A real-time streaming games-on-demand system

Conference Paper · September 2008

DOI: 10.1145/1413634.1413648 · Source: DBLP

3 authors, including:

D. Metafas
University of West Attica
32 PUBLICATIONS **132** CITATIONS

SEE PROFILE

Athens Information Technology

in collaboration with

Carnegie Mellon University
Information Networking Institute
(INI)

"A Real-Time Streaming Games on Demand System"

Thesis Report
submitted in partial fulfillment of the requirements
for the degree of

Master of Science
in Information Networking
(MSIN)

by
THEOFILOS KARACHRISTOS

Supervisor: **PROF. DIMITRIOS METAFAS**
(dmeta@ait.edu.gr)

Athens, Greece
April 2007

Abstract

Gaming is not just for kids anymore. With hundreds of millions people playing interactive games on a regular basis today, gaming is rapidly becoming a dominant player in the entertainment industry.

The broadband-based online games business is gaining momentum on a global scale across many device types and its games business looks like the next "killer app". In the business of broadband one key question is how a broadband service provider will tap into subscription revenues generated from the online games business.

Broadband connection drives to a new, digital, “Future Home” as part of a communications revolution, which will affect every aspect of consumers’ lives, not the least of which is the change it brings in terms of options for enjoying entertainment. Taking into account that Movies and Music provided by outside sources were at home long before the Internet and Broadband, the challenge is to invent content consumption patterns of existing and new types of content and services. One such opportunity is realized by the revolution of On-line Games and Games on Demand.

The current Games on Demand solutions proposed by various vendors require from the subscribers to download the whole lot or part of the game and run it locally. Therefore, these solutions are built on the assumption that the subscribers actually possess the required resources in order to be able to run the game. Thus, the subscribers have to worry about acquiring the necessary equipment as well as keeping it up to date to meet at least with the latest games’ minimum requirements.

The present thesis introduces a Games on Demand system that does not require the possession of special equipment by the subscribers. Each game runs on a machine situated in the vendor’s premises and its display is streamed to the user in real time, whereas the user’s reaction is transmitted back and fed into the Game. Since no data are installed or saved on the subscriber’s side, the system saves the subscriber from the purchase and frequent upgrade of a powerful machine as well as it ensures that even users with little or no computer knowledge can subscribe to and use this service.

The system supports games built under the DirectX technology and it does not require any modification of the original games. Therefore, any game can be used in the system on its very first release day. The game’s frames are captured via a process hooking technique and encoded in real time under the MPEG2 format. The encoded frames are encapsulated on a Transport Stream and streamed to the user’s client under the UDP protocol. The user’s actions are captured as raw inputs and streamed back to the game server via UDP, where they are inserted into the mouse or keyboard event queues and passed to the game process automatically by the operating system. This technique enables the continuous flaw of the data and makes the remote control of a game a reality.

Acknowledgements

I would like to thank my supervisor Prof. Dimitrios Metafas for his continuous support and Mr. Dimitrios Apostolatos for his helpful guidance throughout my efforts towards the completion of the present thesis.

Table of Contents

Introduction **1**
1.1 Existent Games on Demand Systems 1
1.2 Proposed System 4
1.3 Report Outline 5
Technologies **6**
2.1 DirectX 6
2.2 MPEG-2 Standard 7
2.2.1 Data Encoding (Compression) 7
2.2.2 Data Transmission 10
Open Source Projects **12**
3.1 Taksi 12
3.1.1 Introduction 12
3.1.2 Capturing Technique 13
3.1.3 Taksi Library Internals 14
3.2 FFmpeg 16
3.2.1 Introduction 16
3.2.2 Compilation under MS Windows 16
3.2.3 Using FFmpeg in a Visual C++ application 18
3.2.4 FFmpeg Library Internals 19
System Architecture **21**
System Implementation **23**
5.1 Client–Server Handshaking 25
5.2 Frame Capturing 25
5.3 Frame Encoding, Encapsulation and Transmission 26
5.3.1 Encoding and Encapsulation 26
5.3.2 Transmission 28
5.4 User Input Capture, Transmission and Parsing 29
5.4.1 Input Capture and transmission 30
5.4.2 Raw Input Parsing 31
Performance Evaluation and Future Work **32**
6.1 Performance Evaluation 32
6.2 Future Work 33
References **35**
Appendix A **36**
FFmpeg Structures 36

Taksi Structures 43
CaptureApp Structures 46
Appendix B 47
FFmpeg Functions 47
Taksi Functions 50
CaptureApp Functions 53
UserApp Functions 55

Chapter 1

Introduction

The term Games on Demand refers to a technology that is very similar to the one named Video on Demand. Just as the Video on Demand technology describes the disposal of video data via networks, the Games on Demand technology implements the same notion from the electronic games' point of view. The games can be downloaded directly from a web site via a network rather than being purchased from a physical store. This method not only makes the purchasing of a game more convenient, but also saves money for both the game developing companies and their customers.

Since their first release, the electronic games were being released in box sets that, apart from the game itself, they also included printed manuals and some additional material, depending on the release versions. Due to the fact that this method added more costs to the game development, recently the companies decided to dispose their games in small packages like the DVD cases and provide the manuals only in electronic form. Also, with the appearance of the CDs and, later, the DVDs, some security algorithms started being used in order to cut the spreading of the piracy phenomenon.

During all these years it was being more and more obvious that the only solution to all of the problems that appear under the physical disposal of the games was to offer them under a completely electronic method. The rapid development of the broadband networks and the continuously falling subscription costs made this method a reality under the Games on Demand technology.

1.1 Existent Games on Demand Systems

Several types of Games on Demand systems have been developed till now, with their main difference being the way in which the game data are disposed to the user. The following paragraphs describe the three most common types.

The most simple type is the one via which the games are actually purchased, just like what happens in a physical store. After the purchase, the buyers download a copy of the game on their PCs and from that time on they are recognized as the sole owners of that specific copy. It is easily understood that strong Digital Tights Management (DRM) algorithms have to be applied in order to thwart piracy methods. Also, taking into consideration that each game has a size of several GBs, the buyer has to wait some time before being able to play the game, even if a broadband connection is present and

without counting the installation time. Regardless of the issues described above, many vendors use this game disposal method, including the AOL, the Yahoo Games, the StarHub, the Direct2Drive and the Steam.

The second type is less popular, but a lot more sophisticated. Under this system the games are disposed via a subscription method, and with just a single subscription one has access to a variety of games. Instead of downloading the whole lot of each game, data chunks of the game are streamed to the subscriber. When a sufficient amount of the game data has been sent to the subscriber, the latter can start playing, while the rest of the game continues being streamed on the background. For the proper streaming of the data a special client application has to be installed on the subscriber's PC, and the streamed data are cached for later use. The data can be deleted in order to free up disk space if needed.

Although less problematic than the previous type, this system is also subject to piracy methods, since the subscriber acquires parts or, in later time, the whole lot of a game. This type of Games on Demand systems was first developed by Exent and is being used by Comcast, amongst other vendors.

The third and more recently developed type of such systems is even more sophisticated. Same as the previous one, a subscription method is followed, under which each subscriber has an unlimited access to a variety of games. Nevertheless, no game data is sent to the subscriber, but only the game's audio/visual output is streamed. In order to achieve this, several Game Servers situated on the vendor's premises run the games, the system captures their audio/visual output and streams it to the subscribers. Clients on the subscribers' side present the data to the subscribers and receive their reaction. The reaction is sent back to the Game servers where they are processed and the procedure goes on in the same way.

It is obvious that under these systems no piracy issues are met, thus no DRMs need to be applied, which tremendously cuts down on costs. Also, the subscribers do not have to posses a PC, since the system can cooperate with any device that understands the streamed audio/visual data and can send back the user's reaction, for example, a set-top-box. Following we present three systems of this type that worth to be mentioned.

G-Cluster

The most well-known launched system that falls into the third category is named G-cluster, developed by the G-Cluster company, which is a member of the Club-iT group. Although this system implements the functionality described above, there is an issue that has to be mentioned. Every game needs to be ported to the system, meaning that

some modifications have to be performed on the game's code, in order for the system to be able to use it. This, of course, means that the vendor has to acquire first each game's source code and then pay a group of programmers to implement the porting. Apart from the delay that this method incurs, which varies from three days to a couple of weeks, it also induces additional costs to the whole procedure. Also, taking into account the fierce competition, a delay of even a few days on the game's disposal by the vendor could be disastrous.

Phantom Game Service

Another system of the same type that was announced several years ago, but has not yet been launched, is the Phantom Game Service. The system started as a customized games-specific PC console designed to use a direct download content delivery service instead of removable media, but after several financial problems, in August 2006 the company changed the project into a client software that runs on personal computers operating Windows XP and Windows XP Media center operating systems. The last announcement was referring to a launching date in the first quarter of 2007, but till now no further information has been released.

Stream Theory

An event that took place in August 2006 and is related with a company named Stream Theory Inc. is indicative of the significance of the Games on Demand field and the Application Streaming field in general. In August 31, 2006 Stream Theory Inc. was granted a U.S. Patent, number 7,096,253, for "Method and Apparatus for Streaming Software". Although the patent refers to a method for streaming applications in general, this also includes the third type of the Games on Demand systems referred above. Prior to the patent grant, the company had filed on May, 2006 a lawsuit against Softricity, AppStream and Exent (companies providing Games on Demand solutions), claiming the three companies were infringing on its patents. The lawsuit was filed just about the same time that Microsoft announced its intentions of acquiring Softricity. As a consequence, on March 19 2007, Endeavors Technologies, a sister division of Stream Theory, executed an agreement to license their patents to Microsoft and its affiliates. All of the events mentioned above show how hot the application streaming and, hence, the Games on Demand fields are.

1.2 Proposed System

The Games on Demand system proposed and implemented under the present thesis is designed based on the principles of the third type that was described above. The system consists of two main parts. The first part is the server side, which includes the game servers (server farm) along with some additional components which are necessary for providing the subscribing service, like a firewall, a billing server etc. The game servers execute the games and perform the audio/visual capture and transmission, as well as they receive the player's reaction and process it properly. The second part is the client (subscriber) side, which includes the devices that are able to decode the received audio/visual data and capture and transmit the user's reaction. These devices can be a PC, a set-top-box, a mobile phone etc. In the middle sits the network, through which the server and client sides communicate with each other. The network can be either a LAN, the Internet, or, preferably, a DSLAN. The following figure shows the overview of the system.

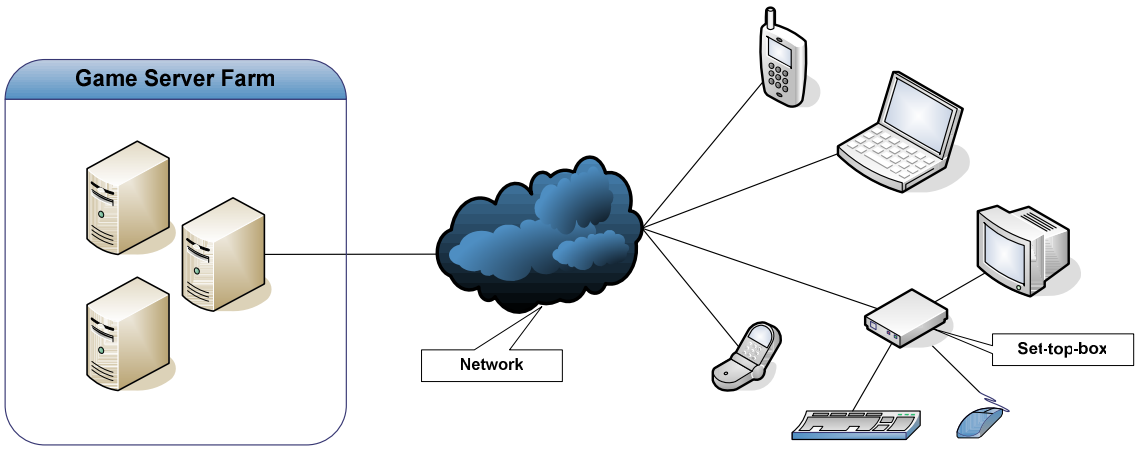

Figure 1: System Overview

Unlike the G-Cluster, the system does not require a game porting, thus, any game can be used in the system in its original form. As mentioned above, this method saves both time and money, since no source code and no additional stuff to do the porting are needed.

It is very important to mention that the audio/visual streaming method is extremely useful not only because of the fact that no game data are transmitted to the users, and, thus, no piracy methods can be applied, but also because it eliminates the platform dependence of the games as far as the users are concerned. So, under this method a subscriber that runs a Linux, Free BSD or MAC OS platform is able to play a game developed for the MS Windows platform. This is a true revolution for the game industry, given the fact the use of non-MS Windows platforms increases each year.

1.3 Report Outline

The present report is comprised of six chapters. The current chapter is an introduction to the Games on Demand systems in general as well as to the system introduced on this thesis. Chapters 2 and 3 provide a description of the technologies used as well as the open source projects whose functionalities were proven extremely useful for the development of the addressed system. Chapter 4 describes the architecture of the real-time streaming Games on Demand system and Chapter 5 analyzes some important implementation aspects of the system. Finally, Chapter 6 presents the results of the system's performance evaluation and proposes some future work that would improve the performance of the introduced system.

Chapter 2

Technologies

2.1 DirectX

DirectX is one of the most popular and well-known 3D game and multimedia applications development kits. As stated in the DirectX SDK Documentation, "it is a set of low-level APIs for creating games and other high-performance multimedia applications. It includes support for high-performance 2D and 3D graphics, sound, and input".

Not much can be written on this report about the internals of this technology, since it is protected by copyright laws. Nevertheless, we can briefly describe the parts that are most important for the better understanding of the system introduced in this thesis.

The DirectX library is divided into three categories/components; the *DirectX Graphics*, also called *Direct3D*, which contains the APIs for the graphics generation, the *DirectInput*, which supports a variety of input devices, including the support for force-feedback technologies, and the *DirectSound*, which comprises the APIs for developing high-performance audio applications.

In this thesis we are mostly interested in the *DirectX Graphics* component and, more specifically, in the technique that is used for the presentation of the scenes. The way that the scenes are presented on the display imitates the operation of the moving pictures. Two main objects are used for this purpose; the front buffer, which contains the scene that is currently being displayed, and the back buffer, which contains the next scene to be displayed. Although only one front buffer can exist, several back buffers can be used, which comprise what is called a swap chain. Whenever the first scene in the swap chain needs to be displayed, it is moved on the first buffer and all of the rest scenes in the swap chain are moved one place along. In this way an application can render scenes continuously without having to care about if and when they are displayed.

One very interesting operation of DirectX is its behavior against the full-screen mode. When an application wants to run in full-screen mode it is given the exclusive ownership of all of the system's adapters, provided that no other application already runs in that mode. This means that, during that period, none of the rest applications can either display any scene or "see" the scene(s) that is(are) being displayed on the system's adapter(s). This is very crucial for the video capturing operation required by

the introduced system and the procedure followed to bypass this obstacle is described in §3.1. It is also important to mention that the presentation of the scenes is performed via the *present* and *reset* functions, which are the ones that the Taksi project exploits to achieve its goals.

2.2 MPEG-2 Standard

MPEG-2 is a multimedia coding standard, which was developed by the Motion Picture Experts Group (MPEG), a committee of ISO/IEC, and it is the MPEG-1 multimedia coding standard successor. It's development aimed to support mainly the applications that concern the digital broadcasting of compressed television. Therefore, compared to the MPEG-1, MPEG-2 induces significant changes such as the support for interlaced video coding and a more efficient coding technique. Its huge success is reflected upon the numerous applications regarding the digital TV broadcasting via cable, satellite and terrestrial channels, as well as upon the fact that it provides the video coding element of DVD video. In the scope of this thesis, we are mostly interested on the data encoding and transmission parts of the standard, which are described briefly in the following paragraphs.

2.2.1 Data Encoding (Compression)

Part 2 of the MPEG-2 standard, also called Video part, defines a video compression codec for interlaced and non-interlaced video signals. By video compression we mean the technique used to compact a video sequence into a smaller number of bits. In MPEG-2 the compression is performed by removing the inherent redundancy of the video sequence's frames in the temporal, special and frequency domains under the use of lossy techniques. In a lossy compression the decoded frames are not identical to the original ones, contrary to the lossless compression in which the original frames can be perfectly reconstructed. The benefit of the lossy compression, as opposed to the lossless one, is a sufficient video quality under the greatest possible compression.

MPEG-2 uses several techniques to achieve the video compression, which, due to their complexity, are not considered necessary to mention in the scope of the current report. Nevertheless, two aspects of the MPEG-2 standard are presented below, which are considered important for the understanding of the present system.

Colour spaces

The name *Colour Space* refers to the method that is chosen to represent the brightness, also called *luminance* or *luma*, and the colour of coloured images. In other words, it is a technique to capture and represent colour information. The most popular colour spaces are the RGB and the YUV, also referred to as YCbCr.

The RGB colour space uses three components to represent the colour information of each picture element, also called *pixel*; a Red, a Green and a Blue (the three additive colours of light). The value of each component represents the proportion of the corresponding colour on the current element and it requires 8 storage bits, which means 24 storage bits for each picture element. This colour space is used by the Colour Cathode Ray Tubes (CRT) and the Liquid Crystal Displays (LCD) to display images.

Due to the fact that the human eye does not distinguish color differences as easy as it does with brightness differences, most of the information provided by the RGB color space is redundant. A wise solution to reduce this redundancy is to separate the luminance from the colour information and represent the former with a greater resolution than the latter. This is what the YUV colour space takes advantage of.

The main idea of the YUV colour space is to calculate a Y (luminance) component as a weighted average of the Red, Green and Blue and represent the colour information as colour difference components, also called *chrominance* or *chroma*. The chrominance of each of the R, G and B is symbolized as Cr, Cg and Cb, respectively, and is defined as the difference of the R, G and B with the luminance component. Given the fact that the outcome of Cr + Cg + Cb is a constant value, we can store only two of the chroma components, therefore, we can store only the Y, Cb and Cr components and each time calculate Cg from the last two.

In the YUV colour space we can use tree different sampling methods, which regard the resolution of each of the Y, Cb and Cr components; the 4:2:0, the 4:2:2 and the 4:4:4 sampling methods. In 4:2:0, each of the Cb and Cr components have half of the Y component's vertical and horizontal resolution. This means that for every 4 luminance samples only 1 sample of each of the chrominance components is used. In the 4:2:2 sampling method, the Cb and Cr components have half the horizontal resolution of Y, but the same vertical resolution, resulting in 2 samples of each of the Cb and Cr for every 4 samples of Y. Finally, in 4:4:4 all of the components have the same resolution. In Figure 2, the sampling differences between these methods are made more clear.

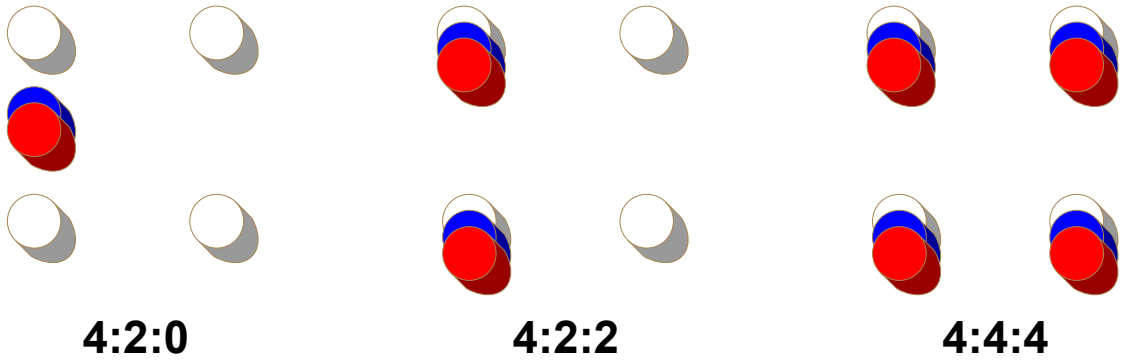

Figure 2: YUV sample formats

It is important to mention that in the 4:2:0 sampling method only 12 bits per pixel are used on average. This is due to the fact that for every four pixels only 6 component samples are used in total; 4 luminance and 2 chrominance. Taking into consideration that we need 8 bits for each sample, we conclude that we need 48 storage bits for every four pixels, meaning only 12 bits per pixel on average, which is half of the storage bits that the RGB colour space requires for the colour information of each pixel. Therefore, if we convert an RGB image into a YUV image, before applying the MPEG-2 encoding, we can achieve an initial 50% compression.

I, P and B pictures

The MPEG-2 standard defines three types of pictures based on the type of the encoding method that is applied on each frame. The first picture is referred to as I-picture and it is intra-coded, meaning that it is not coded differentially with respect to other pictures (coded without prediction). The second picture is called P-picture and it is forward predicted from a previous I or P reference picture. The last type is the B-Picture, which is bi-directionally predicted from both a preceding reference picture and a following reference picture, both of which can be either an I or a P picture. The B pictures are never used for prediction of other pictures and several may appear in a sequence between the I and/or the P pictures.

A sequence of I, P and B pictures is called *Group Of Pictures* (GOP). The first picture of a GOP is always an I picture and, typically, a GOP sequence has a length of 15 pictures. Figure 3 depicts a GOP in which only one B picture appears between each I and/or P pictures. The arrows show the association between the pictures as far as the applied prediction method is concerned.

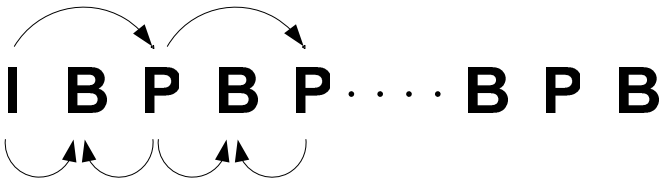

Figure 3: MPEG2 - Group Of Pictures (GOP)

Due to the bi-directional prediction of the B pictures, it is preferable for the decoder to receive the reference pictures before each corresponding B picture. To achieve this, the pictures are encoded and decoded in a different order than they are displayed, which leads to a reduction in buffering requirements. Figure 4 depicts the picture reordering performed on the sequence in Figure 3, along with the associations between the pictures..

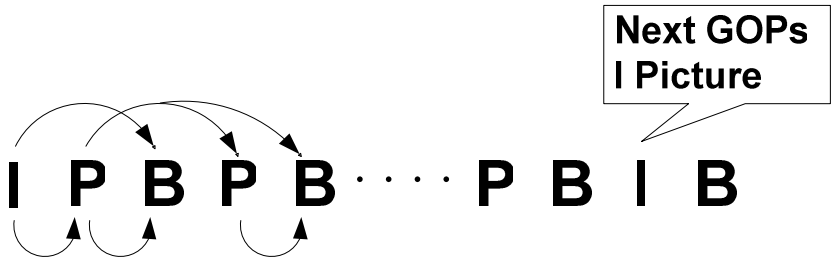

Figure 4: GOP Reordering

2.2.2 Data Transmission

Part 1 of the MPEG-2 standard (MPEG-2 Systems) defines two methods for multiplexing audio and video data, as well as associated information, into streams suitable for transmission. The first method results into a stream called ***Program Stream*** (PS), which is able to carry a single set of audio/visual data, also called a ***Program***, and is used in relatively error-free environments. The PS is also suitable for applications which may involve software processing of system information such as interactive multi-media applications. The second method leads to the creation of a, so called, ***Transport Stream*** (TS), which, contrary to the PS, can combine one or more Programs and is used for environments where errors are likely, such as lossy or noisy media. The TS is also designed for efficiency and ease of implementation in high bandwidth applications. The construction method of the PS and TS streams is depicted on Figure 5.

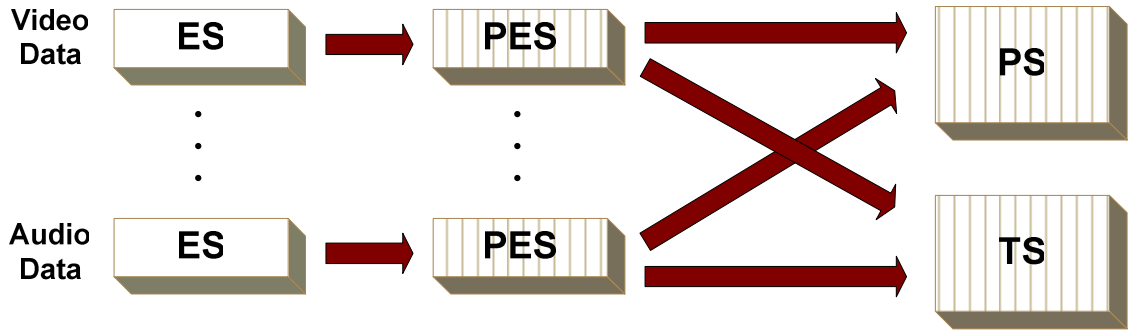

Figure 5: Construction of the Program and Transport Streams

As we observe, for the creation of a PS or a TS we first need a basic element which is referred to as ***Elementary Stream*** (ES). An ES stream is the outcome of the MPEG-2 encoding (compression) of video or audio data and each individual video or audio sequence results in an individual ES. Each of the resulting ESs is packetized to produce a ***Packetized Elementary Stream*** (PES). The PES can include information needed to use the PES packets independently of either the PSs or the TSs, but this is not needed when the PES packets are used to form Program Streams or Transport Streams.

In order to be ready for transmission, one or more PESs must be combined to form PSs and TSs. In the case of a PS, the combined PES packets must have a common time base, meaning to originate from a single Program, whereas in a TS the PES packets can have one or more independent time bases, belonging to different Programs. Both the Program Stream and the Transport Stream provide the necessary information to synchronize the decoding and presentation of the video and audio data as well as to ensure that the data buffers of the encoders will not overflow or underflow. This is achieved via the use of timestamps, which concern both the decoding and presentation of the audio/visual data and also the transmission of the stream itself.

The Program Stream packets may be of variable and relatively great length, whereas the Transport Stream packets have a length of 188 bytes. Also, it may be possible and reasonable to convert between Transport Streams and Program Streams by means of PES packets. Further internal information regarding these streams are not concerned to be worth mentioning in the scope of the present thesis.

Chapter 3

Open Source Projects

3.1 Taksi

3.1.1 Introduction

Taksi is a free open source project for video/screen capturing of 3D graphics applications (such as games). It can capture almost any windows application using DirectX 8 or 9, OpenGL, or GDI and creates an AVI file using any installed VFW codec (XVid, DivX, MSMpeg4, etc.) or still frames in PNG. It was released in 3/8/2004 under the BSD license and since then it is being hosted by the *sourceforge.net*. It is written in C++ and can run over most of the 32-bit versions of MS Windows (95/98/ME/NT 4.0/2000/XP). The project can be downloaded from the site *http://sourceforge.net/projects/taksi*.

The Taksi project includes a dynamic-link library (dll), which implements the video capturing functionality, and a windowed application for the utilization of the dll. In order to bypass the obstacle that DirectX imposes in the full-screen mode, as described in §2.1, the video capturing is performed under a hooking technique, through which the capturing procedure is loaded on the Game process's memory space. In this way, the Taksi library gains access to the game's front and back buffers, which is necessary for the capturing of the frames. Also, it exploits the fact that the DirectX displays the frames with the help of its *Present* and *Reset* functions, with a technique that is explained later on.

In order to load its code on the Game process's memory space, the Taksi library takes advantage of the *SetWindowsHookEx* function, declared in the *winuser.h*. The Taksi's source code is loaded as a computer-based training (CBT) hook procedure, which enables it to be called by the system before a window is activated, created, destroyed, minimized, maximized, moved, or resized, before a system command is completed, before a mouse or keyboard event is removed from the system message queue or before the keyboard focus is set. Although it could check for all of these events, the procedure hooked by the Taksi dll checks only for keyboard focus changes (*HCBT_SETFOCUS* window message). Whenever such an event occurs, the hook procedure performs its tasks on the process that received last the keyboard focus.

3.1.2 Capturing Technique

The Taksi library performs the frame capturing by using a code overwriting technique. It injects its code in the DirectX's *Present* and *Reset* functions at run time by inserting a jump instruction (JMP) at the beginning of these functions. This instruction an unconditional jump to relative address and has a length of 5 bytes (Near Jump). The first byte is the instruction's operation code (opcode - 0xe9) and the rest 4 are the offset, which, in our case, leads to the Taski library's capturing code.

The insertion of the jump instruction is performed by the hook procedure right after it is called due to the change of the keyboard focus. In order to insert the jump instruction, the library simply copies the 5 bytes of the instruction into the (*Present* and *Reset*) functions' 5 starting bytes, after having changed, with the help of the *VirtualProtect* function, their access protection attributes to *PAGE_EXECUTE_READ WRITE*, which enables the write, read and execution operations,.

The starting addresses of the *Present* and *Reset* functions are calculated based on their offsets from the beginning of the *d3d9* library (the loaded DirectX library that contains these functions). Unfortunately, the Taksi library cannot calculate these offsets by itself, since it loads on the game's memory space and the *d3d9* library cannot be loaded twice. Therefore, this is a task that our system has to perform before using the Taksi library and it is described later on. The following Figure depicts the steps performed by the hacking technique described above and a thorough explanation of these steps follows.

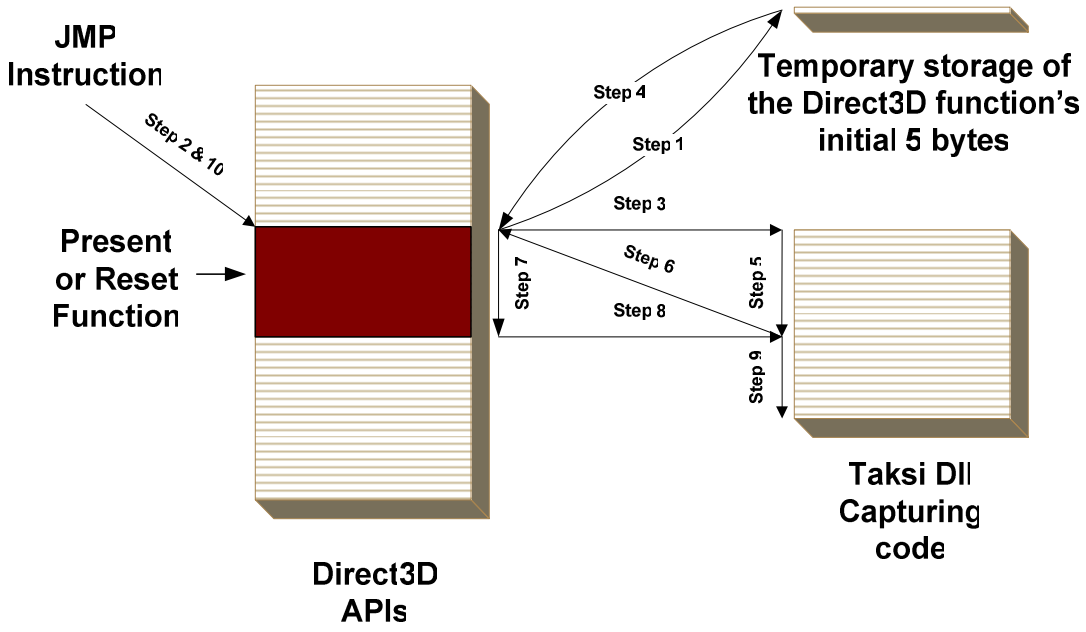

Figure 6: Capturing Technique

The steps 1 and 2 are performed only once by the hook procedure, when it is called due to a keyboard focus event, whereas the steps 3 to 10 are performed whenever the *Present* or *Reset* functions are called, which means that they are performed continuously till the *d3d9* library is unloaded (the execution of the game terminates).

In the first step, the initial 5 bytes of the *Present* and *Reset* functions are stored in a temporary space for later restoration. In step 2, the jump instruction is inserted at the beginning of the functions' code. By the time one of these functions is called, the execution immediately jumps to the Taksi library's capturing code. The first thing that this code does is to restore the function's initial 5 bytes (step 4) and performs the frame capturing as well as some other pre-call tasks, like drawing a recording indication on the current frame's upper left corner (step 5).

After the pre-call tasks, the capturing code calls the *Present* or *Reset* function, and, since the initial 5 bytes have already been restored, this time the function is executed normally (steps 6 and 7). After the function is completed, the execution returns to the capturing code (step 8) and some post-call tasks are performed (step 9). Finally, the capturing code inserts again the 5 bytes of the jump instruction into the beginning of the *d3d9* function and returns.

It is important to mention that the same technique is used with the OpenGL's OGL_WglSwapBuffers function for the support of the OpenGL library.

As stated above, the *Present* and *Reset* functions' offsets inside the d3d9 library must be acquired by the system and passed to the Taksi library. The acquisition is performed via the following way. First, the system loads the d3d9 library and creates an *IDirect3D9* object, which is an interface that includes methods for enumerating and retrieving capabilities of the device. Then, it creates an *IDirect3DDevice9* object, which is an interface that includes methods that perform DrawPrimitive-based rendering, create resources, work with system-level variables, adjust gamma ramp levels, work with palettes, and create shaders. The *Present* and *Reset* functions are two of these methods. Next, the system gets the starting address of the object and calculates the offsets of these functions with the help of a vector table. Finally, it passes these offsets to the Taksi library by storing them in the *sd_Dll* object's *m_nDX9_Present* and *m_nDX9_Reset* members.

3.1.3 Taksi Library Internals

The main interface of the Taksi dll is the *CTaksiDll*, defined in *stdafx.h*. The capture application communicates with the library via this interface, which is inter-process shared, in order to control the library's initialization, hooking, capturing and termination

procedures. The library also exports many other structures, none of which is used by the present system.

The capturing technique described in §3.1.2, is performed by the functions included in the *graphx.cpp*, *graphx_dx8.cpp*, *graphx_dx9.cpp* and *graphx_ogl.cpp* files. The *graphx.h* defines a structure named *CTaksiGraphX*, which is inherited by the structures *CTaksiDX8*, *CTaksiDX9* and *CTaksiOGL*. The last structures include the functions which implement the video capturing for each of the DirectX8, DirectX9 and OpenGL technologies.

Some of the most important functions included in these structures are the *HookFunctions*, *UnhookFunctions* and *GetFrame*, which perform the library's hooking and unhooking on the Game's process and the actual frame capturing, respectively, and are implemented differently for each technology, as well as the functions named *PresentFrameBegin*, which is called whenever a frame is about to be drawn and performs the pre-call tasks mentioned in §3.1.2, and *PresentFrameEnd*, which performs the post-call tasks mentioned in §3.1.2.

The *HookFunctions* function is called by the *AttachGraphXMode*, which is implemented in the same way for the DirectX 8 and 9 but differently for the OpenGL and GDI technologies. This function is called by the function which is declared in the *TaksiDll.h* and has the same name. This last function is the first to be called by the library's Hook Procedure whenever a change in the keyboard focus occurs. Finally, the *RecordAVI_Start* and *RecordAVI_Stop* functions start and stop the frame capturing procedure, respectively.

3.2 FFmpeg

3.2.1 Introduction

FFmpeg is a free open source project that provides a complete solution for recording, converting and streaming audio/visual data. It is released under the GNU Lesser General Public License (LGPL) and right now it is being developed by a group of 15 developers, who continuously and rapidly improve and enhance it. The project first started in 12/6/2000, initially hosted by the *sourceforge.net* and now by the www.*mplayerhq.hu*, and till now it exhibits very high activity. The project's code is provided either via the repository *svn://svn.mplayerhq.hu/ffmpeg/trunk* or as a daily snapshot via the site *http://www.haque.net/software/ffmpeg/svn-snapshots/*.

The FFmpeg project is implemented in pure C language and developed under Linux. Nevertheless, it can be compiled under most operating systems, including Windows. In order to be able to compile it under MS Windows, a tool like MSYS/MinGW or Cygwin is needed. The procedure that must be followed in order to compile FFmpeg with MSYS/MinGW is described later on in this section.

All of the project's functionality is split into three libraries; the ***libavcodec***, the ***libavformat*** and the ***libavutil***. The first library provides a variety of well-known and popular audio/video codecs and the APIs needed for their utilization, the second library, similarly to the first, provides a group of muxers and demuxers and the corresponding APIs, whereas the last library provides some very useful utilities and structures.

Apart from the main executable (***ffmpeg***), the project also includes an experimental streaming server for live broadcast named ***ffserver***, and a media player called ***ffplay***. Also, the compilation of the project can be configured to provide the FFmpeg libraries as static (.lib) or dynamic-link (.dll).

3.2.2 Compilation under MS Windows

The FFmpeg libraries can be compiled under MS Windows via the MSYS/MinGW, which can be downloaded from *www.mingw.org*, or the Cygwin tool, found at *www.cygwin.com*. Given that Cygwin is a very demanding application, in the scope of the present thesis the compilation of the FFmpeg was done via the MSYS/MinGW tool.

MSYS/MinGW installation

Although MinGW runs on top of MSYS, in order to avoid later setups of MSYS, it is preferable to install first the MinGW and then the MSYS tool. Also, since we need the *autoconf*, *automake*, *libtool* and some other packages, we have to also download the msysDTK package from the same site. So, after we download the tools from the web site mentioned above, we install the MinGW on the path *C:/x/msys/1.0/mingw*, where *C* is the partition on which we want to install the tools and *x* is the top-level path we want to use. Then, we install MSYS on *C:/x/msys/1.0*, giving the MinGW path in the post-install script that appears, and, finally, we install the msysDTK package again on *C:/x/msys/1.0*.

In case the individual downloading and installation of the above packages generates errors, there is also the option of the MinGW setup tool, which can be downloaded from various web sites. This tool automatically downloads and installs all of the necessary packages for the correct co-operation of the MinGW and MSYS applications, provided that a network connection is present. The version that was successfully used for the development of the FFmpeg libraries utilized by the present system was the 5.0.3 (*MinGW-5.0.3.exe*).

FFmpeg libraries compilation

Since our system is compiled using Visual C++, in order to be able to use the FFmpeg libraries we have to use their dynamically linked versions (DLLs) and we have to make sure that Visual C++ compatible import libraries are created during the FFmpeg build process. The procedure which one has to follow in order to build and use these libraries follows and it is based on MS Visual C++ 2005.

Provided that we have already installed the MS Visual C++ and the MinGW/MSYS tools, we have to add a call to *vcvars32.bat*, which sets up the environment variables for the Visual C++ tools, as the first line of *msys.bat*. The standard location of the *vcvars32.bat* is *C:/Program Files/Microsoft Visual Studio 8/VC/bin/* and the *msys.bat* is located in *C:/x/msys/1.0/*. So, we just have to add the line "*call "C:\Program Files\Microsoft Visual Studio 8\VC\bin\vcvars32.bat"*" as the first line of *msys.bat*. This method ensures that the compiled FFmpeg libraries will be Visual C++ compatible. To test if everything is correct, we can type *link.exe* on the MSYS shell and if we get a help message with the command line options of link.exe we can be sure that the environment variables are set up correctly and that the Microsoft linker is on the path and will be use by FFmpeg.

The next step is to download the FFmpeg project, if we haven't done already, extract it in a desired path and change to the FFmpeg directory. Before building the project, we first have to create the makefile by typing the command "*./configure --enable-shared --disable-static --enable-memalign-hack*". If we are not interested in the ffserver and ffplay tools that the FFmpeg provides, which is the case in our system, we can disable them by also adding the options "*--disable-ffserver --disable-ffplay*". If everything is ok and no errors were produced, we can proceed with the compilation of the project.

Before building the FFmpeg libraries, it is advisable to modify some lines of the generated *config.mak* file in order to avoid some potential problems with the libraries. The lines to be modified are the ones that include the instructions "*SLIBNAME=...*", "*SLIBNAME_WITH_VERSION=...*" and "*SLIBNAME_WITH_ MAJOR=...*", where "…" corresponds to each variable's value, and must be changed to assign the same value "*'$(SLIBPREF)$(NAME)$(SLIBSUF)'*" to all of them.

Now, it is time to build the libraries by simply typing *make*. If the compilation terminates successfully, the FFmpeg folder's *libavformat*, *libavcodec* and *libavutil* subdirectories should contain the libraries *avformat.dll* and *avformat.lib*, *avcodec.dll* and *avcodec.lib, avutil.dll* and *avutil.lib*, respectively. Since we need only the dynamic linked libraries, we copy the three DLLs to our *system32* directory (usually the *C:\Windows\system32*). On the other hand, if *make* exits with an error, we can type *make clean* to delete all of the intermediate files, run the configure script again with the same options plus the option "--enable-mingw32" and type *make*. In some cases this will solve the problem.

3.2.3 Using FFmpeg in a Visual C++ application

Assuming that we performed all of the above steps and we acquired the desired FFmpeg libraries, we then have to follow the next steps in order to be able to use these libraries in our applications. Before we proceed, we have to make sure that our system has the *inttypes.h* and *stdint.h* include files. If this is not the case, there are several web sites that provide these files for free and we only have to copy them into the Visual C++ include folder, typically the "*C:\Program Files\Microsoft Visual Studio 8\VC\include*". It is also very important to mention that all of the source files of our application must have a *.cpp* extension, otherwise Visual C++ won't compile the FFmpeg headers correctly because in C mode it does not recognize the *inline* keyword.

The first thing we have to do is open the *Project Properties* dialog box. In the *Configuration* box we select *All Configurations*, so that our changes will affect both the debug and release builds. On the *Configuration Properties* we select *C/C++* and then

we select *General* and in the *Additional Include Directories* field we add the complete paths to the *libavformat*, *libavcodec* and *libavutil* subdirectories of out FFmpeg directory. The paths have to be included in double quotes and separated with semicolons. We perform exactly the same addition to the *Additional Library Directories* field in the *Linker/General* option of the *Configuration Properties*.

The next step is to select the *Linker/Input* option and add the names *avformat.lib*, *avcodec.lib* and *avutil.lib*, which have to be separated with spaces, to the end of the *Additional Dependencies*. Then, we select the *C/C++/Code Generation* option, in the *Configuration* box we select *Debug* and we make sure that the *Runtime Library* is set to *Multi-threaded Debug DLL*. Then, we select *Release* and we make sure that the *Runtime Library* is set to *Multi-threaded DLL*.

3.2.4 FFmpeg Library Internals

The functionality of the FFmpeg libraries is build upon five basic concepts; the ***codec***, the ***format***, the ***protocol***, the ***codec parser*** and the ***bitstream filter***. *Codec* is the coder or decoder than can be used for encoding or decoding data. The *format* describes each muxer or demuxer that can be used for multiplexing or de-multiplexing data and, among others, it defines, if needed, the codecs that must be used for encoding or decoding these data. The muxers are defined as output formats (***AVOutputFormat***) and the demuxers as input formats (***AVInputFormat***). *Protocol* (***URLProtocol***) is the structure that defines how the data are to be treated after they have been encoded and muxed or demoded and decoded. For example, the already implemented *file* protocol writes or reads data to/from a file, whereas the *tcp* protocol transmits or receives data via a TCP connection. The ***codec parsers*** and ***bitstream filters*** are of no importance for the current thesis. Nevertheless, it is interesting to mention they are defined and used in a similar way as the formats and the protocols, which enables a uniform treatment of all of these structures.

Each ***AVOutputFormat*** structure can have a ***short name***, a ***long name***, a ***mime type*** and a ***file extension***, all of which must be unique, since they are used for the location of the format. Although not all of these values must be present, apart from the file extension, in order for the location of the format to be easier, it is preferable to have all of them defined. Additionally, valid ***audio*** and ***video codec IDs*** can be defined, both of which can be omitted if not needed. Valid codec IDs are those that have been defined in the FFmpeg libraries, like MP2, MP3, MPEG2 etc. or custom ones that have been registered, as described later on. The format structure also defines three functions, which implement the actual format. These functions are the ***write header***, the ***write packet*** and the ***write trailer*** and perform the operations stated by their names. Some

additional info can also be defined in the format structure, but is of no importance for the scope of this thesis.

Similarly, the ***AVInputFormat*** structure can be given a ***short*** and a ***long name***, for both of which is preferable to be defined, and includes the functions ***read probe***, ***read header***, ***read packet***, ***read close***, ***read seek*** and ***read timestamp***, which implement the format functionality. Since none of the FFmpeg's input formats is being used by the introduced system, there is no need to go into more detail about this format type.

The ***URLProtocol*** structure is smaller than the format structures. It has a ***name***, which is mandatory, and the functions ***url open***, ***url read***, ***url write***, ***url seek*** and ***url close***. The *url open* function performs all of the necessary initializations in order for the protocol to be ready for use, the *url read*, *url write* and *url seek* functions read, write or seek data in the protocol's private data and the ***url close*** does all the necessary cleanup before deactivating the protocol.

The protocol's private data can be a certain value, a structure, a union or any other structured block of memory. In order for every function to have access to these data, a pointer to their starting address is stored as a void pointer in the ***URLContext***, with which the protocol is associated. The retrieval of the data is performed via a simple casting in the private data's type.

Several *formats*, *protocols*, *codecs*, *codec parsers* and *bitstream filters* are already defined within the FFmpeg's libraries. Nevertheless, anyone can define and use a structure of his preference, provided that it follows the right structure and is registered as described below.

In order for any of the *formats*, *protocols*, *codecs* etc. to be available for use, above all they must be registered. In practice, registration is the placement of the object in a linked list for its easiest and quick location and it can be performed in two ways. The first way is to register every single one of the objects by calling the function ***av_register_all.*** Nevertheless, this is true only for the objects that are already defined and implemented in the FFmpeg libraries. The custom defined objects can be registered only with the second way. The second way is to explicitly register the objects we really need by using the functions ***av_register_output_format*, *av_register_input_format***, ***register_protocol***, ***register_avcodec***, ***av_register_codec_parser*** and ***av_register_bitstream_filter***. All of these functions, apart from the *av_register_all*, get as a single parameter the starting memory address of the object to be registered (e.g. av_register_output_format(&mpegts_muxer)).

Chapter 4

System Architecture

The Real-Time Streaming Games on Demand system consists of two main parts; the Game Server and the User Client. The Game Server is the entity that executes each game, performs the video capturing, encoding and transmission, as well as receives and processes user input messages sent by the User Client. On the other side, the User Client receives, decodes and displays the video sequences transmitted by the Game Server and captures and transmits back to the Game Server the user inputs. The two parts communicate with each other via a TCP/IP connection, whereas the video sequences and the user inputs are transmitted via UDP, which is more appropriate for streaming real-time data. The following figure depicts this topology.

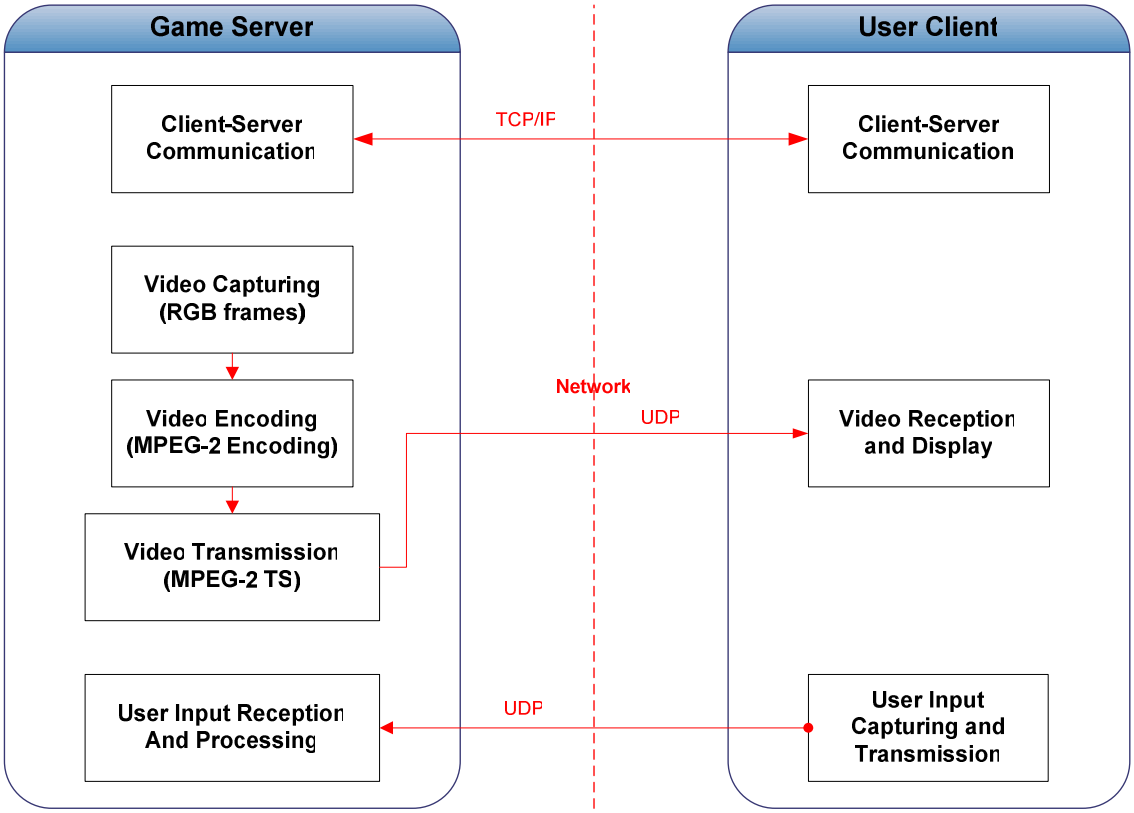

Figure 7: System Architecture

As can be inferred by the above figure, each of the Game Server and the User Client include a UDP server and a UDP client for the interchange of the streaming data. To avoid any later confusion, we will refer to the Game Server's server and client with the names *Video Server* and *User Input Client*, and to the User Client's client and server as *Video Client* and *User Input Server*.

The client-server communication refers to all of the necessary handshaking that the Game Server and the User Client have to perform before starting streaming data to each other. In the currently implemented system, the handshaking protocol involves a simple exchange of the UDP port numbers to which the two parts have to transmit their streaming data. Nevertheless, it can be enhanced to involve a more sophisticated communication procedure, such as game selection or account handling.

In the Game Server side, the video capturing process generates RGB frames that are derived from the game's display under a procedure that is described in paragraph §3.1. The RGB frames are encoded by an MPEG-2 encoder and then inserted in an MPEG-2 Transport Stream. Finally, the MPEG-2 TS is sent to the User Client's Video Client via a UCP connection. The User's Input is received and processed by the User Input Client.

In the User Client Side, the video stream sent by the Game Server is decoded and displayed by the Video Client. The Video Client can be any multimedia player that is able to decode the MPEG-2 data, such as the VLC media player which is used by the implemented system. The user's input is captured and transmitted by the User Input Server to the Game Server's User Input Client via a UDP connection.

The UDP protocol was preferred from the Transmission Control Protocol (TCP), because it offers low-latency transport across IP networks. Nevertheless, due to the fact that it provides no mechanisms for packet loss recovery and synchronization, another protocol must be used to fill this gap. The most appropriate protocol would be the Real-Time Protocol (RTP), which defines a packet structure for real-time data that includes an identifier for the type of the codec used to encode the data, a sequence number for the reordering of the packets and a timestamp for the correct decoding and presentation of the data. Unfortunately, due to technical reasons, an RTP protocol could not be used in conjunction with the UDP protocol and was left as a later improvement.

Chapter 5

System Implementation

The eight components that constitute the Real-Time Streaming Games on Demand system and were presented in the previous chapter, were implemented in three different processes and two threads. The first process (Game Process) performs the capturing of the game's display, which is accomplished through the use of the Taksi project's library. The second process (Game Server's Main Process) encodes each captured frame, encapsulates it in a single Transport Stream and transmits it via UDP to the user's multimedia player (Video Client). These operations are performed with the help of the FFmpeg libraries. The third process (User Client's Main Process) captures the device input on the user's side and transmits it to the Game Server. The transmitted input is received by one of the Game Server's threads (Device Input Client) and fed into the operating system's mouse or keyboard event queue. The client-server communication is performed by the Games Server's second thread (TCP Server) and the User Client's Main Process.

Finally, although any client that is able to decode an MPEG2 Transport Stream can be used for reproducing the received stream, in the scope of the current thesis, the VLC client was used, which is also based on the FFmpeg library. The following figure depicts the system's architecture and a brief explanation of the system's functionality follows.

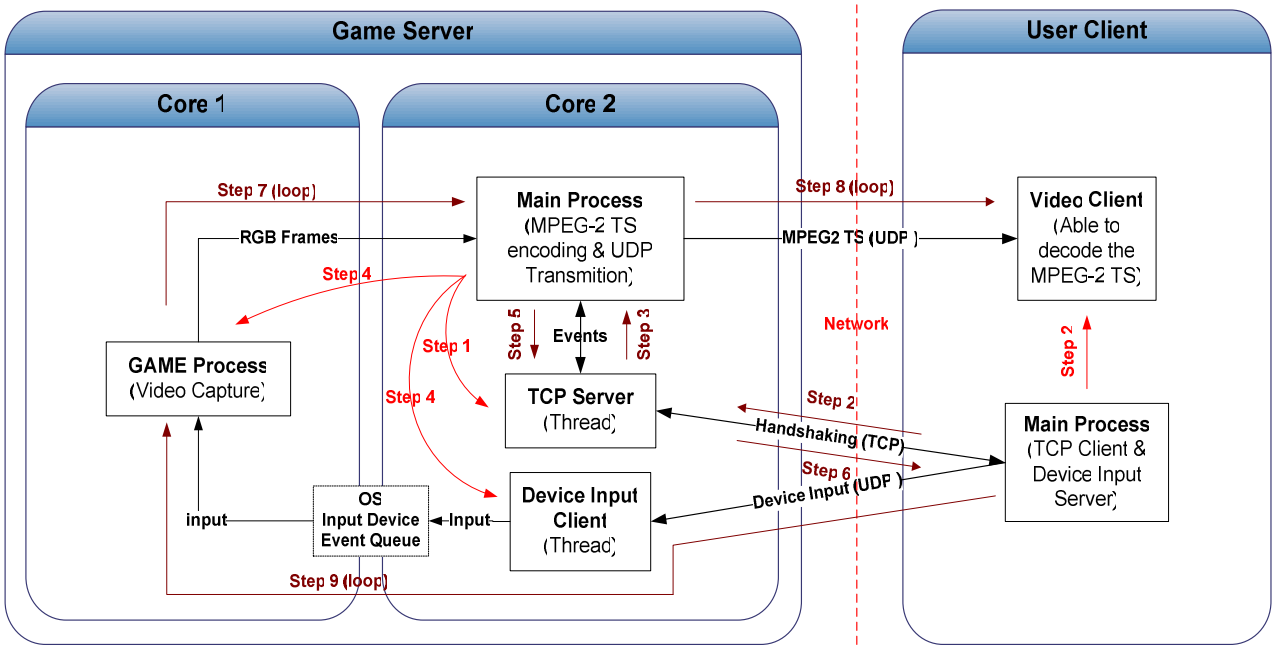

Figure 8: System Components

In the figure above, the red arrows denote process or thread creations, for example, in step 1 the Main Process creates the TCP Server thread. The black arrows show the data flows, for example, the Game Process constantly sends RGB Frames to the Main Process. Finally, the brown arrows refer to the steps during which exchanges of data take place. The word *loop* in steps 7,8 and 9 means that these steps after a certain point are performed continuously till the termination of the client-server communication.

As can be inferred by Figure 8, the whole procedure starts with the execution of the Game Server application (named as *CaptureApp.exe*). One of the first tasks that the application performs is the creation of the TCP Server thread (Step 1), at which point the application gets into a waiting state for client requests. Since the Game Server is up and running, the User Client application (named *UserApp.exe*) can also get started. The application's Main Process launches the Video Client (Step 2 – red arrow) under a state of listening on a UDP port for incoming video data and sends to the TCP Server (Step 2 – brown arrow) the Video client's UDP port number. By the time the TCP Server receives the UDP port number, it assumes that a User Client requested a connection and notifies the Main Process (Step 3). Then, the Main Process creates the Game Process and the Device Input Client thread (Step 4) and, in turn, notifies the TCP Server for the success of the tasks (Step 5). Same as the Video Client, the Device Input Client starts under a state of listening on a UDP port for incoming input device data and the UDP port number is transmitted by the TCP Server to the User Client's Main Process (Step 6).

By this time, a continuous execution of the steps 7, 8 and 9 begins, which goes on till the Game Process's termination or the termination of the client-server communication (termination of the UserApp.exe). It is important to clarify that although these steps are illustrated as sequential operations, in fact they are executed in parallel. The different step numbers were inserted only to mention that these operations are independent of each other.

Figure 8 also depicts the fact that the system takes advantage of the dual core technology. If a dual core machine is present, the system loads the game's process, which also includes the video capturing process, on the first core and the rest of the processes and threads (Main process, TCP server and Device Input client) on the second core. In this way we achieve a better utilization of the cores by avoiding the execution of two heavy operations on the same core, which would induce a substantial overhead.

Finally, two INI files are used, named *CaptureApp.ini* and *UserApp.ini*, for each one of the *CaptureApp* and *UserApp* applications, for the initialization of some important variables. Also, two msi files, located in the CaptureAppInstaller and UserAppInstaller subfolders, are generated by the project for the system's installation.

5.1 Client–Server Handshaking

In order to get things started, a TCP server is implemented on the game server's side and a TCP client is implemented on the user's side. As expected, the TCP server starts its execution in a waiting for TCP connection requests mode. At the beginning of the communication process, the TCP client starts the user's Media Player in UDP mode and sends a request to the TCP server. The TCP server receives the request and notifies the main process to start the UDP client, which receives raw input events, as well as the game process. By the time the UDP client is up, the TCP server sends a reply to the TCP client to begin sending device input events via UDP. For simplicity, the messages commuted between the TCP Server and the TCP client are just the port numbers of each side's UDP client. It is obvious that this is a very simple handshaking protocol and must be improved in order to include a more sophisticated client-server communication.

5.2 Frame Capturing

As mentioned above, the frame capturing is performed via the Taksi dynamic-linked library. In order to fit the Games on Demand system's needs, some modifications had to be applied on the library. First of all, the creation of the AVI file had to be disabled. Unfortunately, not all of the parts of the code that related to the AVI file creation could be changed, but only those that perform the writing of the AVI data to a file. Nevertheless, this change improved the system's performance a lot, due to the reduction of the disk I/O operations. Another modification that was applied to the Taksi library was a slight change to the window messages that the library sends to the application, in order for the system to have a more strong control of the library.

Also, a buffer for the storage of the intermediate captured frames was added, which was considered necessary for the correct synchronization of the Capturing Process and the Encoding and Transmission Process. Additionally, the structure *CTaksiDll*, which is defined in the *stdafx.h* file and is shared by all of the processes, was changed to support some additional functionality. Also, the functions *InitMasterWnd* and *HookCBT_install* where modified in order for the library to hook on a certain process and not on any of the processes that run on the desktop, which is the case with the original library. Finally, all of the library's logging operations where disabled to further reduce the unneeded disk I/O operations.

Note that the modifications on the Taksi library can be found with simple searches in the entire solution, since each block of added or modified code is enveloped between the comments "My Code Start Here" and "My Code Stops Here" and each block of

disabled code is enveloped between the comments “Commented Out Code Start” and “Commented Out Code end”.

5.3 Frame Encoding, Encapsulation and Transmission

The encoding, encapsulation and transmission of the frames is achieved through the functionality provided by the FFmpeg libraries. For our purposes, we are using the existent output format named “mpegts” to encode the frames in MPEG2 format and encapsulate them in a Transport Stream. We also define our own UDP protocol for the immediate transmission of the encoded frames, in a way described below. In order to be able to use these objects, we first have to register them as stated in §3.2.1.

The functions performing these operations are declared and defined in the system’s files ***FFmpegFunctions.h*** and ***FFmpegFunctions.cpp***.

5.3.1 Encoding and Encapsulation

As mentioned above, we encode and encapsulate the frames with the help of the “mpegts” output format, which uses the ***mp2*** codec to encode audio data and the ***mpeg2video_encoder*** codec to encode video, as expected. It also defines the functions, ***mpegts_write_header***, ***mpegts_write_packet*** and ***mpegts_write_end***, which perform the encapsulation of the frames in the TS stream, as stated by their names. These functions are called internally by the ffmpeg library whenever we call the functions ***av_write_header***, ***av_write_frame*** and ***av_write_trailer***, respectively. The encoding of each frame is done via a call to the function ***avcodec_encode_video***.

The encoding and encapsulation of the frames is split in three parts. The first part is the initialization of the necessary FFmpeg structures, performed by the ***FFmpegInit*** function. The second part performs the actual encoding and encapsulation of the frames and is implemented in the function ***FFmpegEncodeFrame***, which is called each time a new frame is available for encoding. The last part, performed by the ***FFmpegClose*** function, writes (transmits) the trailer of the TS stream and releases all of the FFmpeg objects and their allocated memory.

Initialization

The FFmpeg structures that we need to initialize and use during the encoding procedure are, in order of hierarchy, the ***AVFormatContext***, the ***AVOutputFormat***, the ***AVStream,*** the ***AVCodecContext***, the ***AVCodec*** and the ***AVFrame***.

The *AVFormatContext*, is the main structure and, among others, it contains the *AVOutputFormat* and the *AVStream* structures. The *AVOutputFormat* is the output format we mentioned above, named "mpegts". The *AVStream* is the object that defines the information about the output stream and it also includes the *AVCodecContext*. The *AVCodecContext* structure contains all the general information about the codec (e.g. codec type, bit rate, frame rate etc.) and, in turn, it contains the *AVCodec* structure that defines the specific codec to be used (e.g. mpeg2video_encoder). The last structure, named *AVFrame*, contains the data and general info about the frame to be encoded.

The initialization of these structures must be carried out under a specific sequence. First, we initialize the *AVFormatContext* by calling the function ***av_alloc_format_context***. Then, we define the *AVOutputFormat* structure inside the *AVFormatContext*, which can be performed with two different ways. The first way is to call the ***guess_format*** function and let the FFmpeg library guess the input or output format type we want to use. This function gets three arguments and we can either define only one, two or all of them, by specifing as NULL those we don't want to define. For example, if we define the first argument as "*mpegts*" and/or the second to be a filename with extention ".ts", then the function returns the address of the *mpegts_muxer*. The second way is to explicitly assign the address of the *mpegts_muxer* by giving "*&mpegts_muxer*".

The next structure to be initialized is the *AVStream*. By calling the function ***av_new_stream*** we initialize the structure and assign it to the *AVFormatContext*. Then, we initialize the *AVCodecContext* and assign it to the *AVStream* structure that was just created. The next step is to set some parameters of the *AVFormatContext* via a call to the ***av_set_parameters*** function. This function allocates the necessary memory blocks to the *AVFormatContext* structure and calls the format's ***set_parameters*** function, if one is defined. The ***av_set_parameters*** is called only when we are using an output format and it must be called even if the ***set_parameters*** function is not defined by the format.

After all of the above initializations, we must "open" the codec we want to use via the ***avcodec_open***, which initializes the *AVCodec* structure and assigns it to the *AVCodecContext* structure. In order to avoid passing explicitly the codec structure, we can use the ***av_find_encoder*** function, since we refer to an encoder, to imply it via the codec id contained in the *AVStream* object and pass the result to the *avcodec_open* function. The next step is to call the ***avcodec_alloc_frame*** function to allocate the necessary memory and assign the default parameters to the *AVFrame* structure and call

the ***avpicture_fill*** to specify the pixel format and the resolution of the frame as well as to assign a buffer for the frame's data. Since the buffer must be of at least the same size as the size of the frame's data, we can use the function ***avpicture_get_size*** to get the frame's size given the pixel format and the resolution.

The last steps of the initialization involve the opening of the (custom) protocol, as described in §3.2.2, the addition (transmission) of the TS stream's header via a call to the ***av_write_header*** function and the memory allocation to the buffers that will store each frame's YUV data.

Encoding and Encapsulation

Provided we have done all the necessary initializations, the ***FFmpegEncodeFrame*** function is called each time a new frame is available to be encoded. Since the game's frames are in RGB24 pixel format and we are interested in frames in YUV4:2:0 pixel format, we first have to convert each frame. This is done via the ***Rgb24ToYuv420p*** function, whose code was borrowed from the FFmpeg's ***rgb24_to_yuv420p***, declared in the file named ***videogen.c***, which includes codec testing code. Alternatively, we can use the FFmpeg's ***img_convert*** function declared in the ***avcodec.h*** file, provided that we declare a second *AVFrame* object, which will contain the RGB frame's data. Nevertheless, it was not considered necessary to do so, because it added more complexity and delay.

The encoding of each frame is accomplished through a simple call to the ***avcodec_encode_video*** function. In order to encapsulate the frame into the TS stream, we declare an AVPacket object, which we initialize via the ***av_init_packet*** function, in which we assign, among other info, the encoded frame's data and size. The packet can be inserted into the TS stream through a call to the **av_write_frame** function.

5.3.2 Transmission

The transmission of the encoded frames is carried out automatically by the FFmpeg library based on the protocol we define during the FFmpeg initialization. As presented in §3.2.1, there are some protocols already implemented in the FFmpeg library, including a UDP and an RTP protocol. Nevertheless, due to some technical reasons, we decided to define and use our own simple UDP protocol, which is implemented in the ***MyUDP.h*** and ***MyUDP.cpp*** files.

The ***MyUDP*** protocol defines three functions, the ***MyUDPOpen***, which initializes the protocol, the ***MyUDPWrite***, which transmits the given data, and the ***MyUDPClose***, which terminates the protocol. It also uses a structure named ***MYUDPCONTEXT***, which stores the UDP socket's id, the destination's address and port number as well as a delay value which was added for testing purposes.

In order to instruct the FFmpeg library to use this protocol, we must call the ***url_fopen*** function passing as parameters the ***ByteIOContext*** structure, which stores the protocol's info, referenced by the *AVFormatContext*, the ***url*** to be used by the protocol and the ***URL_WRONLY*** specification, which defines that the protocol is used only for writing (transmitting) data. The url can be of the form "*myudp://host:port?delay=0*", where *host* and *port* are the destination's address and port number and *delay* is a value specifying a delay in the transmission of the data and can be omitted. If the delay option is not defined, the url takes the form "*myudp://host:port*". Suffice to say that this protocol was constructed to perform only unicast transmissions, but it can be easily enhanced to also perform other types of transmissions.

The use of a protocol via the FFmpeg library is the most direct way of transmitting the frames by the time they are encoded and encapsulated. Any other way would demand the use of a separate UDP server and a communication protocol between this and the encoding module, which procedure would incur an additional delay.

5.4 User Input Capture, Transmission and Parsing

Although initially planned, the utilization of a set top box for the interaction with the user was not rendered possible. Due to this reason, we had to create a process for capturing the user's input via the Windows XP operating system.

The user's input can be captured via the win32 API in two different ways; through legacy messages or through raw input messages. If the first technique is used, then for every input a different (legacy) message is sent to the application, which means that the application has to take into consideration every possible message. On the other hand, under the second technique, the same message (***WM_INPUT***) is sent to the application for any type of input event and the actual (raw) input event is stored in the message's *l* parameter.

Under the scope of this thesis, it was considered more preferable to use raw device inputs for the user's reaction and transmit them as is to the game server via the UDP protocol. By the time the game server receives a raw input it transforms it into simple input form, due to the reason described in §5.4.2, and inserts it into the mouse or

keyboard event queue. From this point on, the operating system has the responsibility of retrieving the input event from the corresponding queue and passing it to the game process. This technique enables us to pass input events to the game process even if the latter uses the *Direct Input* APIs.

5.4.1 Input Capture and transmission

In order to be able to receive raw device input we must first register the devices we want to receive input from and define a callback function that will receive the *WM_INPUT* messages from the operating system. In order to be able to use such a function we have to create a window and assign this function as the window's callback procedure.

The registration of the devices is being done via the ***RegisterRawInputDevices*** function, which takes as parameter an array of ***RAWINPUTDEVICE*** structures, each one of them corresponding to a raw input device we want to register. In each of these structures we define the top level collection ***usage page*** for the corresponding device, the top level collection ***usage*** for the corresponding device, some ***flags***, which specify how to interpret the information provided by the *usage page* and *usage* values, as long as a ***handle*** to the target application.

For all of the input devices the *usage page* has the value 0x01, whereas the exact device we refer to is defined by the *usage* value. Therefore, a *usage* value of 0x02 or 0x06 refers to a mouse or a keyboard, while the values 0x04, 0x05 and 0x07 refer to a joystick, a game pad and a key pad, respectively.

The *flags* can take one or more of 8 different values. For our purposes we used only two of them; the ***RIDEV_NOLEGACY***, which prevents the corresponding device from generating legacy messages, and the ***RIDEV_INPUTSINK***, which enables our application to receive input events even if it does not have the keyboard focus. In order for the second specification to work correctly we have to specify the handle to the target application to be the handle to our application.

By the time the system receives a raw input message, it retrieves the ***RAWINPUT*** structure from the message's *l* parameter via a simple casting and transmits the whole structure to the game server via UDP.

5.4.2 Raw Input Parsing

For inserting the received input events into the mouse or keyboard input stream, we use the ***SendInput*** function, which takes as parameters an array of ***INPUT*** structures, the number of the structures in the array, as well as the size in bytes of the *INPUT* structure. Since the game server receives *RAWINPUT* structures from the user's client, it is obvious that we have to transform these structure into the corresponding *INPUT* structures. Fortunately, these structures are similar to each other, so the transformation is rather simple.

Each *RAWINPUT* structure includes a ***RAWINPUTHEADER*** structure whose *type* value specifies the source of the input event (mouse, keyboard or other device). Based on this info, we assign the corresponding value to the *INPUT* structure's ***type*** member and parse the rest of the *RAINPUT* structure by parsing the correct member of the ***data*** union (*mouse*, *keyboard* or *hid*) and assigning the correct value on the members of the *INPUT* structure's corresponding member (*mi*, *ki* or *hi*).

In the scope of the system described in this report, the main focus has been given on the precise transformation of the mouse raw input events. Nevertheless, a partial transformation of the keyboard raw input events is also implemented, which has to do basically with the non-system keys.

Chapter 6

Performance Evaluation and Future Work

6.1 Performance Evaluation

According to several studies, in order for the gameplay to be smooth and without any jitter, the delay between the transmission of the user's reaction and the display of the corresponding frame, also called round trip delay, must be less than 200 milliseconds. If the round trip delay is greater than 200 msec, the user's reaction is displayed with an irritating delay that makes the gameplay very difficult, especially in games where speed is a very important factor, like the First Person Shooting and the Real Time Strategy games.

Since all of the required processing is performed on the server side, apart from the user's reaction and transmission which need little processing and are performed on the client's side, almost all of the delay is related to the processing power of the game server and the characteristics of the network. Thus, the system was tested in two different setups, which involved two different servers and two types of networks.

The first setup included an Intel Pentium(R) D CPU processor, which comprises of two cores with 2,8GHz speed each, and a 2GB RAM. Also, the server and the client where connected with each other via a direct connection (crossover cable). Under this setup, the system exhibited a round trip delay of approximately 600msec.

The second setup involved an Intel(R) Core(TM) 2 CPU 6700 processor, which comprises of two cores with 2,66GHz speed each, and a 2GB RAM. The server and the client where connected with each other via a switch under a LAN network. In this setup, the system exhibited a round trip delay of approximately 600 to 800 msec.

We observe that in both setups the round trip delay is more than 3 times the preferable value (200msec). Although this outcome seems disappointing, on the contrary, it is much better than the expected one. Taking into consideration the fact that the open source projects that are used by the system are not fully optimized for the tasks they perform, a round trip delay of 600 msec is actually promising. It shows that a future optimization of the code could lead to a significant decrease.

Besides, a careful observation on the server's and client's monitors reveals a very interesting characteristic of the system's behavior. Each frame that shows a user's

reaction is displayed on the server's monitor almost immediately after the occurrence of the event, whereas it is displayed with a delay on the client's monitor. Given the fact that the frame capturing takes place almost concurrently with the display of each frame on the server's side, as described in §3.1, we conclude that most of the delay is produced by the FFmeg libraries.

After a thorough examination of the FFmpeg's code, it was revealed that the implemented MPEG TS multiplexer does not create the Transport Stream in the correct way. The developers of the FFmpeg libraries used the earlier implemented MPEG PS multiplexer as a basis for the development of the TS multiplexer, but did not change it in order to follow the standard. Practically, this means that the existent client applications, like the VLC player, are mislead in thinking that they read a TS, although what they read is a PS. Thus, the timestamps contained in the TS make the applications think that they receive unordered and late frames and display the frames in an incorrectly way, which is perceived by the user as jitter and delay. Therefore, it is expected that the round trip delay will be dramatically reduced if the FFmpeg's MPEG TS multiplexer is modified in order to really implement the standard.

Note: Due to the problem described above, if the VLC player shows only a still image of the game, we have to perform the following steps in order to be able to see all of the received frames. From the Menu bar we select Settings->Preferences or we type Ctrl+s. On the new window, we select the Video option, we check the "*Advance options*" box on the bottom right corner and we uncheck the "*Drop late frames*" and "Skip frames" options.

6.2 Future Work

Before suggesting some potential improvements and enhancements to the system, it is important to mention that a lot of work needs to take place in order for this system to be regarded as a complete Games on Demand solution. Having that in mind, we suggest some future work that can be performed in order to reduce the round trip delay and also enhance a little bit the system's functionality.

The most important work that must be done before anything else is the correction of the FFmpeg's MPEG TS multiplexer, as described above. Since the TS multiplexer is the reason for most of the additional round trip delay, its correction is essential for the further improvement of the system.

The second critical improvement is to take out all of the Taksi library's useless code. Since the library was developed to be used by a certain application, many things have to be changed in order to completely fit the needs of the present system. For example, the

Taksi application, and, therefore, the Taksi library, saves the captured video on an AVI file, which is a useless operation for the present system, since it immediately transmits the captured frames to the client side. Although the most critical parts of this operation have been disabled, much of the initial functionality is still present, due to the tight dependencies inside the Taksi library's code.

For the transmission of the data in both ways between the server and the client the system uses the UDP protocol. A simple improvement would be to add an RTP communication over the existent UDP, in order to ensure more proper transmission of the streamed data and reduce the packet losses.

A fourth improvement to the system would be to replace the existent chunks of code that perform the RGB to YUV image conversion with the FFmpeg's function called *img_convert*. Besides, the system's code that implements this functionality was inspired by that function, thus, its use would lead to a more complete utilization of the FFmpeg libraries.

A very important addition to the system would be the support of audio capturing. Till now the Taksi library only supports video capturing, thus adding a support for audio capturing would be a hard and time consuming procedure. Therefore, this addition was left as a potential future work, unless, or until, the developers of the Taksi project themselves decide to add it to the Taksi library.

Further additions could be a set-top-box support for the client's side and also for platforms other than the MS Windows, e.g. Linux, MAC OS, Free BSD etc. Also, a quad processor support would further boost the system's performance and the addition of a GUI would make the user-to-system communication easier and more appealing.

A last, but not least, potential future work would be to test the system under the DirectX 10. In this version several security issues that existed in the previous versions have been fixed, therefore it is not certain that the hack that the Taksi library uses for capturing the frames will function correctly. Of course, this is a job that one would expect from the developers of the Taksi project, but, still, a test would reveal whether such a change is necessary.

References

1. I. E. G. Richardson, *H.264 and MPEG-4 Video Compression*, John Wiley & Sons, 2003.
2. ISO/IEC 13818, Information technology: generic coding of moving pictures and associated audio information, 1995 (MPEG-2).
3. ISO/IEC 14496-2, Coding of audio-visual objects – Part 2: Visual, 2001.
4. ISO/IEC 14496-1, Information technology – coding of audio-visual objects – Part 1: Systems, 2001.
5. Colin Perkins, *RTP: Audio and Video for the Internet*, Addison Wesley, 2003
6. IETF RFC 1889, RTP: A transport protocol for real-time applications, January 1996.
7. Taksi home page, http://www.menasoft.com/taksi/.
8. Taksi SourceForge page, http://sourceforge.net/projects/taksi/.
9. FFmpeg home page, http://ffmpeg.mplayerhq.hu/.
10. FFmpeg SourceForge page, http://sourceforge.net/projects/ffmpeg/.
11. VLC Media Player - home page, http://www.videolan.org/.
12. Moving Picture Experts Group (MPEG) home page, http://www.chiariglione.org/mpeg/.
13. Microsoft Developer Network (MSDN) site, http://msdn2.microsoft.com/.
14. Microsoft DirectX SDK documentation for C++, http://msdn2.microsoft.com/en-us/library/bb219737.aspx.
15. Gamedev.net site, http://www.gamedev.net/.

.

Appendix A

In this appendix we present and provide a brief description of some important structures.

FFmpeg Structures

A.1.1 AVCodec

Declared in *avcodec.h* (libavcodec). Used to define a CODEC, like the *mpeg2video*.

```
AVCodec {
    const char *name;
    enum CodecType type;
    enum CodecID id;
    int priv_data_size;
    int (*init)(AVCodecContext *);
    int (*encode)(AVCodecContext *, uint8_t *buf, int buf_size, void *data);
    int (*close)(AVCodecContext *);
    int (*decode)(AVCodecContext *, void *outdata, int *outdata_size, uint8_t *buf, int buf_size);
    int capabilities;
    struct AVCodec *next;
    void (*flush)(AVCodecContext *);
    const AVRational *supported_framerates;
    const enum PixelFormat *pix_fmts;
}
```

Example:

```
AVCodec mpeg2video_encoder = {
    "mpeg2video",
    CODEC_TYPE_VIDEO,
    CODEC_ID_MPEG2VIDEO,
    sizeof(MpegEncContext),
    encode_init,
    MPV_encode_picture,
    MPV_encode_end,
    mpeg_decode_frame,
    .supported_framerates= ff_frame_tare_tab+1
    .pix_fmts= (enum PixelFormat[]){PIX_FMT_YUV420P, PIX_FMT_YUV422P, -1}
    .capabilities= CODEC_CAP_DELAY,
}
```

A.1.2 AVCodecContext

Declared in *avcodec.h* (libavcodec). Used to store the context of a CODEC.

```
AVCodecContext {
   AVClass *av_class;
   int bit_rate;
   […]
   AVRational time_base;
   int width, height;
   int gop_size;
   enum PixelFormat pix_fmt;
   […]
   int max_b_frames;
   […]
   struct AVCodec *codec;
   […]
   char codec_name[32];
   enum CodecType codec_type; /* see CODEC_TYPE_xxx */
   enum CodecID codec_id; /* see CODEC_ID_xxx */
   […]
   int has_b_frames;
   […]
    int profile;
   […]
   int partitions;
#define X264_PART_I4X4 0x001  /* Analyse i4x4 */
#define X264_PART_I8X8 0x002  /* Analyse i8x8 (requires 8x8 transform) */
#define X264_PART_P8X8 0x010  /* Analyse p16x8, p8x16 and p8x8 */
#define X264_PART_P4X4 0x020  /* Analyse p8x4, p4x8, p4x4 */
#define X264_PART_B8X8 0x100  /* Analyse b16x8, b8x16 and b8x8 */
   […]
}
```

A.1.3 AVFormatContext

Declared in *avformat.h* (libavformat). Used to store the context of an I/O format.

```
AVFormatContext {
   const AVClass *av_class;
   struct AVInputFormat *iformat;
   struct AVOutputFormat *oformat;
   void *priv_data;
   ByteIOContext pb;
   unsigned int nb_streams;
   AVStream *streams[MAX_STREAMS];
   char filename[1024];
   int64_t timestamp;
   char title[512];
   char author[512];
   char copyright[512];
   char comment[512];
   char album[512];
```

```
    int year;
    int track;
    char genre[32];
    int ctx_flags;
    struct AVPacketList *packet_buffer;
    int64_t start_time;
    int64_t duration;
    int64_t file_size;
    int bit_rate;
    AVStream *cur_st;
    const uint8_t *cur_ptr;
    int cur_len;
    AVPacket cur_pkt;
    int64_t data_offset;
    int index_built;
    int mux_rate;
    int packet_size;
    int preload;
    int max_delay;
#define AVFMT_NOOUTPUTLOOP -1
#define AVFMT_INFINITEOUTPUTLOOP 0
    int loop_output;
    int flags;
#define AVFMT_FLAG_GENPTS       0x0001
#define AVFMT_FLAG_IGNIDX       0x0002
    int loop_input;
    unsigned int probesize;
    int max_analyze_duration;
    const uint8_t *key;
    int keylen;
}
```

A.1.4 AVFrame

Declared in *avcodec.h* (libavcodec). Used to store an audio/video Frame.

```
AVFrame {
    uint8_t *data[4];
    int linesize[4];
    uint8_t *base[4];
    int key_frame;
    int pict_type;
    int64_t pts;
    int coded_picture_number;
    int display_picture_number;
    […]
    int reference;
    […]
    int interlaced_frame;
    […]
}
```

A.1.5 AVInputFormat

Declared in *avformat.h* (libavformat). Used to define an input format (decoder and demultiplexer), like the *mpegts*.

```
AVInputFormat {
    const char *name;
    const char *long_name;
    int priv_data_size;
    int (*read_probe)(AVProbeData *);
    int (*read_header)(struct AVFormatContext *, AVFormatParameters *ap);
    int (*read_packet)(struct AVFormatContext *, AVPacket *pkt);
    int (*read_close)(struct AVFormatContext *);
    int (*read_seek)(struct AVFormatContext *,
                int stream_index, int64_t timestamp, int flags);
    int64_t (*read_timestamp)(struct AVFormatContext *s, int stream_index,
                        int64_t *pos, int64_t pos_limit);
    int flags;
    const char *extensions;
    int value;
    int (*read_play)(struct AVFormatContext *);
    int (*read_pause)(struct AVFormatContext *);
    const struct AVCodecTag **codec_tag;
    struct AVInputFormat *next;
}
```

A.1.6 AVOutputFormat

Declared in *avformat.h* (libavformat). Used to define an output format (encoder and multiplexer), like the *mpegts*.

```
AVOutputFormat {
    const char *name;
    const char *long_name;
    const char *mime_type;
    const char *extensions;
    int priv_data_size;
    enum CodecID audio_codec;
    enum CodecID video_codec;
    int (*write_header)(struct AVFormatContext *);
    int (*write_packet)(struct AVFormatContext *, AVPacket *pkt);
    int (*write_trailer)(struct AVFormatContext *);
    int flags;
    int (*set_parameters)(struct AVFormatContext *, AVFormatParameters *);
    int (*interleave_packet)(struct AVFormatContext *, AVPacket *out, AVPacket *in, int flush);
    struct AVOutputFormat *next;
}
```

Example:

```
AVOutputFormat mpegts_muxer = {
    "mpegts",
    "MPEG2 transport stream format",
    "video/x-mpegts",
    "ts",
    sizeof(MpegTSWrite),
    CODEC_ID_MP2,
    CODEC_ID_MPEG2VIDEO,
    mpegts_write_header,
    mpegts_write_packet,
    mpegts_write_end,
};
```

A.1.7 AVPacket

Declared in *avformat.h* (libavformat). Defines a stream packet.

```
AVPacket {
    int64_t pts;
    int64_t dts;
    uint8_t *data;
    int   size;
    int   stream_index;
    int   flags;
    int   duration;    void  (*destruct)(struct AVPacket *);
    void  *priv;
    int64_t pos;
}
```

A.1.8 AVPicture

Declared in *avcodec.h* (libavcodec). Used to store an image. Four components are given, with the last being the Alpha.

```
AVPicture {
    uint8_t *data[4];
    int linesize[4];       ///< number of bytes per line
}
```

A.1.9 AVStream

Declared in *avformat.h* (libavformat). Defines an I/O stream.

```
AVStream {
    int index;
    int id;
```

```
    AVCodecContext *codec;
    AVRational r_frame_rate;
    void *priv_data;
    int64_t codec_info_duration;
    int codec_info_nb_frames;
    AVFrac pts;
    AVRational time_base;
    int pts_wrap_bits;
    int stream_copy;
    enum AVDiscard discard;
    float quality;
    int64_t start_time;
    int64_t duration;
    char language[4];
    int need_parsing;
    struct AVCodecParserContext *parser;
    int64_t cur_dts;
    int last_IP_duration;
    int64_t last_IP_pts;
    AVIndexEntry *index_entries;
    int nb_index_entries;
    unsigned int index_entries_allocated_size;
    int64_t nb_frames;
#define MAX_REORDER_DELAY 4
    int64_t pts_buffer[MAX_REORDER_DELAY+1];
}
```

A.1.10 ByteIOContext

Declared in *avio.h* (libavformat). A buffer for I/O data.

```
ByteIOContext {
    unsigned char *buffer;
    int buffer_size;
    unsigned char *buf_ptr, *buf_end;
    void *opaque;
    int (*read_packet)(void *opaque, uint8_t *buf, int buf_size);
    int (*write_packet)(void *opaque, uint8_t *buf, int buf_size);
    offset_t (*seek)(void *opaque, offset_t offset, int whence);
    offset_t pos;
    int must_flush;
    int eof_reached;
    int write_flag;
    int is_streamed;
    int max_packet_size;
    unsigned long checksum;
    unsigned char *checksum_ptr;
    unsigned long (*update_checksum)(unsigned long checksum, const uint8_t *buf, unsigned int size);
    int error;
}
```

A.1.11 URLContext

Declared in *avio.h* (libavformat). Used to store the characteristics of a stream.

```
URLContext {
    struct URLProtocol *prot;
    int flags;
    int is_streamed;
    int max_packet_size;
    void *priv_data;
#if LIBAVFORMAT_VERSION_INT >= (52<<16)
    char *filename;
#else
    char filename[1];
#endif
}
```

A.1.12 URLProtocol

Declared in *avio.h* (libavformat). Used to define data (file or transmission) protocols, like the *rtp*.

```
URLProtocol {
    const char *name;
    int (*url_open)(URLContext *h, const char *filename, int flags);
    int (*url_read)(URLContext *h, unsigned char *buf, int size);
    int (*url_write)(URLContext *h, unsigned char *buf, int size);
    offset_t (*url_seek)(URLContext *h, offset_t pos, int whence);
    int (*url_close)(URLContext *h);
    struct URLProtocol *next;
}
```

Example:

```
URLProtocol rtp_protocol = {
    "rtp",
    rtp_open,
    rtp_read,
    rtp_write,
    NULL, /* seek */
    rtp_close,
}
```

Taksi Structures

A.2.1 CTaksiDll

Declared in *stdafx.h*. Defines the Taksi library's interface. It is inter-process shared, so it can't have a constructor or contain any data types that do.

```
CTaksiDll {
public:
        bool InitDll();
        void DestroyDll();
        void OnDetachProcess();
        HRESULT LogMessage( const TCHAR* pszPrefix );
        bool IsHookCBT() const;
        static LRESULT CALLBACK HookCBTProc(int nCode, WPARAM wParam, LPARAM
                                                        lParam);
        //bool HookCBT_Install(void);
        bool HookCBT_Uninstall(void);

        //* Start of modified or added code *
        bool HookCBT_Install( DWORD dwHookThreadId );
        bool InitMasterWnd( HWND hWnd, DWORD dwHookThreadId );
        int GetFrameSize();
        SIZE GetWindowSize();
        void SetHotKey( int eHotKey );
        bool ReadFrame( BYTE *framePtr, HANDLE processHandle );
        void WriteFrame( BYTE *framePtr );
        //* End of modified or added code *

        […]
#ifdef USE_DX
        // DX8
        UINT_PTR m_nDX8_Present;
        UINT_PTR m_nDX8_Reset;
        // DX9
        UINT_PTR m_nDX9_Present;
        UINT_PTR m_nDX9_Reset;
#endif

public:
        […]
        DWORD m_dwHotKeyMask;
        […]

        //* Start of modified or added code *
        int m_FrameSize;
        SIZE m_WndSize;
        FRAMEBUFFER m_FrameBuffer;
        //* End of modified or added code *
}
```

A.2.2 CTaksiGraphX

Defined in *graphx.h* and implemented in *graphx.cpp*. Defines a generic graphics mode base class and is inherited by the *CTaksiDX8*, *CTaksiDX9*, *CTaksiOGL* and *CTaksiGDI* structures.

```
CTaksiGraphX {
public:
        CTaksiGraphX() : m_bHookedFunctions(false), m_bGotFrameInfo(false)
        {}

        virtual const TCHAR* get_DLLName() const = 0;
        virtual HRESULT AttachGraphXMode();
        void PresentFrameBegin( bool bChange );
        void PresentFrameEnd();
        void RecordAVI_Reset();

protected:
        virtual bool HookFunctions(){
                ASSERT( IsValidDll());
                DEBUG_MSG(( "CTaksiGraphX::HookFunctions done." LOG_CR ));
                return true;
        }
        virtual void UnhookFunctions(){
                DEBUG_MSG(( "CTaksiGraphX::UnhookFunctions" LOG_CR ));
                m_bHookedFunctions = false;
        }

        virtual HWND GetFrameInfo( SIZE& rSize ) = 0;
        virtual HRESULT GetFrame( CVideoFrame& frame, bool bHalfSize ) = 0;
        virtual HRESULT DrawIndicator( TAKSI_INDICATE_TYPE eIndicate ) = 0;

private:
        HRESULT RecordAVI_Start();
        void RecordAVI_Stop();
        bool RecordAVI_Frame();
        HRESULT MakeScreenShot( bool bHalfSize );

public:
        static const DWORD sm_IndColors[TAKSI_INDICATE_QTY];
        bool m_bHookedFunctions;        bool m_bGotFrameInfo;
        //* Start of modified or added code *
        TCHAR szFileName[_MAX_PATH];
        //* End of modified or added code *
}
```

A.2.3 CTaksiDX9

Defined in *graphx.h* and implemented in *graphx_dx9.cpp*. Defines a DirectX 9 graphics mode class. It inherits the *CTaksiGraphX* class.

```
CTaksiDX9 {
public:
        CTaksiDX9();
        ~CTaksiDX9();

        virtual const TCHAR* get_DLLName() const {
                return TEXT("d3d9.dll");
        }

        virtual bool HookFunctions();
        virtual void UnhookFunctions();
        virtual void FreeDll();

        virtual HWND GetFrameInfo( SIZE& rSize );
        virtual HRESULT DrawIndicator( TAKSI_INDICATE_TYPE eIndicate );
        virtual HRESULT GetFrame( CVideoFrame& frame, bool bHalfSize );

        HRESULT RestoreDeviceObjects();
        HRESULT InvalidateDeviceObjects();

public:
        IDirect3DDevice9* m_pDevice;
        ULONG m_iRefCount;
        ULONG m_iRefCountMe;
        CHookJump m_Hook_Present;
        CHookJump m_Hook_Reset;
        UINT_PTR* m_Hook_AddRef;
        UINT_PTR* m_Hook_Release;
}
```

A.2.4 FrameBuffer

Declared in *stdafx.h*. Defines a buffer for the synchronization of the Capture process and the Encoding and Transmission process.

```
FrameBuffer{
        BYTE *frames[NUMBER_OF_FRAMES];
        bool full[NUMBER_OF_FRAMES];
        int read;
        int write;
}
```

CaptureApp Structures

A.3.1 CTaksiDll (See A.2.1.)

A.3.2 ThreadInfo

Declared in *CaptureApp.h*. Stores the data to be passed to the application's server and client threads during their creation.

```
ThreadInfo
{
        char *ipAddress;
        unsigned short serverPortNumber;
        unsigned short clientPortNumber;
        HANDLE gameEvent;
}
```

A.3.3 MyUDPContext

Declared in *MyUDP.h*. Defines the context of the MyUDP protocol. The structure is stored in a URLContext by the MyUDPOpen function and retrieved by the MyUDPWrite and MyUDPClose functions.

```
MyUDPContext
{
        SOCKET udpSocket;
        struct sockaddr_in udpDestAddr;
        int udpDestLength;
        long udpDelay;
}
```

Appendix B

In this appendix we present and provide a brief description of some important functions.

FFmpeg Functions

B.1.1 av_alloc_format_context

Declared in *utils.c* (libavformat). Allocates memory to an AVFormatContext.

B.1.2 av_init_packet

Declared in *avformat.h* (libavformat). Initializes an AVPacket.

B.1.3 av_new_stream

Declared in *utils.c* (libavformat). Adds a new stream to a media file. Can only be called in the *read_header()* function. If the flag AVFMTCTX_NOHEADER is in the format context, then new streams can be added in *read_packet()* too. Takes as parameters the media file handle and the file format dependent stream id.

B.1.4 av_register_codec_parser

Declared in *parser.c* (libavcodec). Registers an AVCodecParser.

B.1.5 av_register_input_format

Declared in *utils.c* (libavformat). Registers an AVInputFormat.

B.1.6 av_register_output_format

Declared in *utils.c* (libavformat). Registers an AVOutputFormat.

B.1.7 av_set_parameters

Declared in *utils.c* (libavformat). Sets the AVFormatParameters of an AVFormatContext.

B.1.8 av_write_frame

Declared in *utils.c* (libavformat). Writes a packet to an output media file. The packet shall contain one audio or video frame. It takes as parameters the media file handle and the packet.

B.1.9 av_write_header

Declared in *utils.c* (libavformat). Allocate the stream private data and write the stream header to an output media file. Takes the media file handle as a single parameter.

B.1.10 av_write_trailer

Declared in *utils.c* (libavformat). Write the stream trailer to an output media file and free the file private data. Takes the media file handle as a single parameter.

B.1.11 avcodec_alloc_frame

Declared in *utils.c* (libavcodec). Allocates an AVPFrame and sets it to defaults. This can be de-allocated by simply calling free().

B.1.12 avcodec_encode_video

Declared in *utils.c* (libavcodec). Encodes an AVFrame based on the given AVCodecContext.

B.1.13 avcodec_open

Declared in *utils.c* (libavcodec). Opens, initializes the AVCodecContext.

B.1.14 avpicture_fill

Declared in *imgconvert.c* (libavcodec). Allocates memory to an AVPicture based on the given format.

B.1.15 avpicture_get_size

Declared in *imgconvert.c* (libavcodec). Returns the size of an AVPicture based on the given format, width and height.

B.1.16 guess_format

Declared in *utils.c* (libavformat). Extracts the AVOutputFormat from the given format name, filename or mime type.

B.1.17 img_convert

Declared in *imgconvert.c* (libavcodec). Converts images between various pixel formats.

B.1.18 register_avcodec

Declared in *utils.c* (libavcodec). Registers a AVCodec.

B.1.19 register_protocol

Declared in *avio.c* (libavformat). Registers a URLProtocol.

B.1.20 mpegts_write_end

Declared in *mpegtsenc.c* (libavformat). Writes the trailer of the MPEG-2 Transport Stream.

B.1.21 mpegts_write_header

Declared in *mpegtsenc.c* (libavformat). Writes the header of the MPEG-2 Transport Stream.

B.1.22 mpegts_write_packet

Declared in *mpegtsenc.c* (libavformat). Inserts a packet into the MPEG-2 Transport Stream.

B.1.23 url_fopen

Declared in *aviobuf.c* (libavformat). Opens/Initializes a ByteIOContext and associates it with a protocol.

Taksi Functions

B.2.1 AttachGraphXMode

Declared in *graphx.h* and defined in *graphx.cpp*, *graphx_ogl.cpp* and *graphx_gdi.cpp*. Has the same implementation for the DirectX 8 and DirectX 9 and different for the OpenGL and the GDI. Checks if the library that supports the current graphics mode is present, without forcing it to load.

B.2.2 DestroyDll

Declared in *stdafx.h* and defined in *TaksiDll.cpp*. Unhooks the library from the currently hooked thread.

B.2.3 GetFrame

Declared in *graphx.h* and defined in *graphx_dx8.cpp*, *graphx_dx9.cpp, graphx_ogl.cpp* and *graphx_gdi.cpp*. It performs the frame capturing and is implemented differently for each of the graphics modes.

B.2.4 GetFrameSize

Declared and defined in *stdafx.h*. Returns the size of the RGB frame which is created by the capturing process.

B.2.5 GetWindowSize

Declared and defined in *stdafx.h*. Returns the resolution of the Game's display.

B.2.6 HookCBTInstall

Declared in *stdafx.h* and defined in *TaksiDll.cpp*. Hooks the library on the thread with the given thread id.

B.2.7 HookCBTProc

Declared in *stdafx.h* and defined in *TaksiDll.cpp*. The hook procedure which is called by the operating system. The procedure checks only for HCBT_SETFOCUS messages, meaning that a change on the keyboard focus occurred.

B.2.8 HookFunctions

Declared in *graphx.h* and defined in *graphx_dx8.cpp*, *graphx_dx9.cpp, graphx_ogl.cpp* and *graphx_gdi.cpp*. It performs the code modification described in §3.1.2 and is implemented differently for each of the graphics modes.

B.2.9 InitMasterWindow

Declared in *stdafx.h* and defined in *TaksiDll.cpp*. Declares the window referenced by the given handle as the main window to sent messages to and hooks the library on the desktop thread with the given thread id. Was modified to hook on a certain thread rather than any running thread.

B.2.10 PresentFrameBegin

Declared in *graphx.h* and defined in *graphx.cpp*. This is the function to which the execution jumps whenever the DirectX *Present* or *Reset* function is called, meaning that a frame is about to be drawn. Till this function is called, a hook is not truly complete.

B.2.11 PresentFrameEnd

Declared in *graphx.h* and defined in *graphx.cpp*. If needed, it unhooks the DirectX functions and stops the graphics mode.

B.2.12 ReadFrame

Declared in *stdafx.h* and defined in *TaksiDll.cpp*. Retrieves a frame pointer from the Framebuffer [A.2.2]. The frame pointer is used by the CaptureApp to read the frame data from the Game's memory space. The CaptureApp has the access rights to perform such an operation since it is the game process's parent process.

B.2.13 RecordAVI_Start

Declared in *graphx.h* and defined in *graphx.cpp*. It starts the frame capturing procedure.

B.2.14 RecordAVI_Stop

Declared in *graphx.h* and defined in *graphx.cpp*. It stops the frame capturing procedure.

B.2.15 SetHotKey

Declared and defined in *stdafx.h*. Triggers the given hotkey by properly modifying the hotkey mask.

B.2.16 UnhookFunctions

Declared in *graphx.h* and defined in *graphx_dx8.cpp*, *graphx_dx9.cpp, graphx_ogl.cpp* and *graphx_gdi.cpp*. It restores the original functions that were modified by the code modification described in §3.1.2 and is implemented differently for each of the graphics modes.

B.2.17 WriteFrame

Declared in *stdafx.h* and defined in *TaksiDll.cpp*. Inserts a frame pointer in the FrameBuffer [A.2.2].

CaptureApp Functions

B.3.1 Test_DirectX8 and Test_DirectX9

Declared in *CaptureApp.h*. They calculate the offsets of the DirectX *Present* and *Reset* functions from the start of the DX8 and DX9 dlls.

B.3.2 ReadIniFile

Declared in *CaptureApp.h*. Reads the *CaptureApp.ini* file.

B.3.3 ServerThread

Declared in *CaptureApp.h*. Impements the TCP server.

B.3.4 ClientThread

Declared in *CaptureApp.h*. Impements the Device Input Client.

B.3.5 RawInputToInput

Declared in *CaptureApp.h*. Converts the *RAWINPUT* structures into *INPUT* structures in order to be inserted into the mouse or keyboard input streams with the *SendInput* function.

B.3.6 FFmpegInit

Declared in *FFmpegFunctions.h*. Initializes all of the necessary FFmpeg structures for the data encoding and transmission.

B.3.7 FFmpegEncodeFrame

Declared in *FFmpegFunctions.h*. Performs the data encoding and transmission.

B.3.8 FFmpegClose

Declared in *FFmpegFunctions.h*. Closes/de-allocates all of the initialized FFmpeg structures.

B.3.9 Rgb24ToYuv420p

Declared in *FFmpegFunctions.h*. Converts and RGB24 frame into a YUV4:2:0. Borrowed from the FFmpeg tests named *videogen.c* and *rotozoom.c*.

B.3.10 FlipYuvData

Declared in *FFmpegFunctions.h*. Flips horizontally the YUV images generated by the Rgb24ToYuv420p function.

B.3.11 MyCreateWindow

Declared in *WindowFunctions.h*. Creates the CaptureApp's window, which is not shown on the desktop, but used to receive window messages from the Taksi library.

B.3.12 WndProc

Declared in *WindowFunctions.h*. It is the CaptureApp's window callback procedure. Receives and processes the window messages that the Taksi library posts to the CaptureApp's window.

B.3.13 MyUDPOpen

Declared in *MyUDP.h*. Opens/initializes the MyUDP protocol.

B.3.14 MyUDPWrite

Declared in *MyUDP.h*. Writes/transmits data based on the MyUDP protocol.

B.3.15 MyUDPClose

Declared in *MyUDP.h*. Closes/terminates the MyUDP protocol.

B.3.16 MyUrlSplit

Declared in *MyUDP.h*. Retrieves the hostname and the port number from a given url. Borrowed by the FFmpeg's *url_split* function.

B.3.17 MyResolveHost

Declared in *MyUDP.h*. Retrieves the host address from the given hostname. Borrowed by the FFmpeg's *resolve_host* function.

UserApp Functions

B.4.1 ReadIniFile

Declared in *UserApp.h*. Reads the *UserApp.ini* file.

B.4.2 RegisterRIDevices

Declared in *UserApp.h*. Registers the Input Devices from which the UserApp application wants to receive Raw Input events.

B.4.3 MyCreateWindow

Declared in *WindowFunctions.h*. Creates the UserApp's window, which is not shown on the desktop, but used to receive the Raw Input messages.

B.4.4 WndProc

Declared in *WindowFunctions.h*. It is the UserApp's window callback procedure. It receives and processes the Raw Input messages sent to the UserApp's window.

B.4.5 GetRawInput

Declared in *WindowFunctions.h* and used by the *WndProc*. Retrieves the Raw Input data from the received Input message and transmits them to the Game Server's Device Input Client.

B.4.6 Str_SkipSpace

Declared in *Common.h*. Skips the space characters at the beginning of a given string. Borrowed from the Taksi library (*Common.h*).

B.4.7 Str_IsSpace

Declared in *Common.h* and used by the *Str_SkipSpace*. Checks whether the given character is a space character. Borrowed from the Taksi library (*Common.h*).