more sluggish than Warcraft, but there may be many subjective factors involved in this observation. For instance, Unreal requires very precise aiming to play well, which could amplify the sensation of reduced game performance if mouse movements are less sensitive. This may not be as apparent in Warcraft, which relies less heavily on precise mouse movement.

Game impact

In this section we look at how the different workloads affect game performance,which in this case is measured in frames per second. There are many

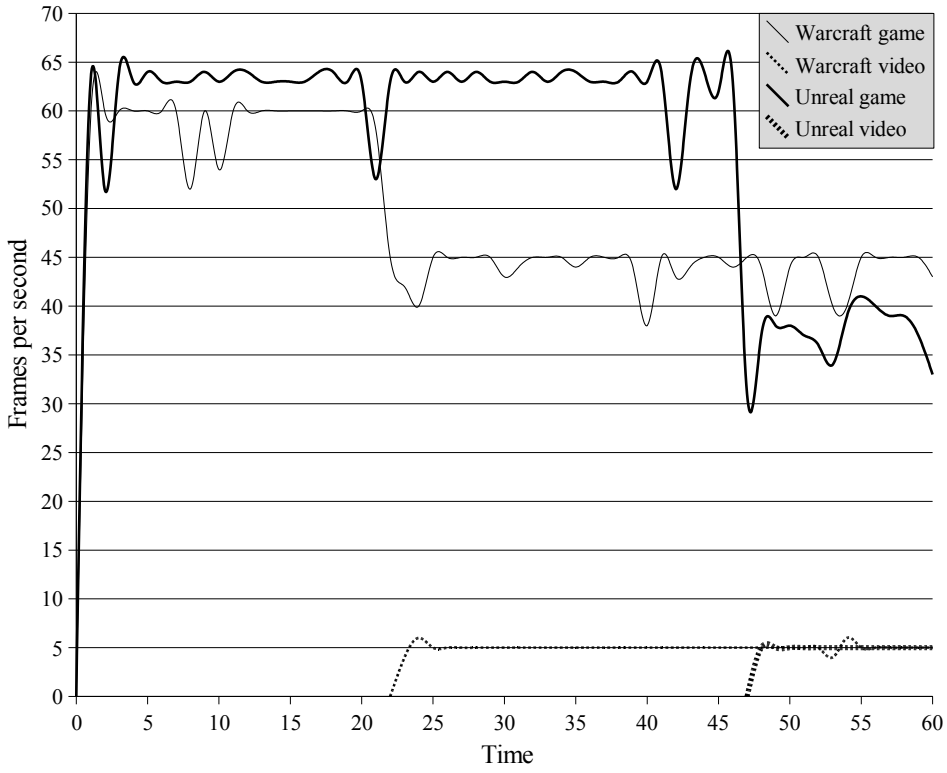


Figure 7.11: FPS when starting sharing

other ways to measure game performance, but FPS provides a fairly reliable and objective way to compare different performance levels. There will be some comments on other effects as well.

Figure 7.11 shows the impact of starting sharing for the two games. Sharing starts at approximately 22 seconds for Warcraft and 42 seconds for Unreal, and in both cases one can see a sharp drop in framerate.

Unreal's higher fall in framerate is no doubt because of the higher workload that the HUD exhibits when sharing in that game. In fact, if one

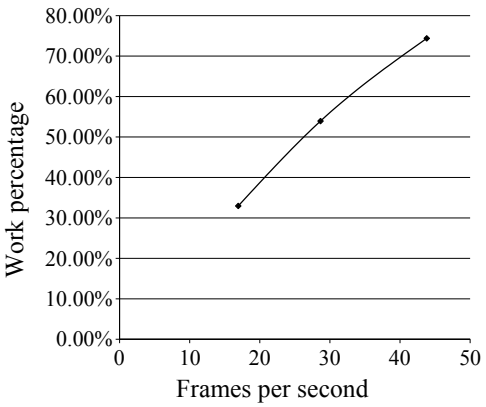


Figure 7.12: Correlation between game time and FPS

calculates the percentage that the framerate drops, it produces a number that is very close to the percentage of the game workload. For example:

$$\frac{FPS_{sharing}}{FPS_{idle}} \approx \frac{Gametime_{sharing}\%}{Gametime_{idle}\%}$$

$$\frac{45}{60} \approx \frac{76\%}{100\%} \tag{7.2}$$

$$0.75 \approx 0.76$$

Repeating this calculation using different settings yields similar results and suggests that there is a roughly proportional relationship between the time the game has to do its work, and the resulting framerate. The results of the calculation can be seen plotted in figure 7.12. The game framerate (and hence game time) was altered by adjusting the framerate of the video.

There is a quite important property about the reduction in framerate that should be noted. The mouse cursor behaves differently when Geelix causes the game to lower its framerate, as opposed to when the game lowers the framerate in another way (for example by having a CPU-intensive Windows task running). The mouse cursor normally has smooth movements, but when used with sharing. it makes small jumps when moved. This is is caused by the graphics driver locking itself in a system wide critical region, which means that no other interrupts will be processed. This is a documented feature

of the `Direct3DSurface::LockRect()` call (which is called to access the framebuffer) [33], and can be turned off. However, for this to work, Geelix must be improved to make sure that the game does not call any graphics mode functions behind its back (for example through a thread), otherwise the program would not be thread safe. Hopefully by doing this, the game would feel smoother even when reducing its framerate.

7.2.5 Conclusion

So what have we found out from this testing? First of all, we have confirmed that the HUD remains an extremely lightweight application when not in use, consuming less than 1% of the CPU resource on average on the test system.

The voice functionality also has a light processing footprint, but unfortunately, has some work to be done on the space efficiency part, with a network stream more than three times as large as the randomly selected competitor. Switching out the codec would be a very effective first step, because the current codec does not support bitrates below 64 Kbps, which is (according to TeamSpeak) unnecessarily high for voice communication.

Viewing someone else is where the CPU cycles start to add up, but it still holds within the 22% mark. The actual decoding takes up a relatively small part of this percentage, so any future work would be wise to start by looking at improving rendering efficiency. Several applications exist that do overlaying of graphics onto games (see section 3.1 on page 15), so investigating the methods of these applications could be a useful start.

Sharing has been the most interesting part of the testing, and it clearly represents the biggest challenge towards making the live streaming an enjoyable experience. Having sharing running exceeds the 25% workload mark on the test system, and gives a small, but definitely noticable reduction in game smoothness. It is also the activity with the highest bandwidth footprint, occasionally reaching as high as 150 KB/s. For reasons unknown, the two test games seemed to have a quite different effect on the bandwidth consumption. Warcraft fluctuated wildly between a high maximum value and low minimum value, while Unreal Tournament was more stabilized in between. Future work should spend some time on finding out why this is the case, and determine the possibility of adjusting parameters to reduce fluctuation in affected games.

Several parameters were tested to determine their effect on Geelix's resource consumption, and they were quite varied in how they affected the results. For a user wishing to free up CPU time, he has a choice of several parameters. The most effective one is clearly to lower FPS, or, if that is not an option, he could change the game resolution or color depth. Especially

the last option lowers CPU time quite significantly while causing very low visual distortion, but they both have the disadvantage of needing support from the game.

If the user is primarily interested in lowering bandwidth consumption, then adjusting bitrate is a very effective solution, as is lowering either the video or game resolution.

The Geelix sharing feature is not without problems, but it has been implemented with a rich set of parameters that can be tuned by the end user to suit his special needs, and so if Geelix does not provide an adequate gaming experience while doing sharing, at least it provides the option for its user to try to make it so.

Chapter 8

Future work

This chapter takes a look at some of the opportunities that exist for improvement of Geelix in the future.

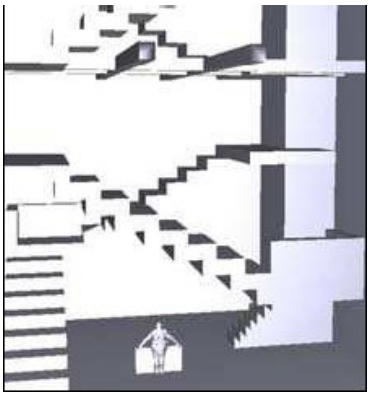
8.1 Custom codec

One obvious candidate for improvement is the codec used in the live streaming. Today's codecs are very quality/space efficient, but while this area of science is certainly interesting, a lot of research has already gone into this area. Alternatively, another interesting codec research area is the resilience to packet loss. The current codec introduces a lot of visual distortion when packets are lost because the codec is not built to handle it. Geelix would benefit from a codec that deteriorates more gracefully when losses are introduced.

8.2 Geometric capture

The experience sharing model in the current Geelix is based on videos, but there it could be interesting to explore other means of capturing experiences and sharing them. One such approach is geometric capture of DirectX 3D game application streams, which has been explored by Pan, Wei and Yang [38]. The way it works is by intercepting the data stream, command stream and rendering states from the DirectX pipeline and use it to recreate the scenes on the viewer's computer. This would allow freedom beyond what can be offered by conventional videos, by not only allowing the viewer to watch, but possibly also move around in the recorded environment, seeing things from different angles, and possibly even interacting with the scene in some way.

(a) Original scene captured from 3DMark03



(b) Scene without texture

(c) Scene with texture, but no lighting

Figure 8.1: Recreated scenes after capture. Images taken from [38]

Figure 8.1 shows some examples of how the scene can be reconstructed after a capture. Figures 8.1(b) and 8.1(c) show how the scene looks if textures or lighting are removed.

8.3 User testing

This thesis has been focused on the technological aspect of Geelix, but one very intriguing work prospect is to look into user's reactions to Geelix, if

they like the system, or if they have any ideas of their own. It would also be interesting to do some research on how the HUD is used in practice; if it enhances the social experience, and if so, what part of Geelix is the crucial part in that experience.

8.4 Remote gaming

A topic that always fired up a debate in the office was that of taking Geelix's way of interacting with the game even further, by allowing Geelix's users not only to be game observers, but game players. By this, we mean to let Geelix become the gateway to a game that is not even on your own computer. Such an advancement obviously faces vast challenges in respect to bandwidth capacity, network latency and processing power, but exploring this topic would mean trying to find compromises in the design which would allow Geelix to perform this feat under some given circumstances; something that would be quite unique.

8.5 Echo filtering

One of the problems when sharing with Geelix is that when two users wish to hear the game sounds from both their own game, and their friend's game, problems occur because of echoing. The problem is caused by the sound card not knowing the difference between internal game sounds, and the sounds produced by Geelix's remote game sound, hence it records them both together and this causes the signal to loop back and forth between the users.

A possible work prospect would be to look at methods to try to filter out this echo from the signal in order to produce a clean signal from both users, yet being able to hear the sounds from both games. The problem becomes even more challenging when the echo is caused by a feedback loop between the speakers and the microphone, in which case the signal will most likely be noise ridden.

8.6 Admin services in GXLS/GXVS

Another somewhat more mundane project would be looking at the possibility of adding administration facilities to the custom Geelix servers, in order to have greater control over their operation.

Chapter 9

Evaluation

This chapter will go through the evaluation of the project process, as well as the implementation and finished product.

9.1 Process

This section looks at the process of getting from the master thesis description at the beginning of the semester, to the final product and report at the end. It will touch upon some of the events that led things to being the way they are, as well as an evaluation of some of the decisions made by the Gridmedia management, and their consequences on the process, the implementation and the final product.

9.1.1 Structure

As mentioned in section 1.1 on page 5, Gridmedia Technologies AS has two employees in their office in San Francisco, as well as two graduate students, Richard Tingstad and the author. The CTO of Gridmedia, Ole-Ivar Holthe, which is also the author's supervisor, has been the main person participating in discussions about ideas for the project. He has also been the person to have the final word in matters where no agreement was reached.

Apart from Holthe's role as the one to have the final word, the structure during the project has been relatively flat, with input from each party considered just as important as that of any other party. Since we all have backgrounds with games, this led to a very free flow of ideas, where the direction of the discussion was often motivated by the parties acting as gamers, rather than as engineers or marketers. The author believes that this has been

a key element in the discussion, leading to several innovative ideas such as the visual representation of user input in videos.

It also meant the we, the students, were allowed to work on ideas that we found interesting and meaningful through a gamer's eyes.

9.1.2 Management decisions

The implementation saw many changes as it went from the design phase to the written code. As mentioned in section 6.2, many of the decisions to change the design were made by the Gridmedia management, and not the author himself. This section will shed some light on the discussion surrounding these decisions, and will go into some performance and design related aspects of their consequences.

As described in section 5.3.1 on page 44, the overall module and class design of the new HUD application was a joint effort between Richard Tingstad and the author. The Gridmedia management, consisting of O. Holthe, was largely not involved in the process at the time.

After the design phase was over, and implementation was well on its way, Holthe started taking an interest in the design at one of the weekly meetings. He expressed a general dissatisfaction with the design, saying that he was worried that the encoder was not working at the moment. Since the state of the implementation does not imply bad design, we wanted to understand the cause for criticizing the design, but our attempt was not very successful, with the main reason being cited as "It looks too much like Java".

The author does not believe that this is a valid reason to change the design, but nevertheless, we were willing to make modifications to the design in order to satisfy the management. We therefore asked Holthe to pinpoint the specific disadvantages that he was seeing in the design, so we could come up with an alternative design together. However, he was not able to pinpoint any specific flaws in the design, other than that he found it hard to understand. We offered to try to explain to him how the different modules worked, but Holthe instead said that he would have to think about the design himself. Without being much wiser about which parts of the design to change, Richard and the author had no other choice than to continue implementing as planned.

After a while, Holthe announced that he had implemented his own recorder class (replacing the high definition recorder), doing both capturing, encoding and saving to disk. The new recorder class did borrow a lot of code from the students' implementation, especially from encoding, but the interface was completely different. Tingstad's recorder was finished right about the same time, but Holthe wanted to use his recorder instead. The

author took this as a sign that he did not trust the students to do a good job on their own.

Since the new recorder did all the three steps of capturing, encoding and saving to disk in one operation, the author was curious as to how the recorder was going to fit into the design. For example, the sharing function was using a different encoding and also had no intention of writing anything to disk. When asked about this, Holthe said that the code from his module should simply be copied into a new class and modified to suit the needs of the sharing function. This is what led to the new class diagram seen in figure 6.1 on page 57. The `CGXCapture` class was divided into classes for each intended recipient of the framebuffer, and the `Encoders` and `Decoders` modules were also collapsed into single classes upon the management's request.

The next sections will go through the reasons why the author believes that these decisions were suboptimal to both the students, and the Geelix project as a whole.

Code quality

The new design by Holthe is based on modeling the classes after the intended recipient of their result, rather than after their function. There are several reasons why this is not good software design. For example, in the case of `CGXRecordVS`; the functions that this class performs are to capture a frame, encode the captured data, and feed the result to the network using `CGXShareVS`. The "VS" in the class name indicates that this class is for use with the Video Server only, as is indeed the case. A problem arises whenever we need this data for other reasons. Let us assume that there is a new feature in Geelix that require the same services. We can use the class as it is, but this means that the class name no longer corresponds to where it is being used (VS was only for Video Server, right?), which will undoubtedly cause confusion for readers of the code. Alternatively, we can duplicate the class and give the new one a name corresponding to its new data recipient, but this duplicates code unnecessarily, increasing both program size and maintenance cost. Finally, we can rename the first class into something more general, which fits both recipients, but then every existing class reference has to be updated as well.

The problem gets even worse if the new feature requires only a subset of the services in the class, in which case there is no other choice than to copy it. In all cases, there is some disadvantage whenever the class is being considered for use outside of the context where it was originally put.

A more optimal solution is to model classes after their function, and functions with no direct dependency on each other should be put in separate

classes (or at least separate methods). The clear advantage is that these classes can be reused by any number of other classes without causing name inconsistencies, and the maintenance all happens in one place. In addition, by breaking the functions up into distinct pieces (capture, encode and streaming being distinct), we increase the probability that an unforeseen new class can use a subset of those functions. This is what motivated the original creation of the `CGXCapture` class, a class which has one task: to capture video and audio. It was also the inspiration for making the `Encoders` and `Decoders` hierarchies, to make additions of new encoders simple by providing the same interface to all.

The `CGXRecordVS` class does feature some class reuse, since it uses the external classes `CGXFFmpegEncoder` and `CGXShareVS` for encoding and streaming, but capture is still done internally in the class. A more optimal solution would be to have the separate `CGXCapture` object still present, and have `CGXRecordVS` and `CGXVoiceVS` use this instead. This would minimize the code inside those two classes, hence making maintenance easier.

Thread management and synchronization is also handled internally in the `Capture` classes, which means that this code also has to be duplicated for almost identical scenarios. This is what motivated the creation of the `CGXController` class in figure 5.9 on page 50.

Performance

Apart from the impact on code quality, the new design also has implications for the performance of the application. Due to the duplication of capture code, each of these instances will do their own capturing calls, thus duplicating the expensive operation if more than one class needs the framebuffer. This happens for instance if someone wants to share at the same time that they are recording. It also happens if voice is used while recording or sharing, but in this case, only audio will be captured twice.

We now take a look at figure 6.2 on page 60, where the sequence of calls involved when sharing in the new design can be seen. One can see by the "Busy" response in the diagram that locking, capturing and encoding is skipped entirely if the previous encoding is not complete. This sequence of events actually guarantees that capture and encoding will never run simultaneously, which defeats some of the intended purpose of the thread. The encoding may still run while the game is running, but this is less efficient, because the game will compete for CPU. The capture operation is a memory transfer and hence primarily bus intensive, and for this reason it is better suited to run alongside a CPU intensive thread like encoding. This was the

motivation for designing the sequence seen in figure 5.8 on page 48, where capturing and encoding runs in parallel as often as possible.

Reimplementation

Due to the fact that Gridmedia's dissatisfaction with the design that was not discovered until well into the implementation phase, a lot of time was spent rewriting and adapting the already written code to the new design. This is clearly something that should have been handled better by the students. Originally, we thought that the management was satisfied with us handling the details of the implementation, partly because that had been the case during the fall project the semester before. It is obvious that we were too quick to make this assumption, and we should have discussed the design with Holthe before proceeding with the implementation.

Conclusion

This section has looked at the decisions that were made about the Geelix HUD client design. It explained how the design was originally planned and then revised by Gridmedia. Because the changes were introduced quite late in the process, they did cause considerable amounts of extra work to adapt the existing code to the new design.

It should be noted that the author respects every right that Gridmedia has to change the design to fit their needs. What made the situation frustrating was the time at which the changes were announced, and the management's failure to justify the need for them. It is the author's firm belief that the quality of both the code and final product had been better, had the original design been followed.

9.2 Product

The final Geelix product, the HUD, has certainly made a lot of progress since the project started in January. The HUD now makes quite impressive recordings, even when they are shared with others live. It has been fun to see in the past weeks how newcomers have visited the site, tried Geelix and uploaded clips from their gameplay. Even when watching from the office, one can feel some of the social aspects of sharing gaming experiences. By watching the clips, you realize that others play just like you, and that you in fact have something in common. The ability to take part in these experiences certainly makes the Geelix HUD an exciting application.

If one looks for flaws in the application, the user interface is definitely one of the things that come up. The GUI is still quite "raw", and it needs a real overhaul to behave more intuitively and more like a regular application. The GUI also displays bugs every now and then, both in terms of user interface bugs and compatibility problems. Difficulty moving windows is a classic GUI bug, and the HUD also has problems working correctly on certain games and computers. Now that many of the exciting features are in place technically, the author thinks that these bugs are the last issues "holding Geelix back" before it can be accepted as a mainstream social gaming application.

Bibliography

[1] Mikael Jakobsson and T.L. Taylor. *The Sopranos Meets EverQuest: Social Networking in Massively Multiplayer Online Games*, 2003. `http://hypertext.rmit.edu.au/dac/papers/Jakobsson.pdf`. Last visited: 9 Jul 2007.

[2] Tony Manninen. Interaction Forms and Communicative Actions in Multiplayer Games. *Game Studies*, Volume 3, Issue 1, May 2003. `http://www.gamestudies.org/0301/manninen/`. Last visited: 9 Jul 2007.

[3] John Halloran, Geraldine Fitzpatrick, Yvonne Rogers, and Paul Marshall. Does it matter if you don't know who's talking?: multiplayer gaming with voiceover IP. In *CHI '04 extended abstracts on Human factors in computing systems*, pages 1215–1218, 2004.

[4] Tony Manninen and Tomi Kujanpää. The Hunt for Collaborative War Gaming - CASE: Battlefield 1942. *Game Studies*, Volume 5, Issue 1, October 2005. `http://www.gamestudies.org/0501/manninen_kujanpaa/`. Last visited: 9 Jul 2007.

[5] *Windows Live Messenger (MSN)*. `http://get.live.com/messenger/overview`. Last visited: 9 Jul 2007.

[6] *YouTube - Broadcast Yourself*. `http://www.youtube.com/`. Last visited: 9 Jul 2007.

[7] Jeff Dyck, David Pinelle, Barry Brown, and Carl Gutwin. Learning from games: HCI design innovations in entertainment software. In *Graphics Interface*, pages 237–246. CIPS, Canadian Human-Computer Communication Society, A K Peters, June 2003. `http://www.graphicsinterface.org/proceedings/2003/159/index.html`. Last visited: 9 Jul 2007.

[8] *Playlinc*. http://www.playlinc.com/technology.html. Last visited: 9 Jul 2007.

[9] *Xfire*. http://www.xfire.com/. Last visited: 9 Jul 2007.

[10] *Game Overlay*. http://www.gameoverlay.com/. Last visited: 9 Jul 2007.

[11] *Windows Media Services 9 Series*. http://www.microsoft.com/windows/windowsmedia/forpros/server/server.aspx. Last visited: 9 Jul 2007.

[12] *Darwin Open Source Streaming Server*. http://developer.apple.com/opensource/server/streaming/index.html. Last visited: 9 Jul 2007.

[13] *Helix Server*. http://www.realnetworks.com/products/media_delivery.html. Last visited: 9 Jul 2007.

[14] *VideoLAN*. http://www.videolan.org/. Last visited: 9 Jul 2007.

[15] *VideoLAN - Features*. http://www.videolan.org/vlc/features.html. Last visited: 9 Jul 2007.

[16] *RealVNC (VNC)*. http://www.realvnc.com/. Last visited: 9 Jul 2007.

[17] *Asterisk :: An Open Source PBX and telephony toolkit*. http://asterisk.org/about. Last visited: 9 Jul 2007.

[18] *RFC3550: RTP: A Transport Protocol for Real-Time Applications*, 2003. http://tools.ietf.org/html/rfc3550. Last visited: 9 Jul 2007.

[19] *RFC768: User Datagram Protocol*, 1980. http://tools.ietf.org/html/rfc768. Last visited: 9 Jul 2007.

[20] *RFC2326: Real Time Streaming Protocol (RTSP)*, 1998. http://www.ietf.org/rfc/rfc2326.txt. Last visited: 9 Jul 2007.

[21] *Windows Media Networking Protocol Kit*. http://www.microsoft.com/windows/windowsmedia/licensing/netprokit.aspx. Last visited: 9 Jul 2007.

[22] *SIP Overview*. http://www.telecomspace.com/vop-sip.html. Last visited: 9 Jul 2007.

[23] *H.323 Overview.* http://www.telecomspace.com/vop-h323.html. Last visited: 9 Jul 2007.

[24] Iain E. G. Richardson. *Video Codec Design - Developing Image and Video Compression Systems.* John Wiley & Sons Ltd, 2002.

[25] Iain E. G. Richardson. *H.264 / MPEG-4 Part 10 White Paper: Overview of H.264*, October 2002. http://www.rgu.ac.uk/files/h264_overview.pdf. Last visited: 9 Jul 2007.

[26] Elliot Mebane. *Selecting a Flash 8 video encoder*, August 2006. http://www.adobe.com/devnet/flash/articles/selecting_video_encoder.html. Last visited: 9 Jul 2007.

[27] *MPEG-4 Users Frequently Asked Questions*, February 2002. http://www.m4if.org/resources/mpeg4userfaq.php. Last visited: 9 Jul 2007.

[28] *Windows Media Video 9 Series Codecs.* http://www.microsoft.com/windows/windowsmedia/forpros/codecs/video.aspx. Last visited: 9 Jul 2007.

[29] *The MP3 history.* http://www.iis.fraunhofer.de/fhg/iis/EN/bf/amm/mp3history/mp3history02.jsp. Last visited: 9 Jul 2007.

[30] *GNU General Public License*, June 1991. http://www.gnu.org/licenses/gpl.txt. Last visited: 9 Jul 2007.

[31] *FFmpeg.* http://ffmpeg.mplayerhq.hu/. Last visited: 9 Jul 2007.

[32] *GNU Lesser General Public License*, February 1999. http://www.gnu.org/licenses/lgpl.txt. Last visited: 9 Jul 2007.

[33] *Direct3D Reference.* http://msdn2.microsoft.com/en-us/library/bb172964.aspx. Last visited: 9 Jul 2007.

[34] Julio Sánchez and Maria P. Canton. *Patterns, Models, and Application Development: A C++ Programmer's Reference*, chapter 1. CRC Press, 1997.

[35] *Wireshark: The World's Most Popular Network Protocol Analyzer.* http://www.wireshark.org/. Last visited: 9 Jul 2007.

[36] Zhou Wang, Alan C. Bovik, and Ligang Lu. Why is image quality assessment so difficult? In *IEEE International Conference on Acoustics, Speech, & Signal Processing*, May 2002. `http://www.cns.nyu.edu/~zwang/files/papers/icassp02a.pdf`. Last visited: 9 Jul 2007.

[37] *TeamSpeak*. `http://www.goteamspeak.com/`. Last visited: 9 Jul 2007.

[38] Zhigeng Pan, Xiaochao Wei, and Jian Yang. Geometric model reconstruction from streams of DirectX 3D game application. In *ACE '05: Proceedings of the 2005 ACM SIGCHI International Conference on Advances in computer entertainment technology*, pages 242–245. ACM Press, 2005.

Appendix A

GXVS (Geelix Video Server) stream protocol

This document describes the protocol used by the GXVS server.

The protocol is based on UDP, and is therefore based on individual packets, not a continuous stream. It is also prone to packet loss.

Each packet consists of any number of commands (limited only by maximum packet size), potentially followed by a number of parameters. Each command is a text string on a single line, terminated by a "\n" (line feed) sequence, although receivers should be prepared to receive "\r\n" also (carriage return/line feed). Most parameters are also each on a single line, with some exceptions (such as a media frame, which is binary data). A command is terminated by two "\n" in a row. Most commands are followed by some sort of reply, as described in the table below.

All integers and numbers are in plain text, unless indicated by "<binary>". It is allowed to prefix them with zeros in order to make length calculations easier.

Every packet from the client to the server must begin with either the "LOGIN" command or the "SESSION" command.

Request	Reply
"LOGIN" – Signals a login action by the client • "VERSION 1" • Username <string> • Password <string> • "VIEW"/"SHARE" – Whether the client wants to view a stream or share a stream. • Login ID <integer> – Random number that serves as an ID to separate one login from another. After the login is completed, this ID is discarded. • Only if action was "VIEW": – Session ID <integer> • Only if action was "SHARE" – Video width <integer> – Video height <integer> – Frames per second <integer> – Audio channels <integer> – Audio rate <integer> – Audio sample size in bytes <integer> – Game ID <integer> – Number of custom fields <integer> ∗ Custom fields <string> – These fields are used for arbitrary data that the client wishes to include in the login.	If login successful: • "LOGIN OK" • Login ID <integer> • Session ID <integer> – For "SHARE", this is a new session number, for "VIEW" it is the same as the one given in the login. • Video width <integer> • Video height <integer> • Frames per second <integer> • Audio channels <integer> • Audio rate <integer> • Audio sample size in bytes <integer> • Game ID <integer> • Number of custom fields <integer> • Custom fields <string> – These fields are used for arbitrary data that the client provided in the login. If the login is unsuccessful: • "LOGIN FAILED" • Login ID <integer> • Error message <optional string>

Request	Reply
"SESSION" – Signals that the packet is part of an ongoing session. Both client and server • Session ID <integer>	If the session ID is valid, there is no explicit reply. If it is invalid (only the server will send this): • "SESSION INVALID" • The session ID that was given. • Error message <optional string>

Request	Reply
"ECHO" – Both the client and server can send this and it generates the reply to the right. It serves as a keepalive mechanism.	"ECHO REPLY"

Request	Reply
"FRAME" – Can be sent by both server and client, and signals a media frame. • Flags <single letter> – "v" for video, "a" for audio and "u" for user input. If frame is a keyframe, the letters are uppercase. • Offset <integer> – If the frame is split over more packets, this signals the where this packet's offset starts. • Frame size <integer> – The length of the complete frame. • Timing <integer> – Timing of this particular frame, in milliseconds. Which number is not important as long as successive frames have the correct relative difference (and video and audio meant to be played simultaneously should have the same timing). • Data length <integer> – The length, in bytes, of the binary frame data, without line termination. Remember to make enough room for the terminating "\n\n" in the actual packet though. • Frame <binary> – Frame data	It only makes sense for the sharing user to send frames. Frames from viewers will be silently discarded.

Request	Reply
"REFRESH SESSION" – This can only be sent by a client that logged in using "SHARE". It generates a new session ID.	"NEW SESSION" • New session ID <integer> Both session IDs will be valid for a short period (typically 5–10 seconds) after which the old will be invalidated. The meaning of this command is to use it when permissions change, and then only give the new session ID to viewers that are authorized. The server will not send the "NEW SESSION" reply to anyone else than the command issuer (GXLS is responsible for distributing session IDs to others). For viewers: "PERMISSION DENIED: REFRESH SESSION"

Request	Reply
"REQUEST VIEWERS" – A client that logged in with "SHARE" can issue this command to request a list of current viewers.	"VIEWERS" – A list of current viewers • Number of viewers <integer> • For each viewer: • Username <string> This reply may be sent also without a prior request. The server may send this reply periodically as viewers come and go. For viewers: "PERMISSION DENIED: REQUEST VIEWERS"

Request	Reply
"LOGOUT" – This is sent by the client when it wishes to disconnect. It can also be sent from the server to clients when it wishes to shut down a session.	"LOGOUT OK" – Viewers are simply removed from the session. Sharers cause the session ID to be invalidated immediately.