3D Video Playback
A modular cross-platform GPU-based approach for flexible multi-view 3D video rendering

Håkan Andersson
Abstract

The evolution of depth-perception visualization technologies, emerging format standardization work and research within the field of multi-view 3D video and imagery addresses the need for flexible 3D video visualization. The wide variety of available 3D-display types and visualization techniques for multi-view video, as well as the high throughput requirements for high definition video, addresses the need for a real-time 3D video playback solution that takes advantage of hardware accelerated graphics, while providing a high degree of flexibility through format configuration and cross-platform interoperability. A modular component based software solution based on FFmpeg for video demultiplexing and video decoding is proposed, using OpenGL and GLUT for hardware accelerated graphics and POSIX threads for increased CPU utilization. The solution has been verified to have sufficient throughput in order to display 1080p video at the native video frame rate on the experimental system, which is considered as a standard high-end desktop PC only using commercial hardware. In order to evaluate the performance of the proposed solution a number of throughput evaluation metrics have been introduced measuring average frame rate as a function of: video bit rate, video resolution and number of views. The results obtained have indicated that the GPU constitutes the primary bottleneck in a multi-view lenticular rendering system and that multi-view rendering performance is degraded as the number of views is increased. This is a result of the current GPU square matrix texture cache architectures, resulting in texture lookup access times according to random memory access patterns when the number of views is high. The proposed solution has been identified in order to provide low CPU efficiency, i.e. low CPU hardware utilization and it is recommended to increase performance by investigating the gains of scalable multithreading techniques. It is also recommended to investigate the gains of introducing video frame buffering in video memory or to move more calculations to the CPU in order to increase GPU performance.

Keywords: 3D Video Player, Multi-view Video, Lenticular Rendering, Auto-stereoscopy, 3D Visualization, FFmpeg, GPU, OpenGL, C.
Acknowledgements

I would like to thank my supervisor Roger Olsson, Ph. D in telecommunications, Mid Sweden University, Sundsvall, Sweden for all his support throughout this work. Reviews of my report and discussions regarding technical problems and design issues that appeared during the progress of this project were very valuable to me and helped me achieve the results presented within this thesis.
Table of Contents

Abstract ............................................................................................................. ii
Acknowledgements .......................................................................................iii
Terminology ...................................................................................................vii

1 Introduction ............................................................................................1
  1.1 Background and problem motivation ..............................................1
  1.2 Overall aim ....................................................................................2
  1.3 Scope ..............................................................................................3
  1.4 Concrete and verifiable goals ..........................................................4
  1.5 Outline ...........................................................................................5
  1.6 Contributions ................................................................................6

2 Three dimensional visualization ............................................................8
  2.1 Human depth perception .................................................................8
  2.2 Stereoscopy .....................................................................................10
  2.2.1 Positive parallax .........................................................................10
  2.2.2 Negative parallax .........................................................................11
  2.2.3 Rendering stereo pairs .................................................................12
  2.3 Auto-stereoscopy .............................................................................12
  2.3.1 Barrier strip displays ..................................................................13
  2.3.2 Lenticular displays .....................................................................14

3 Three dimensional video .......................................................................16
  3.1 3D Video Formats ..........................................................................16
  3.1.1 Pre-processed raw video .............................................................16
  3.1.2 Multi-view video .........................................................................17
  3.1.3 Video-plus-depth .........................................................................18
  3.2 3D Video Players ............................................................................19
  3.2.1 Stereoscopic player ......................................................................19
  3.2.2 Visumotion 3D Movie Center ......................................................20
  3.2.3 Spatial View SVI Power Player ....................................................20
  3.3 Video application development frameworks ...................................21
  3.3.1 Apple Quicktime .........................................................................21
  3.3.2 Microsoft DirectShow ..................................................................22
  3.3.3 FFMPEG .....................................................................................23
3D Video Playback - A modular cross-platform GPU-based approach for flexible multi-view 3D video rendering

Håkan Andersson

Table of Contents

2010-11-25

3.3.4 Components ..............................................................23
3.3.5 FFDSHOW ..................................................................26
3.4 Hardware Requirements .................................................26

4 Hardware-accelerated graphics ...........................................29
4.1 Video card overview .....................................................29
4.1.1 Vertex processor ......................................................30
4.1.2 Fragment processor ..................................................31
4.2 OpenGL .........................................................................32

5 Methodology .....................................................................33
5.1 Experimental methodology to evaluate performance ........33
5.2 Evaluation of cross-platform interoperability .................36
5.3 Verifying sub-pixel spatial multiplexing .........................36
5.4 Hardware and software resources ....................................36

6 Design ............................................................................38
6.1 Alternative design solutions ...........................................38
6.2 Design considerations and system design overview ..........40
6.2.1 Design considerations for high performance ...............42
6.2.2 Design for flexibility and platform independence .......43
6.3 3D video player component design ..................................43
6.3.1 Architectural overview .............................................43
6.3.2 Demultiplexing .........................................................45
6.3.3 Video frame decoding ..............................................45
6.3.4 Color conversion .....................................................46
6.3.5 Synchronization, filtering and rendering .....................47
6.4 3D video filter pipeline ..................................................49
6.4.1 Defining input and output .........................................49
6.4.2 Filter input parameters ............................................50
6.4.3 Video filter processing pipeline ..................................51
6.4.4 Generic multi-view texture mapping coordinates .......53
6.4.5 Texture transfers .....................................................53
6.4.6 Spatial multiplexing ..................................................55
6.5 Optimization details .....................................................57

7 Result .............................................................................58
7.1 Throughput as a function of video format bitrate .........58
7.2 Throughput as a function of video resolution .................59
7.3 Throughput as a function of the number of views ..........60
7.4 Average frame rate in relation to native frame rate ........61
Table of Contents

7.5 Load balance .................................................................61
7.6 Cross-platform interoperability ........................................62
7.7 Validation of spatially multiplexed video .........................63

8 Conclusion..........................................................................64
8.1 Evaluation of system throughput ......................................64
8.2 Multi-view and GPU performance ...................................65
8.3 Evaluation of load balance and hardware utilization ...........66
8.4 Evaluation of cross-platform interoperability .....................67
8.5 Evaluation of spatial multiplexing correctness ...................68
8.6 Future work and recommendations .................................68

References..............................................................................70

Appendix A: System specification for experiments.................75
Hardware specification ..........................................................75
Specification of software used throughout this project .............75

Appendix B: Pixel buffer object (PBO) performance ...............76
## Terminology

### Mathematical notation

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>( f_{AVG} )</td>
<td>The measured average frame rate in terms of processed frames per second with video synchronization turned off.</td>
</tr>
<tr>
<td>( f_{NATIVE} )</td>
<td>The native (intended) video frame rate.</td>
</tr>
<tr>
<td>( \Delta f )</td>
<td>The difference between the measured average frame rate and native video frame rate.</td>
</tr>
<tr>
<td>( N_C )</td>
<td>Disparate view index of the view from which the RGB-sub component ( C \in {R, G, B} ) should be fetched from.</td>
</tr>
<tr>
<td>( S_N )</td>
<td>A spatially multiplexed video frame (texture) that contains pixel data from ( N ) disparate views.</td>
</tr>
<tr>
<td>( T_N )</td>
<td>The set of ( N ) texture mapping coordinate quadruples or point pairs ((u_1, v_1), (u_2, v_2)) that corresponds to the view alignment in a tiled multi-view texture with ( N ) views.</td>
</tr>
<tr>
<td>( V_n )</td>
<td>A set of multiple-views (multiple textures) that corresponds to the ( n:th ) video frame in a sequence of multi-view video frames.</td>
</tr>
</tbody>
</table>
1 Introduction

The optical principles of displaying images with inherent depth and naturally changing perspective have been known for over a century, but until recently, displays have not been available that are capable of presenting 3D images and video without requiring user-worn glasses, also known as auto-stereoscopic displays. Different compression formats are in the process of being standardized by the motion pictures expert group (MPEG) as well as ISO/IEC for stereo and multi-view based 3D video [1]. Software capable of decoding the different compression and encoding formats as well as present 3D video on different display types using standardized players is a vital part in the commercialization and evolution of 3D video. The multi-perspective nature of 3D video enforces multiple data sources. Hence there is a high demand for fast data processing and as a direct implication of that, hardware accelerated solutions are of particular interest.

1.1 Background and problem motivation

The wide variety of 3D display types and visualization techniques for multi-view video require that the pixels from each view be mapped and aligned differently depending on the video format and the specific features of the visualization device. This implies that a video format exclusively generated for a specific device type will not be displayed correctly on other kinds of visualization devices. This is obviously a problem as the same video content has to be replicated in several different versions in order to be presented correctly on different visualization devices and screen resolutions.

Several different video formats such as for example multi-view video and video-plus-depth formats have been proposed, representing generic 3D video formats to address this problem as well as video compression issues. Using any of these generic formats, video can be interpreted and processed in real-time to generate the correct pixel mappings required in order to display the 3D video content correctly. A generic format for representing 3D video thus eliminates the need to generate several different pre-processed versions of the same video content [2].
The number of publicly available software video players capable of decoding and displaying 3D video is very limited and the few players available involve licensing fees, such as 3D Movie Center by Visumotion [3] and Stereoscopic player by Peter Wimmer [4]. In addition to the small range of available 3D video players, only a handful of pre-defined video formats are usually supported, most commonly conventional stereo. Another disadvantage of these players is platform dependency.

The lack of flexibility and limited functionality in current 3D video players means that there is a need for a flexible cross-platform playback solution that can easily be configured and extended to support a wide range of both current and emerging displays and 3D video format standards. It is also desirable to investigate any associated hardware bottlenecks.

1.2 Overall aim

The overall aim of this project is to design and implement a video playback solution, capable of displaying the basic 3D video formats. The possibilities of creating a playback solution built on top of a cross-platform, open source libraries for video decoding and hardware accelerated graphics will also be investigated in this work.

Interpreting video data and converting it to a format that is compliant with the display in real-time places high demands on system hardware as well as algorithm efficiency and therefore it is also of great interest to identify bottlenecks in the processing of 3D video content. By measuring hardware utilization and video throughput in terms of frame rate, this thesis aims at emphasising throughput related hardware problems associated with real-time 3D video processing. This, in turn, might be valuable for future research, especially in fields such as 3D video compression and video encoding as well as playback system design.

Moreover, the project aims to identify and propose a software architecture sufficiently efficient to process and present high definition 3D video in real-time, yet be sufficiently flexible to support both current 3D video formats and emerging standards. It is also highly desirable to exploit the possibilities of implementing 3D video support in currently available video player software by means of extending the functionality
of the existing software. This would eliminate the need for implementing synchronization, audio decoding, audio playback etc. which would be required if a video player was to be designed and implemented from scratch.

1.3 Scope

This study is primarily focused on designing and implementing a 3D video playback solution for multi-view auto-stereoscopic content, primarily for lenticular displays. Hence, software architectural design and generalization of 3D video formats and display technologies are of greater interest than the implementation of extensive support for specific formats, display types or display brands.

The comparison of suitable frameworks and libraries to use within this project is restricted to only giving consideration to cross-platform and open-source solutions. In addition, only frameworks that are non-commercial and free of charge are of interest. The choice of frameworks, libraries and platforms used throughout this project will be based purely on the results from the theoretical studies of related work and existing technologies publicly available. No experiments or benchmarking regarding this matter will be performed within this study.

The theoretical part of this work is moreover limited to only offering the reader a brief introduction to the research field of auto-stereoscopy and 3D visualization which is required in order to understand this work. Frameworks and libraries considered for this project will only be described briefly except for key parts and technologies of particular interest for this work.

The practical part of this work aims at implementing a video playback solution as a simple prototype for research purposes according to the technical requirements of this thesis. No extensive testing of this prototype other than simple developer tests during the implementation phase will be conducted within the scope of this project. Performance measurements and the results obtained will be restricted to be only performed on one system, OS and hardware configuration.
1.4 Concrete and verifiable goals

A survey covering the possibilities and limitations of creating a software 3D video playback solution on top of an existing video decoder and hardware accelerated graphic frameworks must be produced. However, such an analysis would involve an endless list of candidates and combinations. Therefore only two popular APIs for hardware accelerated graphics and three APIs or video decoding are to be considered.

The implemented solution should aim at being platform independent. Hence it must be able of compiling and running on several completely different platforms including at least Microsoft Windows, Mac OS and Linux distributions. This will require that the source code of the implementation contains several different code segments where access to operating system APIs is required. It would not be feasible to compile and test the implemented prototype on all platforms within the scope of this project, but there should be no calls to system dependent functions within the source code without compile-time pre-processor conditional branching. Cross-platform interoperability would also be considered as being fulfilled if the libraries and APIs used in the software claims to be portable or implements a standardized interface.

The implemented rendering framework for 3D video must be a real-time system which is sufficiently fast to display 3D video content at the frame rate it was intended for. This requirement is necessary to guarantee flawless video playback and a high degree of usability. A higher frame rate than that intended would indicate that there is headroom for additional or more advanced processing. A lower frame rate would be considered as unacceptable for flawless video playback.

The 3D video output can be verified subjectively by observing a video or generated still image using a supported display. However the results can be verified in an objective manner by calculating the expected output and verifying the output using pixel by pixel comparison.

The minimum requirements for the theoretical part of this work are:

- Identify at least two possible candidate APIs which support cross-platform hardware accelerated graphics.
• Identify at least three widely used video decoder frameworks potential for being modified and integrated within an existing standard video player and/or serve as an underlying decoding library.

The minimum requirements for the practical part of this work are:

• Design a modular and flexible 3D video playback solution that can be used to spatially multiplex 3D video from display and video format parameters. The solution should be built around the two most promising frameworks for video decoding and hardware accelerated graphics revealed in the theoretical part of this thesis.

• Measure throughput of the designed solution in order to identify potential bottlenecks caused by bandwidth limitations in hardware or anomalies between different 3D video encoding and compression formats.

• Measure hardware utilization in order to determine if the proposed software solution takes full advantage of available hardware resources.

• Verify that pixels are mapped (spatially multiplexed) correctly for an auto-stereoscopic lenticular 25-view display (LG “42, 25-view lenticular display) using the proposed model.

• Verify that conventionally compressed multi-view full-HD 1,080p (1,920x1,080) 3D-video can be displayed at the frame rate it was intended for (typically 25-30 frames per second).

• Verify that the playback solution potentially can be compiled and run on at least Microsoft Windows, Mac OS and Linux distributions.

1.5 Outline

The report is designed to present its content in such a way that it can be translated directly from the research conducted throughout this project.
• Chapter 2 explains the theory of depth-perception, stereoscopy and visualization techniques for 3D video and imagery.

• Chapter 3 describes the concepts of 3D video formats, video demultiplexing, video decoding and hardware requirements for 3D video.

• Chapter 4 consists of a brief overview of video card hardware and industry standard libraries for hardware accelerated graphics.

• Chapter 5 describes the methodology used throughout this project, covering experimental methods, evaluation metrics and methods as well as available resources.

• Chapter 6 describes the design considerations made when attempting to identify suitable software architecture for 3D video processing and well as important implementation details vital in order to understand the problems of real-time 3D video processing.

• Chapter 7 presents the results obtained from the experiments conducted within this work, covering throughput in terms of measured average frame rates for a number of video files used for experimental evaluation.

• Chapter 8 concludes this report and contains an evaluation of this work and directions for further improvements to the proposed software model as well as recommendations for 3D video processing in general.

1.6 Contributions
This work has contributed to the research and development of 3D visualization by presenting a prototype framework for real-time 3D video playback and processing, capable of displaying many of the common 3D video formats. Several important observations and results have been obtained throughout this work, especially considering performance bottlenecks when rendering to lenticular displays. The prototype 3D video player software solution is to this date the only
available cross-platform software supporting auto-stereoscopic content. The prototype video player is also the only known 3D video player which is sufficiently generic to display multi-view video with an arbitrary number of views. Hopefully this work will be further improved to provide a more sophisticated solution and contribute to 3D video research as a convenient way to practically test and evaluate new algorithms and hypotheses.

Roger Olsson, at Mid Sweden University has contributed to this work by supplying demultiplexing and multiplexing pixel mapping routines for the 25-view tiled LG 42” lenticular display. This code was implemented in MATLAB. The code was then modified and ported to GLSL in order to adapt to the technologies used within the implemented 3D video filtering framework.

The internal structure and workflow of the implemented video player is heavily influenced by the tutorial on video playback using FFmpeg [5], written by and published by Stephen Dranger [6] who in turn based his work on the tutorial written by Martin Böhme [7]. However the workflow has been greatly modified in this work to adapt to multi-stream video demultiplexing and decoding as well as extended parallelism in the video processing pipeline. The underlying graphics framework has also been interchanged from SDL to native OpenGL.
2 Three dimensional visualization

3D video and 3D TV has gained much attention during the last couple of years and has been the subject of extensive research at both universities and in industry. Auto-stereoscopic displays are now available as commercial products and this enables 3D visualization without wearing specialized glasses. The technology behind the displays is built to stimulate human depth perception to a greater extent than traditional visualization devices.

2.1 Human depth perception

Traditional displays such as computer screens, TVs etc. visualize images and video by displaying a fine grained grid of pixels. This creates a flat two dimensional image that can only illustrate depth in a limited number of ways. However, the human visual system uses several cognitive cues to percieve depth from two-dimensional images as pointed out by M. Siegel et al. [8]. These depth perception cues include:

- **Interposition and partial occlusion** – If an object is blocking a part of another object it is closer to the observer than the partially covered object.

- **Shadows and lightning** – Gives information on the three dimensional form of objects as well as on the position of a source of light. Brighter objects of two otherwise identical objects are perceived as being closer to the observer.

- **Relative size** – Objects of the same size but in varying distances cast different retinal image sizes. The size-distance relation gives a cue about the distance of objects of known absolute or relative size.

- **Perspective** – As a consequence of the size-distance relation, physically parallel structures seem to converge in infinite distance.
Aerial perspective – Atmospheric attenuation and scattering by dust make distant objects appear blurred and less sharp than objects closer to the observer. In other words: object detail decreases with increasing distance.

Familiarity – It is easier to perceive depth in pictures with familiar objects than in pictures with unfamiliar or abstract scenery.

However, there exist other depth perception cues that are not possible to visualize on traditional displays. These involve:

- Stereopsis or binocular disparity – The human eyes are separated horizontally by the interocular distance (distance between the eyes). Binocular disparity addresses the difference in the images projected onto the back of the eye and then onto the visual cortex. Hence, depth perception is stimulated by binocular perspective parallax between left and right eye views and motion parallax even if unrecognizable objects are visualized. [8]

- Accommodation – The muscle tension needed to change the focal length of the eye lens in order to focus at a particular depth is sent to the visual cortex where it is used to interpret depth [9].

- Convergence – This is the muscle tension required to rotate each eye so that it is facing the focal point [9].

Binocular disparity is considered the most dominant depth perception cue for the majority of people [8]. This implies that in order to create a stereo image pair to visualize depth, one needs to create two images, one for each eye. It is vital that the images are created in such a way that when independently viewed they will present an acceptable image to the visual cortex. When viewed as a stereo-pair, the human visual system will fuse the images and extract the depth information as it does in normal viewing. Conflicts in any of the depth cues may result in one cue being dominant, depth perception may be reduced or the image may be uncomfortable to watch. In the worst case the stereo pairs may not fuse at all and will be viewed as two separate images. [9]
2.2 Stereoscopy
Several different display technologies exist for viewing stereoscopic content. Commonly, stereoscopic image pairs are presented, that create a virtual 3D image with correct binocular disparity and convergence cues. However, accommodation cues are inconsistent as both eyes are looking at flat images. The human visual system will tolerate this accommodation to a certain level. A maximum separation on the display of 1/30 of the distance of the viewer to the display is seen as a good reference value for the maximum separation. [10]

2.2.1 Positive parallax
When viewing stereo image pairs on a computer display, the display surface is used as the projection plane for the three dimensional scenery. If an object is placed behind the projection plane as illustrated in Figure 1, the projection point for the left eye will be placed on the left side in the projection plane and the projection point for the right eye will be placed to the right in the projection plane. The distance between the left and right eye projections is called the horizontal parallax. If an object lies at the projection plane then its projection onto the focal plane is coincident for the left and right eye, hence there is zero parallax.

![Figure 1: Positive parallax](image)

When the projections are on the same side as their respective eyes, as illustrated in Figure 1, this is called positive horizontal parallax. The maximum positive parallax occurs when the object to be projected is
placed infinitely far away. At this point the horizontal parallax is equal to the distance between the left and right eye, also referred to as the *interocular distance*. [10]

### 2.2.2 Negative parallax

The opposite of positive parallax is *negative parallax*, which arises when an object is placed in front of the projection plane. Hence, the left eye projection will be placed on the right side of the projection plane and the right eye projection will be placed on the left side of the projection plane as illustrated in Figure 2.

![Negative parallax diagram](image)

*Figure 2: Negative parallax – Projection for the left eye is on the right side and projection for the right eye is on the left side [10].*

The negative horizontal parallax equal to the interocular distance occurs when the object is half way between the projection plane and the center of the eyes. When the object moves closer to the viewer, the negative horizontal parallax converges to infinity. [9]

The degree to which an observer’s visual system will fuse large negative parallax depends on the quality of the projection system (degree of ghosting). High values of negative parallax are a key contributor to eyestrain. Hence, limiting the negative parallax distance plays a key role in the design of stereoscopic content. [10]
2.2.3 Rendering stereo pairs

The correct way to render stereo pairs requires a non-symmetric camera frustum, which is offered by some software rendering packages, for example OpenGL [11]. Figure 3 illustrates how to correctly set up two non-symmetric camera frustums for stereo pair generation [10].

![Non-symmetric camera frustums for left and right eye projection when rendering stereo image pairs](image)

2.3 Auto-stereoscopy

Auto-stereoscopic displays provide a spatial image without requiring the user to use any special device and are often based on the idea of a barrier strip or lenticular lens displays. The different techniques are similar, but have different properties when it comes to viewing angles and resolution.

Common to both techniques is the fact that the infinite number of views an observer can see in the real world is partitioned into a finite number of available viewing zones as illustrated in Figure 4 [12]. Barrier displays prevent the observer from seeing more than one stereo pair by blocking other views [13], while lenticular displays use optical lenses to direct light in different angles to form viewing zones for disparate views [12].

Each view is dominant for a given zone of the viewing space and the observer perceives a spatial image as long as both eyes are in the viewing space and observes the image from different view zones.
respectively. Changes in the observer’s position results in different spatial depth perceptions which means that multiple observers can be accommodated simultaneously where each observer has a different spatial perception according to his/her point of view in the viewing space [12].

![Image](image.png)

**Figure 4:** Auto-stereoscopic displays divides view space into a finite, discrete number of view zones or viewing cones. [14]

### 2.3.1 Barrier strip displays

Barrier strip displays have a limited viewing angle which depends on how many images are used. The resolution of barrier strip displays is also limited and is inversely proportional to a function of the number of images used. Using barrier strip displays it is possible to use a camera head tracking system to align the images correctly depending on the observer’s head. Left and right eye stereo image pairs are used to produce a parallax stereogram image as illustrated in Figure 5. [13]

The parallax stereogram image consists of stripes from both the left eye image and the right eye image with a vertical pixel height that corresponds to the vertical screen resolution and a pixel width that corresponds to the size of the barrier of the barrier strip display. A barrier strip display makes it possible for the image strips in the parallax stereogram image to be exclusively divided in such a way that the left and right images are separated. This is achieved by allowing a barrier to prevent the right eye from looking at the image strips intended for the
left eye and to prevent the left eye from looking at the image strips intended for the right eye. Hence there is no need for additional glasses or other helper devices to create a perception of depth. The idea behind the interaction of the barrier strip display and the parallax stereogram image is illustrated in Figure 6. [13]

Figure 5: Assembly of a parallax stereogram image from left and right eye images to a parallax stereogram image [13].

Figure 6: Top view illustration of a barrier strip display with two views [13].

### 2.3.2 Lenticular displays

Lenticular displays are coated by optical lenses in order to make the underlying RGB-pixels be emitted into different zones in the viewing space. Two techniques are typically used: *lenticular sheets* and *wavelength-selective filter arrays*. Lenticular sheets consist of long
cylindrical lenses which focus on the underlying image plane and are aligned in such a way that each viewing zone sees different sets of pixels from the underlying image plane. This enables multiple views to be spatially multiplexed as different pixel columns in the underlying image.

Wavelength-selective filter arrays are based on the same principle, except that the lenses are diagonally slanted. By using diagonally oriented lenses, it is possible to provide sub-pixel resolution of view zones. This is achieved by allowing the three colour channels of RGB-pixels in the underlying display (usually LCD) correspond to a different view zones. [12] An example of how RGB components can be aligned for a nine-view display is illustrated in Figure 7. Notice how the highlighted patch of RGB-component mappings is repeated diagonally and this is dependent on the angle $\alpha$.

Figure 7: Sub-pixel alignment on a nine-view lenticular wave-length selective filter display [15].
3 Three dimensional video

As described in chapter 2.3, 3D displays require pixel or sub-pixel multiplexed spatial images in order to enable the observer to perceive depth. Since the multiplexing of view pixels or sub-pixels is dependent on the display resolution and visualization technique, this places restrictions on, and affects the representation of 3D content in several ways. A wide variety of displays and video formats are available, but no uniform method to playback 3D video regardless of the display and video setup. This makes it important to analyze different 3D video formats, current solutions for 3D video playback and video decoding as well as the hardware and processing requirements for 3D video playback.

3.1 3D Video Formats

When discussing the concept of 3D video it is important to distinguish between the 3D video format and the 3D video encoding format. The 3D video format discussed in this thesis defines how each view or set of views is aligned within a frame as well as what type of information is represented. 3D video encoding format on the other hand is reserved to define how the frames are encoded and compressed. Still it is vital to understand that the way data is represented may also affect the encoding scheme and vice versa. There are many different representations of 3D video formats available today and some of the basic, well renowned formats are outlined below.

3.1.1 Pre-processed raw video

3D video playback is highly display dependent as different algorithms have to be used to spatially multiplex pixels from disparate multi-view video into an interlaced video frame depending on the properties of the 3D display. The process to generate an interlaced (spatially multiplexed) video frame representing depth through several merged disparate views is a process that can be performed beforehand, where each frame in a video file is a pre-processed frame aimed towards a particular display type and resolution. However, even though this technique is straight-
forward and eliminates the need for additional post-production processing in the video playback system, this video format is very limited as it can only target a particular display type and screen resolution (assuming that the mapping function is unknown). An example of a 25-view pre-processed spatial image is illustrated in Figure 8.

![Figure 8: Pre-processed spatial image containing multiplexed pixel data from 25 disparate views.](image)

3.1.2 Multi-view video

Multi-view video content may be created from an arbitrary number of views, but at least two disparate views are required to represent depth. The most straightforward way of representing two or multiple views is to align them side-by-side within a single video frame or in multiple video streams in the same media container file. For this family of formats, each view represents the same scene at a given time, but with different perspectives. The disparate views can then be multiplexed into a spatial image using a pixel mapping algorithm that takes the properties of the particular 3D-visualization system into account. Hence this format is more flexible as several display types can be supported by the same video file since the disparate views may be scaled and processed before they are processed by the multiplexer. If a multi-view
video with $N$ disparate views is to be multiplexed and displayed on a multi-view display with $M$ disparate views, this is relatively simple (not considering possible aliasing due to down-sampling) if $N \geq M$ by leaving some views unused. However, if $N \leq M$, $M-N$ additional views must be generated which would involve complex synthesizing and interpolation algorithms or the full view dynamics of the display will not be used. This problem does not exist for formats such as video-plus-depth (see chapter 3.1.3).

It should be noted that so as to multiplex multiple views into a spatial image, properties of the display such as 3D-visualization technique, resolution and in case of lenticular displays, lens configuration must be known. If all these parameters are known it is also possible to apply the inverse and demultiplex a spatial image into disparate views. An example of a tiled multi-view texture corresponding to the texture in Figure 8 is illustrated in Figure 9.

![Image](image_url)

**Figure 9**: Horizontal and vertical tile representation of multi-view video consisting of 25 disparate views.

### 3.1.3 Video-plus-depth

Another approach to represent 3D video data is to separate texture from geometrical data. This can be achieved by estimating the depth of the image from several disparate image pairs or measuring the distance
from the camera to the objects represented within the image when the video is captured. The distance can then be represented as a depth map, which is typically a greyscale image that represents the Z-value (depth value) of each pixel. Black pixels represent the maximum distance to the object while white pixels represent the minimum distance to the object.

By separating texture and geometry, views can be approximated from the texture and depth map and an arbitrary number of disparate views be synthesized. This avoids aliasing problems that appear when the number of views in the multi-view video and the number of views in the 3D display do not match, which is the case for tiled multi-view formats. An example of a texture with corresponding depth map is illustrated in Figure 10. [16]

![Figure 10: Texture is separated from geometric depth data using a grayscale depth map that represents depth by pixel luminance.][16]

## 3.2 3D Video Players

A number of commercial 3D video players are available that supports 3D-video playback for several video formats and encoding formats. However, these players have more or less limited functionality in a numbers of different ways as described below.

### 3.2.1 Stereoscopic player

The Stereoscopic player created by Peter Wimmer [4], is a specialized video player for stereoscopic video and DVD playback. It is based on Microsoft DirectShow and hence supports numerous file formats including: AVI, ASF, WMV, MPEG etc., but can only run on Microsoft
Windows operating systems. Stereoscopic player requires the user to specify the video format in order to inform the program with redargs to how the image pairs are aligned within a tiled texture and which type of stereoscopic display or gear is used: anaglyph glasses or interlaced display with polarized glasses. However, some formats support automatic encoding configurations through recognition and, by connecting to an online server, a configuration file will be automatically downloaded. The video graphics are hardware accelerated using the Microsoft DirectX framework. This player only supports stereo content to be displayed and does not support multi-stream video. [4]

3.2.2 Visumotion 3D Movie Center

The Visumotion 3D Movie Center [3] is a multi-format video player that comes in three different versions with a common video player function but different additional features. The video player supports 3D video formats including: MPEG-4 multi-stream format (mp4-files), Philips 2D+Z tile formats (s3d-files), MPEG2/MPEG4-EightTile formats, MPEG2/MPEG4-NineTile etc. The player uses Microsoft DirectX as the underlying rendering library and hence only runs on Microsoft Windows OS.

The video player requires the Visumotion 3D Movie Center Display Configurator driver to be installed, which allows the user to set up which type of 3D display is connected. The display driver supports both stereoscopic displays as well as auto-stereoscopic multi-view displays. Display modes for full-screen viewing and fixed position, windowed mode are supported. [3]

3.2.3 Spatial View SVI Power Player

SVI Power Player [17] from Spatial View is a multi-format video player that supports playback of side-by-side and 3x3 tiled formats on several stereoscopic and auto-stereoscopic displays. The player is also capable of displaying 3D-models in VRML/3DS/OBJ format on 3D-displays using real-time rendering techniques. The player supports all video codecs supported by Microsoft DirectShow, for example DivX and WMF. [17]
3.3 Video application development frameworks

Several frameworks and software libraries exist which support media container demultiplexing and video decoding of multiple video and audio encoding formats. The frameworks mentioned here are some of the most widely used software video processing tools and may be considered as industry standard application programming interfaces (APIs). Apple Quicktime API [18], Microsoft DirectShow [19] and FFmpeg [5] frameworks are briefly described in this section and an analysis of FFDShow, which is a DirectPlay plug-in filter based on FFmpeg concludes this section.

3.3.1 Apple Quicktime

Applications built-on the Apple QuickTime [18] framework for multimedia decoding and display runs on Mac OS and Windows as well as for some handheld devices. QuickTime is developed and distributed by Apple and hence is integrated within Mac OS. For Windows it can be downloaded as a stand-alone package or integrated in the iTunes application bundle. The QuickTime framework is a set of tools and plug-in components which support multiple multimedia container formats. The QuickTime API consists of several different parts. One essential part is the Movie Toolbox which is used to initialize QuickTime, open, play, edit and save movies as well as to manipulate time-based media. The Image Compressions Manager is a device- and driver-independent compression and decompression utility for image data. The Sequence Grabber is a framework to support recording of real-time sources such as video and audio inputs and the streaming API supports transmission of real-time streams using standard protocols such as real-time transport protocol (RTP) real time streaming protocol (RTSP).

QuickTime is built around a component based architecture where different media handlers are responsible for handling different media formats. New media types can be added by creating a new media handler which can be integrated in QuickTime through a plug-in architecture. There are also components which support the control of playback, data access, image compression, image decompression, image filtering and audio playback. An overview of the QuickTime API including tool sets and components is depicted in Figure 11. [18]
3D Video Playback - A modular cross-platform GPU-based approach for flexible multi-view 3D video rendering
Håkan Andersson

Figure 11: The tool sets and components that constitutes the Apple QuickTime API [18].

The output of a QuickTime application is typically sound or video to a visible controller, but output to a hardware interface such as FireWire is also supported. The actual output is handled by lower level technologies including: DirectX, OpenGL, Core Image, Core Audio or the Sound Manager. The actual technology to be used is selected dependent on the system or platform that the application runs on. It is possible to process the QuickTime output by creating an implementation that processes for example the individual video frames. QuickTime 7 and later versions support the creation of visual contexts to a specific output format such as OpenGL textures. By doing this, the visual context must also be manually rendered to screen. [18]

QuickTime supports some cross-platform interoperability [18], but since the API is not provided as open-source, QuickTime does not fulfil the initial requirements stated for this thesis (see chapter 1.4 for details).

3.3.2 Microsoft DirectShow
The Microsoft DirectShow API [19] is part of the Windows software development kit (Windows SDK) and is a multi-streaming architecture for the Microsoft Windows platform. The API provides functionality and mechanisms for video and audio media playback and capture.
DirectShow supports a wide range of popular encoding formats including: Advanced Systems Format (ASF), Motion Picture Experts Group (MPEG), Audio-Video Interleaved (AVI), MPEG Audio Layer-3 (MP3), and WAV sound files. DirectShow supports automatic detection of hardware accelerated video cards and audio cards which are used whenever possible. DirectShow is based on the component object model (COM) and is designed for C++ even though extensions are available for other programming languages. DirectShow supports the creation of new customized DirectShow components and supports new formats and custom effects to be added. [19]

DirectShow is flexible through a relaxed filter plug-in architecture which supports the addition of new codes, but neither supports cross-platform interoperability nor is open-source. Hence, DirectShow is not a qualified candidate for this project according to the requirements stated in chapter 1.4.

3.3.3 FFMPEG

FFmpeg [5] is a complete cross-platform solution for digital video and audio recording, conversion and streaming. FFmpeg is free and is licensed under the LGPL (Less General Public License) or GPL (General Public License) depending on the choice of configuration options and FFmpeg users must adhere to the terms of the licence. FFmpeg is open-source and is used by several open source video player projects such as MPlayer [20] and VLC Media Player [21]. Hence, FFmpeg is a good candidate for the design of a cross-platform 3D video playback solution and is qualified as a video framework according to the requirements stated in chapter 1.4.

3.3.4 Components

FFmpeg is a software suite that is composed of several different open source sub-components. Some of these components have been created explicitly for the FFmpeg project, while some are also used by other projects. Some essential parts of FFmpeg are [5]:

- `ffserver` - which is a hyper text transfer protocol (HTTP) and real time streaming protocol (RTSP) multimedia streaming server for live broadcast.
3D Video Playback - A modular cross-platform GPU-based approach for flexible multi-view 3D video rendering

Håkan Andersson

### Three dimensional video

- **ffplay** - which is a simple video player based on the *simple direct media layer* (SDL) and the FFmpeg libraries.
- **libavcodec** - which is a LGPL licensed library of codec's for decoding and encoding digital audio and video data.
- **libavformat** - is a library containing multiplexers and demultiplexers for the different multimedia container formats.
- **libavutil** - is a helper library containing functions to simplify FFmpeg development. The library contains pseudo random number generators, data structures and mathematical functions for common codec operations like transforms.
- **libavdevice** - is a library containing I/O devices for grabbing from and rendering to many common multimedia I/O software networks.
- **libswscale** - is a library containing highly optimized image scaling functions and colour space/pixel format conversion operations.

### Multiplexing and demultiplexing

One of the core components of FFmpeg is **libavformat** which is a collection of multiplexers and demultiplexers for different multimedia container formats [5]. Container formats are used to specify the way data is stored rather than how it is encoded and through multimedia container formats both multiple video, audio and additional data such as subtitles may be stored along with synchronization data in the same file or data stream. Some examples of supported multimedia container formats are: MP4 (Standard audio and video container for the MPEG-4 multimedia portfolio), MOV (Standard QuickTime container from Apple) [22] and AVI which is the standard Microsoft Windows container [23].

The codec library **libavcodec** and the multiplexer/demultiplexer suite **libavformat** may be considered as being the essential core parts of FFmpeg. When a multimedia format is identified by FFmpeg for playback (decoding) it is looked up in a static compile-time generated register within the **libavformat** component as illustrated in Figure 12.
3D Video Playback - A modular cross-platform GPU-based approach for flexible multi-view 3D video rendering
Håkan Andersson

Figure 12: Schematic view of media container demultiplexing and decoding of N streams using libavformat and libavcodec of FFmpeg [5].

The register contains a list of all the supported demultiplexers within the current build and returns an appropriate demultiplexer function if the format is a known format. The demultiplexer function can be used to read the different data streams within the multimedia container: video, audio etc. to extract encoded data packets for each stream.

Information regarding the encoding format of each stream can be retrieved from the media container and appropriate decoder functions can be retrieved from a codec look-up in the decoder registry of libavcodec. The decoder retrieved from the libavcodec component that matches the encoding format of a demultiplexed data packet stream can then be used to decode the packets in order to retrieve uncompressed data [24]. Figure 12 illustrates the behaviour described above schematically for a multimedia container with N streams, where $S_N$ corresponds the $n$:th stream.

25
Video decoders supplied by libavcodec represent uncompressed video frames as pixel arrays with additional color format information, synchronization information etc. Pixel data is represented in the color-space defined by the encoding scheme and hence color conversion may be required to display video decoded with libavcodec on a RGB-based display. The libswscale library of FFmpeg supplies optimized color-space conversion routines between the most common formats, such as for example YCbCr (YUV) and RGB. In addition to color transformation, libswscale also contains image scaling functions to rescale video frames for different resolutions to those of the native video or image resolution.

3.3.5 FFDSHOW

FFDSHOW Tryouts [25] is an open-source project based on the DirectShow decoding filter [19] for decompressing DivX, Xvid, H. 264, FLV1, WMV, MPEG-1, MPEG-2 and MPEG-4 movies. FFDSHOW Tryouts uses the libavcodec library for decoding, which is part of the FFmpeg [5] project for video decompression. For post processing, FFDSHOW Tryouts borrows code from MPlayer [20] to enhance the visual quality of low bit rate movies. FFDSHOW Tryouts is based on the original DirectShow filter from XviD [25].

FFDSHOW Tryouts does not come with any particular player. Instead FFDSHOW Tryouts can automatically be used as a filter plug-in by DirectShow compatible software [26]. FFDSHOW Tryouts continues to support more encoding formats as FFmpeg developers add more encoding formats to libavcodec. FFDSHOW Tryouts combines DirectShow and FFmpeg to support a wide range of encoding formats and to support FFmpeg decoding within DirectShow based players like Microsoft Media Player [25]. However, even though it is integrated with FFmpeg which supports cross-platform video decoding, FFDSHOW does not support cross-platform interoperability as DirectShow is only supported by Windows OS. Hence, FFDSHOW Tryouts does not qualify as a video framework to use for cross-platform 3D video playback.

3.4 Hardware Requirements

The 1,080p video standard (also known as “Full HD”) is one of the HDTV family video modes which addresses the fact that each frame
3D Video Playback - A modular cross-platform GPU-based approach for flexible multi-view 3D video rendering

Håkan Andersson

Three dimensional video

consists of 1,080 horizontal scan lines. The 1,080-formats usually imply a resolution width of 1,920 pixels, which creates a resolution of 1,920x1,080, giving 2,073,600 pixels in total [27]. Considering uncompressed throughput for the playback of standard 1,080p 24-bit color video content, the system throughput must be at least 155.52 MB/s for an update frequency of 25 frames/s. Considering the different raw 3D-video formats (see chapter 3.1) there will be very high data bandwidth requirements for the client system if the resolutions should adapt to the 1,080p standard as illustrated in Table 1. This highlights the need for efficient compression algorithms to reduce the requirements on the hard drive or network access data rates. Even though elaborating on video compression formats is not within the scope of this report, the system must satisfy the minimum theoretical throughput requirements in order to perform 3D video playback.

It is most convenient to express the minimum 3D video throughput as the minimum data rate expressed as mega bytes per second (MB/s) which can be derived from the total video resolution (number of pixels), color depth and number of frames per second (FPS) according to:

$$Minimum\ \text{throughput} = \frac{\text{Width} \times \text{Height} \times \text{Bit color depth} \times \text{FPS}}{8 \times 10^6} \ \text{MB/s}$$

(3.1)

The minimum system throughput for the 3D video formats described in chapter 3.1 can be derived from equation (3.1) and this is described in Table 1.

Table 1: Throughput expressed in mega-pixels (MPixels) and mega-bytes per second (MB/s) for uncompressed progressive high definition (HD 1080p) 3D video.

<table>
<thead>
<tr>
<th>Video format</th>
<th>Width</th>
<th>Height</th>
<th>Color depth (bits)</th>
<th>FPS</th>
<th>Throughput (MB/s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Raw</td>
<td>1920</td>
<td>1080</td>
<td>24</td>
<td>25</td>
<td>155.52</td>
</tr>
<tr>
<td>Tiled stereo</td>
<td>3840</td>
<td>1080</td>
<td>24</td>
<td>25</td>
<td>311.04</td>
</tr>
<tr>
<td>Tiled</td>
<td>1920</td>
<td>1080</td>
<td>24</td>
<td>25</td>
<td>155.52</td>
</tr>
<tr>
<td>2D+Z</td>
<td>1920</td>
<td>1080</td>
<td>32</td>
<td>25</td>
<td>207.36</td>
</tr>
<tr>
<td>Multistream</td>
<td>1920</td>
<td>1080</td>
<td>24</td>
<td>25</td>
<td>155.52</td>
</tr>
</tbody>
</table>

For raw pre-processed 3D video the data rate will be 155.52 MB/s, which is equal to the data rate of standard 1,080p video. Stereo content with
two disparate views may share either vertical or horizontal resolution and hence result in a data rate of 155.52 MB/s, or if no degradation of resolution is desirable result in a data rate of 311.04 MB/s. Video-plus-depth (2D+Z) formats requires a fourth colour component apart from RGB to represent depth and hence will require a data rate of 207.36 MB/s. Tiled and multi-stream formats will require a data rate of 155.52 MB/s if multiple views share either horizontal and/or vertical resolution. Since full-HD tiled stereo and over-sampled multi-view formats may in reality be of any resolution 2D+Z can be seen as the minimum requirement as it contains full 1,080p resolution texture and per-pixel depth information. Hence, 207.36 MB/s can be viewed as the minimum throughput requirement for a 3D video playback system.

The system throughput of a PC desktop computer used for 3D video playback is limited by either of the following: system bus speed, hard drive data transfer rate, memory access time, peripheral interface bandwidth, CPU speed and GPU speed. Modern systems have very high bandwidths considering the system bus, PCI Express interface and volatile memory access. Hence, hard drive data transfer rate or CPU/GPU processing speed is likely to be the bandwidth bottleneck in a 3D video playback system.

*Hard drive data transfer rate* in this particular case targets the time it takes to transfer data from the media container file stored in persistent storage (typically hard drive). The total data transfer time is a function of both internal speed (physical properties, buffers and mechanics of the drive) and external speed (I/O interface) and can be divided into *disk to buffer data rate* and *buffer to memory data rate*. [28] The disk to buffer data rate is usually slower than the buffer to memory data rate since the mechanics involved in the hard drive is slower than transferring data from the buffer memory to the system memory.

If a high media compression level is used when encoding 3D video, file size may be reduced dramatically in order to eliminate the hard drive access time from being a potential bottleneck. Modern interfaces such as SATA are capable of speeds within the range 150-300 MB/s and this is likely to be sufficient for highly compressed video even if the data is highly fragmented. The hard drive used for this project is a Seagate Barracuda 7200.12 SATA, which has a 125 MB/s sustained data rate. [29]
4 Hardware-accelerated graphics

Real-time video playback, and especially 3D video playback, places high demands on the system hardware as described in chapter 3.4. Hence, it is vital to design software that incorporates high hardware utilization and efficient algorithms. Most video cards nowadays are very sophisticated with dedicated graphical processing units (GPUs) enabling hardware-accelerated graphics to unload the CPU. Since video playback and video decoding is very computationally intensive it is crucial to take advantage of hardware-accelerated graphics.

Two industry standard APIs exist for hardware accelerated graphics in desktop computing: Microsoft DirectX and Khronos Group OpenGL. However, Microsoft DirectX APIs is proprietary and only supports Microsoft platforms such as Windows OS and Xbox. [30] Hence, this section presents an overview of the OpenGL framework and a brief introduction to modern video card hardware.

4.1 Video card overview

Modern video cards consist of several sub-components in which the most essential are: Graphics processing unit (GPU), video memory and one or several output ports to which display devices are connected. The GPU is a dedicated processor to optimize graphics processing and is usually a specialized single-instruction-multiple-data (SIMD) architecture processor in which each processing unit executes the same instructions at any given clock cycle. This is very efficient for graphics, as each processing unit can operate on different graphical data elements, for example pixels or vertices simultaneously. [31]

Modern video cards process graphical data in different processing stages which constitute the rendering pipeline. There exist both fixed-function pipelines (most common on earlier video cards), where the processing stages are fixed, and programmable pipelines, where the rendering process might be customized by uploading small programs to the GPU chipset. These programs are also known as shader programs or simply shaders.
The rendering pipeline produces computer imagery from 3D geometry, textures and other data through projection, transformation and other matrix operations on 3D geometry which is defined by points in space usually referred to as vertices. Vertices are combined into triangles that define surfaces in model space. Vertex data is stored in vertex and index buffers in video memory.

Modern programmable rendering pipelines can be divided into three main stages: vertex processing, fragment processing and raster operations. An important difference between different GPUs is the number of vertex and fragment processing units. The performance of a GPU is heavily dependent on the capabilities of these processors, but also on the GPU clock frequency and video memory speed.

4.1.1 **Vertex processor**

Modern video cards use a vertex processor to transform 3D geometry (vertex data) to screen pixels. This involves several different processing steps:

- **Vector and scalar processing, primitive assembly** – Converts all vertex coordinates from model space to world space and these are combined in the primitive assembly stage.

- **Back-face culling and clipping** – Removes triangles that are pointing away from the observer (back faces) or are not visible to the observer (clipping).

- **Viewport transform and triangle setup** – Changes world coordinates into viewing space coordinates which are then combined into triangles.

The output result from the vertex processor is visible and lit triangles that are ready to be scan-converted by the rasterizer which in turn translates the triangle output into pixels. However, for programmable pipeline architectures, vertex shader programs make it possible to bypass or modify the fixed pipeline stages described above by customized transforms implemented in vertex shader programs. [15] This makes it virtually possible to process any data stored in vertex buffers by a vertex shader program. It does not necessarily have to be geometry data if the
intention is to process arbitrary data within the GPU instead of generating 3D imagery. This approach may be used to unload the CPU by assigning general purpose calculations to the GPU instead of the CPU.

4.1.2 Fragment processor

The fragment processor (also referred to as a pixel shader) is a processor in which a pixel shader program can be loaded to operate on pixel data. Pixel shader programs are executed in parallel by several fragment units to increase performance. Pixel shader programs receive per-pixel input data such as color and alpha from textures stored in video memory. The pixel shader usually performs texture mapping from texture coordinates as well as additional pixel processing and filtering before the pixel result is written to the frame buffer or render target. Pixel shaders have access to texture memory and are able to conduct pixel data look-ups in any texture that is fetched into a texture unit in the GPU. A texture cache memory is used to improve processing performance even further by storing texture data retrieved from a texture look-up or sampling operation. [30] The GPU texture cache exists primarily to accelerate filtering and hence is only required to be as large as the filtering kernel for the texture sampler, resulting in a very small cache designed for locality in two dimensions. This can be contrasted to CPU cache memories which are much larger and operate on higher clock frequencies. [32]

The number of fragment processing units determines how many pixels may be processed in parallel. However, no dependencies are allowed between pixels. Most video cards are equipped with several texture samplers which means that texture data from several textures might be combined. Modern fragment shaders also support branching constructs, handled by a special branch processor.

As for vertex shader programs, pixel shaders might be used to process any data that is stored as a texture in video memory. However, pixel shader programs execute in parallel on SIMD architecture and can only write to the pixel in the frame buffer that the current program is executed for, which is internally controlled by the GPU.
4.2 OpenGL

OpenGL [11] is a cross-language, cross-platform API for writing applications that produce 2D and 3D graphics. OpenGL was originally developed by Silicon Graphics Inc., but is now managed by the Khronos Group. OpenGL is built as a hardware abstraction layer and is constructed as a graphics pipeline, which is also known as the OpenGL state machine. Most OpenGL commands sends primitives to the OpenGL pipeline or configure how the pipeline will operate on the primitives. [33] Programmable video card rendering pipelines are supported through OpenGL shader language (GLSL) which makes it possible to write and load customized vertex and fragment shader programs into the video card chipset [34].

OpenGL allows vendors to supply extensions to the OpenGL library when new video card hardware technology is released. Extensions may introduce new functions and constants related to a specific technology or standard. Special libraries such as GLEW and GLEE have been developed to make it easier for developers to use OpenGL extensions.

Several libraries and APIs have also been developed on top of OpenGL to provide functionality not present in OpenGL itself. Some examples of such libraries are GLUT and GLEW which supplies cross-platform functions for accessing operating system routines for window and GUI management, access I/O devices etc.

Several versions of OpenGL have been released throughout the years. The first version of OpenGL was 1.0 in 1992 with very limited functionality and the latest official release was 3.2 in August 2009 which added support for new technology, such as for example geometry shader support. All releases of OpenGL are backward compatible. [33]
5 Methodology

The high system requirements for 3D video processing and the lack of 3D video format standards requires flexible, yet efficient algorithms. In combination with requirements of cross-platform interoperability this affects both the design and implementation of 3D video playback software. To evaluate the proposed solution it is important to determine whether it is sufficiently efficient to handle the throughput requirements described in chapter 3.4 as well as to quantify the degree of cross-platform interoperability in order to evaluate how confirmative the software is to changes in hardware and underlying operating system. The following chapter describes methodology and measurements for the evaluation of the 3D video playback solution.

5.1 Experimental methodology to evaluate performance

To measure and evaluate the performance of the 3D video player solution, a number of different 3D video files should be used to find anomalies in the performance which are dependent on the encoding and compression format, video resolution and number of views. Several different video files should be used to increase the credibility of the results. The video files used for evaluation of this work are presented in detail in Figure 13. It is noticeable that the video files described in Figure 13 is a set of multi-view video files available for free on the web, as well as video files supplied by Mid Sweden University. The number of disparate views in the multi-view video files ranges from two to eight and the files include both tiled and multi-stream formats.

The throughput requirements stated in chapter 1.4 are quantified in frames per second (FPS). The average frame rate \( f_{AVG} \) can be measured by disabling video synchronization and the vertical synchronization (VSync) of the display in order to determine the total processing time \( t \) for the video and dividing this by the number of processed frames \( N \) by the time \( t \) according to:

\[
f_{AVG} = \frac{N}{t} \ (frames/sec)
\]  

(5.2)
Hardware timers are used directly in the implementation of he source code in order to measure \( t \), hence minimizing the overhead caused by the profiling routines.

<table>
<thead>
<tr>
<th>Bitrate (Mb/s)</th>
<th>Average FPS</th>
<th>Peak FPS</th>
<th>V</th>
<th>S</th>
<th>Width</th>
<th>Height</th>
<th>MB/s</th>
<th>MB/s</th>
<th>Codec</th>
<th>AR</th>
<th>Duration</th>
<th>Size (MB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>A 12.00</td>
<td>6.6</td>
<td>960</td>
<td>696</td>
<td>4.01</td>
<td>24</td>
<td>96.22</td>
<td>288.65</td>
<td>MP4V</td>
<td>1.38</td>
<td>0.234</td>
<td>225.76</td>
<td></td>
</tr>
<tr>
<td>B 16.00</td>
<td>8.8</td>
<td>800</td>
<td>480</td>
<td>3.07</td>
<td>30</td>
<td>92.16</td>
<td>276.48</td>
<td>MP4V</td>
<td>1.67</td>
<td>0.41</td>
<td>81.39</td>
<td></td>
</tr>
<tr>
<td>C 16.08</td>
<td>8.8</td>
<td>800</td>
<td>600</td>
<td>3.84</td>
<td>30</td>
<td>115.20</td>
<td>345.60</td>
<td>MP4V</td>
<td>1.33</td>
<td>0.14</td>
<td>27.48</td>
<td></td>
</tr>
<tr>
<td>D 6.60</td>
<td>5.5</td>
<td>640</td>
<td>480</td>
<td>1.54</td>
<td>30</td>
<td>46.08</td>
<td>138.24</td>
<td>MP4V</td>
<td>1.33</td>
<td>0.08</td>
<td>7.82</td>
<td></td>
</tr>
<tr>
<td>E 15.98</td>
<td>8.8</td>
<td>760</td>
<td>456</td>
<td>2.77</td>
<td>30</td>
<td>93.17</td>
<td>249.52</td>
<td>MP4V</td>
<td>1.67</td>
<td>0.33</td>
<td>65.68</td>
<td></td>
</tr>
<tr>
<td>F 14.95</td>
<td>6.6</td>
<td>832</td>
<td>624</td>
<td>3.12</td>
<td>25</td>
<td>77.88</td>
<td>233.63</td>
<td>MP4V</td>
<td>1.33</td>
<td>0.19</td>
<td>145.89</td>
<td></td>
</tr>
<tr>
<td>G 4.00</td>
<td>6.00</td>
<td>720</td>
<td>576</td>
<td>0.83</td>
<td>25</td>
<td>20.74</td>
<td>62.21</td>
<td>WMV3</td>
<td>1.25</td>
<td>0.03</td>
<td>56.74</td>
<td></td>
</tr>
<tr>
<td>H 6.00</td>
<td>9.00</td>
<td>1280</td>
<td>720</td>
<td>1.84</td>
<td>25</td>
<td>46.08</td>
<td>138.24</td>
<td>WMV3</td>
<td>1.78</td>
<td>0.03</td>
<td>85.16</td>
<td></td>
</tr>
<tr>
<td>I 8.00</td>
<td>12.00</td>
<td>1440</td>
<td>1080</td>
<td>3.11</td>
<td>25</td>
<td>77.76</td>
<td>233.28</td>
<td>WMV3</td>
<td>1.33</td>
<td>0.03</td>
<td>111.07</td>
<td></td>
</tr>
<tr>
<td>J 6.00</td>
<td>9.00</td>
<td>1440</td>
<td>480</td>
<td>0.69</td>
<td>30</td>
<td>20.74</td>
<td>62.21</td>
<td>WVC1</td>
<td>1.78</td>
<td>0.10</td>
<td>48.22</td>
<td></td>
</tr>
<tr>
<td>K 10.00</td>
<td>20.00</td>
<td>1280</td>
<td>1440</td>
<td>1.84</td>
<td>30</td>
<td>55.30</td>
<td>165.89</td>
<td>WVC1</td>
<td>1.78</td>
<td>0.10</td>
<td>80.50</td>
<td></td>
</tr>
<tr>
<td>L 30.00</td>
<td>60.00</td>
<td>1520</td>
<td>2160</td>
<td>4.15</td>
<td>30</td>
<td>124.42</td>
<td>373.25</td>
<td>WVC1</td>
<td>1.78</td>
<td>0.10</td>
<td>239.08</td>
<td></td>
</tr>
<tr>
<td>M 19.35</td>
<td>19.35</td>
<td>1920</td>
<td>1080</td>
<td>2.07</td>
<td>30</td>
<td>62.21</td>
<td>186.62</td>
<td>MPG4V</td>
<td>1.78</td>
<td>0.26</td>
<td>326.31</td>
<td></td>
</tr>
</tbody>
</table>

Figure 13: List of video files used for experimental evaluation.

A set of 4 evaluation measurements are used to quantify the performance of the playback solution based on the throughput requirements stated in chapter 1.4:

- **FPS as a function of bitrate** – Such an analysis will indicate whether the hard drive access time is a limiting factor for 3D video playback. Throughput, in terms of average frame rate \( f_{\text{avg}} \), is measured as a function of the varying video bit rate by measuring video files with different resolutions and varying compression formats implying different data bit rates.

- **FPS as a function of resolution** – This metric indicates whether high video resolution is a potential bottleneck for 3D video playback. Throughput, in terms of average frame rate \( f_{\text{avg}} \), is measured as a function of the total pixel resolution by measuring video files with different horizontal and vertical resolutions.
• **FPS as a function of the number of views** – This measurement will suggest whether the number of views in a multi-view video affects the performance of the GPU for 3D video playback. Throughput, in terms of average frame rate \( f_{AVG} \), is measured as a function of a varying number of views by simulating tiled format using a single 1080p video to keep bit rate, compression format and resolution constant.

• **Load balance** – By measuring the average processing time per-frame for each process or thread in the proposed software solution, a balance between CPU and GPU utilization may be determined so as to indicate whether the implementation has efficient hardware utilization or not.

In order to evaluate whether the throughput requirements are fulfilled it is convenient to compare average frame rate \( f_{AVG} \) to the native frame rate of the video. A throughput or update frequency of 27 frames/second for a video encoded in 30 frames per second is considered as inferior, while a throughput of 27 frames/second for a video encoded in 25 frames per second is considered as acceptable. The difference between the maximum average frame rate \( f_{AVG} \) and native frame rate \( f_{NATIVE} \) is defined as:

\[
\Delta f = f_{AVG} - f_{NATIVE} \text{ (frames/sec)}
\]  \hspace{1cm} (5.3)

Hence, if \( \Delta f < 0 \), the system is not able to playback the video at the sustained frame rate. On the other hand if \( \Delta f \geq 0 \), the system fulfills the performance requirements. The performance is highly dependent on the software design and implementation as well as the capabilities of the hardware, but the experiments within this work are limited to be performed on one machine only as further specified in chapter 5.4.

Experiments should be conducted for all video files represented in Figure 13 and repeated several times to increase the credibility of the measured execution time. It might be tempting to measure the execution time of each frame in order to be able to calculate the standard deviation of the processing time on a per-frame basis, but this is not practical since it would require time stamps to be saved into memory for every frame.
This would imply greater overhead due to memory access time and dynamic memory allocation time.

5.2 Evaluation of cross-platform interoperability

One of the requirements stated in section 1.4 is that the 3D video playback solution supports cross-platform interoperability. This requirement is directly dependent on the libraries used to realize the video playback software and if any system specific function calls are required. System dependent function calls should be implemented with pre-processor conditional branching or an equivalent solution to enable cross-platform compilation. Implementing cross-platform compilation capabilities for all operating systems where system calls are required is not possible with the limited time available for this thesis project, but pre-processor conditional branching should be implemented everywhere in the source code where system calls are indispensable in order to highlight the necessity for future extension of platform support. However, no system calls should be used that do not have equivalents on other platforms.

To evaluate the cross-platform capabilities of the video playback solution, a table should be created based on all libraries used to realize the software. The supported platforms are then considered to be the set of platforms that are supported by all libraries.

5.3 Verifying sub-pixel spatial multiplexing

In order to verify that pixels are processed and spatially multiplexed correctly for the lenticular 25-view display (LG “42, 25-view lenticular display) used for experiments, available image pairs of deinterlaced and interlaced 25-view tiled multi-view video (supplied by Mid Sweden University) will be used as the system input and for output validation respectively. By feeding the proposed system with the deinterlaced image, the output should be equivalent to the interlaced counterpart, which is determined by sub-pixel component arithmetic subtraction.

5.4 Hardware and software resources

The system used for implementations and experimental measurements is a machine from Dell containing the following hardware: Intel Xeon
E5520 2.25 GHz CPU, 2.50 GB RAM, NVIDIA Quadro FX 580 graphics card and a Seagate Barracuda 7200.12 SATA 3Gb/s 250GB hard drive. The primary reason for this is that it is the only system available during this project and that experiments on several systems falls outside the scope of this work. However, this system is considered as a standard high-end system to the current date and is likely to produce experimental results that are similar to those for any high-end desktop computer today. See Appendix A for more details about system hardware and the software used for development and experiments.

Throughout this work it will be possible to access two different 25-view lenticular displays from LG in order to verify functionality subjectively.
6 Design

The high performance requirements of 3D video processing places high demands on both software architecture and algorithm efficiency. On the other hand, the lack of standards and emerging 3D formats addresses the need to provide a generic video format representation. This makes it vital to carefully consider both performance and flexibility when designing a 3D video playback solution. The different design considerations taken and the proposed 3D video playback solution are presented in this chapter.

6.1 Alternative design solutions

Most standard video player software is built upon one or several demultiplexer, decoder and hardware accelerated graphic libraries. The solution that has been created within the scope of this thesis is a stand-alone 3D video player application built on top of FFmpeg [5], using libavformat for its media container demultiplexing and libavcodec for its video decoding. However, two different software solutions were considered before it was decided to create a stand-alone application:

- Virtual demultiplexer and decoder – Integrating 3D video processing logic within the FFmpeg library, enabling standard 2D video players built upon dynamically linked FFmpeg libraries to display 3D video.

- Stand-alone 3D video player application – Creating a stand-alone 3D video player application built on top of FFmpeg, using OpenGL for hardware accelerated graphics.

In order to create a highly portable solution with minimal effort, the integration of 3D video processing within an open-source demultiplexer and decoding library such as libavformat and libavcodec of FFmpeg is desirable. Since standard video players may not support multi-stream video, a virtual demultiplexer could be used as a wrapper around the actual media container demultiplexer of FFmpeg. In this manner the virtual demultiplexer would manage both demultiplexing and decoding
of video streams. The logic for interlacing the multi-view video could then be used as a final stage for the decoded video frames in order to generate a single video stream of spatially multiplexed video. The virtual video stream with already decoded spatially multiplexed video and non-video streams can then be made visible to the application that instantiated the virtual demultiplexer, but the actual video streams are not visible to the application as illustrated in Figure 14.

Virtual demultiplexer

![Diagram of Video Demultiplexer and Video Decoder Concept for 3D Video Processing](image)

Figure 14: Virtual demultiplexer and video decoder concept for 3D video processing.

Since the video stream with spatially multiplexed video has already been decoded after being processed by the virtual demultiplexer it is also vital to provide a virtual video decoder that simply outputs video frames generated by the multi-view multiplexer logic to the application.

The concept associated with a virtual demultiplexer as illustrated in Figure 14 appears to be a solid solution at first glance, but closer investigation reveals complexity and obscurity. Modern standard video players as for example VLC Video Player [21], rely on several different libraries for demultiplexing and decoding, and use different combinations of these depending on the container type and encoding format. Other solutions such as FFShow Tryouts [25] use libavcodec decoders, but rely on DirectPlay for demultiplexing. Hence, it is not possible to exclusively perform 3D video processing inside a decoder if multi-stream video formats are used. However, it would be possible to move all logic into the virtual demultiplexer under the condition that the spatially multiplexed video stream is encoded again after it has been
processed. This would imply that a regular video stream could be provided as output and eliminate the dependence between the demultiplexer and decoder. However, problems arise even in this scenario. Neither the demultiplexer nor the decoder manages image scaling nor has any information regarding the size of the video application window (scaling is typically done after the decoding stage). The solution may work if the video resolution is perfectly matched with the screen resolution, but since data has to be processed within the virtual demultiplexer, decoded, processed within the GPU, sent back to the system RAM and then encoded again, the performance is very likely to suffer and the solution would not be sufficiently flexible to support different display configurations. Such a solution might also face problems with thread-safety or cause load imbalance if used in conjunction with a multithreaded video player application.

Creating a stand-alone 3D video player application, which is the solution chosen for the prototype software presented in this thesis, is a much more convenient and straight-forward solution than a virtual demultiplexer since it allows full control of demultiplexing, decoding and scaling as well as minimizing the overhead. However, it requires more work since synchronization, sound support and additional video player functions must also be implemented in order to create a working video player.

### 6.2 Design considerations and system design overview

To increase modularity and flexibility it is convenient to separate the 3D video processing from the video player functionality. Therefore the 3D video player application has been divided into two separate software components:

- **Video player component** – Containing high level video player functionality, for example: media container demultiplexing, decoding and synchronization.

- **3D video filter component** – Containing 3D video processing logic in the context of spatial multiplexing, RAM to video memory pixel data copying, GPU shader management etc.
The video player has been designed to use the 3D video filter as an intermediate composite component in the video processing pipeline. This is conceptually illustrated in Figure 15.

![Conceptual composite structure of the 3D video player application](image)

**Figure 15:** Conceptual composite structure of the 3D video player application, taking a multi-view video media container file as input and rendering 3D video to a display as output.

From Figure 15 it is also clearly visible that the GUI system has been separated from the 3D video player component. The reason for separating the rendering of the video output from the 3D video player component is that the 3D video player component becomes a stand-alone component in itself, only loosely-coupled with the graphical user interface (GUI) rendering window. Hence the GUI window system can easily be interchanged, which is an advantage for cross-platform interoperability as well as modularity. This design also allows the 3D video player logic to be used as an integrated component in a video application that renders output to the screen or to be used in any application context to send spatially multiplexed 3D video output data to any I/O-device or file.

The 3D video filter or processing component can abstractly be seen as a black-box which takes a multi-view video signal and a display configuration as its input, transforms it and produces an output video signal that is a valid spatially multiplexed video signal for the particular display as illustrated in Figure 16.
6 Design considerations for high performance

The video encoding, compression format and file size is crucial for system throughput as stated in chapter 3.4. Low compression or no compression at all will decrease or make minimal impact on the CPU load while decoding video, while, at the same time, file size will increase and hard drive access time may constitute a primary bottleneck. High compression will in turn result in a high CPU load as an effect of the complex video decoding algorithms, while the file size will decrease and the hard drive access will not be as dominant. However, these parameters are format and hardware bound and cannot be affected by the video player design. Instead the video player is designed to provide high throughput and minimal additional processing apart from media container demultiplexing, video decoding, spatial multiplexing and rendering. This is achieved by means of light-weight data multiplexing and pre-computation of as much data as possible. Another critical aspect in memory intensive real-time systems is efficient memory access. Therefore, memory copy routines have been optimized and implemented to maximize data locality in order to enable high CPU cache utilization.

Sequential processing is likely to result in execution bursts for the different processing stages. Therefore, multithreading is used to take advantage of OS context-switching and time-sharing mechanisms to execute different video processing stages concurrently which will result in smoother execution. On multi-core or many-core systems this will also increase hardware utilization and may provide better load balance depending on the OS thread scheduling algorithms.
6.2.2 Design for flexibility and platform independence
A high degree of cross-platform interoperability requires that only
libraries targeted for multi-platform usage are integrated into the
software. For that reason, only libraries and frameworks that have rich-
support for cross-platform compilation are used within this project.

When considering the choice of programming language for
implementation, C was chosen since FFmpeg is written in C. There was
greater motivation for this choice based on both high execution
performance as well as a high degree of cross-platform interoperability
as C compilers are available for most operating systems.

To increase flexibility, a component based design approach has been
used. This makes it possible to reuse different logical components and
creates a distinctive structure for the framework. In order to enable new
video processing algorithms to be added to the system architecture, a
plug-in architecture is also used in the design of the 3D video filter.

6.3 3D video player component design
The 3D video player component has been designed on top of the video
demultiplexer library *libavformat* and video decoder library *libavcodec* of
FFmpeg. The video player architecture and algorithm design presented
here is heavily influenced by the designs proposed by S. Dranger [6] and
M. Böhme [7]. However, native OpenGL [11] and POSIX threads [35] are
used instead of SDL for hardware accelerated graphics and
multithreading respectively. The design presented here has also been
heavily modified to better comply with 3D video processing and
playback of multi-view content.

6.3.1 Architectural overview
The 3D video player data flow can be defined by five distinctive steps.
This makes it convenient to design the video player logic as a processing
pipeline with five separate processing stages, involving:

- *Demultiplexing* - Read data packets from media container source
  file and demultiplex data streams.
• **Video decoding** – Decode video data packets to video frames represented as pixel arrays.

• **Color conversion** – Convert the video frame color format from native format to RGB representation.

• **Filtering** – Process all views through the 3D rendering filter to achieve a display dependent bitmap video frame.

• **Synchronization and rendering** – Use synchronization data associated with each video frame to synchronize video rendering to a hardware clock.

Each processing stage of the video pipeline is executed on a separate thread for smooth throughput and increased hardware utilization. Blocking queues are used to provide thread-safety and mutual exclusion through locking mechanisms between each processing stage as illustrated in Figure 17.

![Video player system architecture](image)

**Figure 17:** Video player system architecture. Each processing stage is executed on a separate thread to enable smooth throughput and high hardware utilization.
In the case of multi-stream video, a separate packet queue, video frame queue and RGB video frame queue is created for each stream. By specifying a maximum capacity for each queue, the producer thread will block until the consumer thread has popped an item from the queue and left space for the producer thread to add yet another item. This producer/consumer pattern prevents race conditions and enables a fixed size memory to be used for the queue buffers.

6.3.2 Demultiplexing
The `libavformat` library of FFmpeg contains functionality for packet-oriented demultiplexing of streams within a media container. The demultiplexer thread is responsible for reading data packets from the media container file and pushes packets to the packet queue of each video stream. One packet queue exists for each stream. Non-video stream packets, such as audio stream packets or other types of stream packets should be handled separately or be discarded. No matter how many video streams exist in the media container only one demultiplexer thread is created. The reason for this is that file access routines would not benefit from further parallelization, since the operation is I/O related and not computationally intense.

6.3.3 Video frame decoding
The `libavcodec` library of FFmpeg is used to decode video stream packets produced by the demultiplexer. A video frame may consist of one or several data packets depending on the video encoding format. The video decoder thread iterates over all video streams and decodes packets until a complete video frame can be retrieved. The video frame is then pushed to the video frame blocking queue and the process starts over again.

Video textures (video frames) are represented as pixel arrays and hence requires memory to be allocated on the heap to store pixel data. Since memory allocation and reallocation wastes many CPU cycles, video frame data buffers are allocated in a memory pool when acquired and returned to the pool when the buffer is no longer required. Thus memory blocks are reused, which minimizes memory allocation and reallocation overhead.
The pixel buffer memory pool must be synchronized as buffers are released from the color conversion thread which is the next step in the video processing pipeline (see chapter 6.3.4). Locking mechanisms are also used to provide thread-safety for memory pool access. The main mechanisms of the decoder thread algorithm are illustrated in Figure 18. It should be noted that a pixel buffer memory pool is created for each stream to support non-uniform view resolutions among video streams.

![Diagram of video processing pipeline](image)

**Figure 18:** The video decoder thread consumes video stream packets and produces video frames represented as pixel arrays.

### 6.3.4 Color conversion

The video decoder produces and enqueues video frames/textures represented as pixel data in a buffer. However, the color space depends on the codec and it is probable that this is not represented as a GPU compatible format. Hence, if the color format is non-RGB, it must be converted to RGB using a transformation function. For example, the YCbCr color space, or more generally YUV, is used by many video codecs as it takes human perception into account and defines colors in terms of luma (Y) and chrominance (CbCr) [36].

The *sws* library of FFmpeg has built in functions for color conversion from YCbCr and several other formats to 24-bit RGB. The color conversion thread algorithm consumes native color space video frames and transforms them to RGB using a conversion function. Different color spaces may require different number of bytes to represent pixel data and hence there is a requirement to allocate new memory for RGB pixel data. In a similar manner to the video decoding mechanism a memory pool is used to minimize overhead as illustrated
in Figure 19. Note that a pixel buffer memory pool is created for each stream to support non-uniform view resolutions and independent color formats.

Figure 19: The video colour conversion thread consumes native colour space video frames and transforms them to RGB video frames.

6.3.5 Synchronization, filtering and rendering

All video streams have a determined frame rate bound to them that specifies the speed of display for the video frames. However, if video is synchronized by simply counting frames and multiplying by frame rate, audio and video may fall out of sync with each other. Instead, packets from a video stream might have a decoding time stamp (DTS) and a presentation time stamp (PTS). These two timestamps are closely related to the manner in which video is encoded, where the DTS specifies when to decode and the PTS specifies when to display. [6]

When a packet from a video stream is received from the media container demultiplexer it will contain the PTS and DTS values for the information inside that packet. However, the libavcodec library reorders packets internally and might return invalid PTS values. Hence, a workaround is required in order to synchronize the video.

S. Dranger [6] presents a video synchronization algorithm that is also used in this 3D video playback solution. The algorithm stores the PTS of the first packet of a video frame and this specific PTS will be the PTS of the video frame when it has been completely decoded. So if the stream does not have a DTS, the stored PTS will be used instead. In order to synchronize video, a delay threshold value is used together with a
hardware clock. When a frame is displayed (rendered to screen) the threshold value is calculated for the next frame in relation to the hardware timer and the PTS of the next frame. When the set amount of time has passed, the next frame is displayed and the same procedure is repeated again. However, there is a problem in knowing when the next PTS will occur. The video frame rate could be used to calculate the next PTS from the current PTS and the frame rate, but certain video formats call for frames to be repeated, so this is taken into account.

In order to render graphics to the display, OpenGL [11] is used to benefit from cross-platform hardware accelerated graphics. However, OpenGL requires an OS-dependent window context in order to display graphics. The OpenGL Utility Toolkit (GLUT) [37], provides cross-platform abstraction of window management routines, vertical synchronization and additional GUI related functions. To avoid thread-safe synchronization of OpenGL functions, i.e. access to the GPU, the filter for spatial multiplexing is triggered when the synchronization mechanism calls for a new video frame to be displayed. Hence, both the synchronization algorithm and filter will run on the same thread. When a video frame should be displayed, an RGB video frame is popped from the video frame queue and passed to the filter. In order to improve performance, the texture read-back is bypassed (see chapter 6.4.3 for details) and the filter output texture in the frame buffer is rendered to the screen directly from video memory. This is conceptually illustrated in Figure 20.

Figure 20: The synchronization and rendering thread consumes RGB video frames and renders a spatially multiplexed equivalent to the screen.
6.4 3D video filter pipeline

6.4.1 Defining input and output

There are no accepted standards for 3D-video formats or 3D-video codecs, which address the requirement for flexible and configurable input and output formats. The 3D video filter is designed to be used as an output filter for a video or image decoder and constitutes the last processing stage in the video pipeline of the proposed 3D video player application. In the absence of standards and well-defined formats, the application is responsible for recognizing the input format and providing appropriate parameters to the input filter. This implies that the structure of the input format may vary, while the filter output is limited to be a two dimensional pixel array, representing a spatially multiplexed video frame.

The input to the 3D video filter may be represented as a tiled multi-view format in a single texture or by multiple video streams where each video stream texture corresponds to a disparate view. For tiled formats, mapping coordinates for the different views must be provided to the filter. This is necessary in order to locate the position of the different disparate views within the tiled texture map. To be consistent, it is also convenient to assemble the multiple textures in multi-stream formats into a tiled texture. This is also advantageous since video cards typically have no more than four texture units to simultaneously retrieve textures. The assembly process of merging multiple view textures into a tiled format may be straight-forward for views of equal resolution, but may be complex to accomplish efficiently for frames of different sizes. To perform efficient assembly of non-homogenous view textures a packing algorithm could be used, but since the format is assumed to be known by the decoder, the assembly structure could also be constructed as an input parameter to the filter. The latter method is the approach used in the prototype implementation, primarily due to its simplicity and that it is a straightforward method that simplifies the filter, while additionally not causing a reduction in flexibility. Hence all input formats may be represented by a texture mapping data structure, which are sufficiently generic to provide a consistent input management of textures (views) within the rendering framework.
6.4.2 Filter input parameters

The 3D video filter requires information concerning the display configuration, the pixel data of the multiple views and a description of the multi-view format in order to interpret and process 3D video. For example, for a video with \( N \) disparate views, the filter must be fed with pixel data for all \( N \) views every frame. As long as the video format does not change during playback, this only has to be set once. This avoids redundant calculations. Depending on the format of the video, \( N \) texture mapping coordinates corresponding to the views must also be provided (See section 6.4.4 for details on mapping coordinates). This is required to assemble all \( N \) views into a single tiled texture.

In addition to providing format description data, it is also convenient to provide parameters for the output resolution in order to enable scaling. This is necessary if the native video resolution does not match the display resolution and thus resampling of the video texture to fill the screen is desired.

The function to multiplex \( N \) disparate views into a single spatially multiplexed texture is display dependent. However, it can be represented by a mathematical function which describes which pixel or pixel RGB-component to map from a view texture to a certain pixel in the spatially multiplexed texture. The function will not change as long as the display is not interchanged, which allows for pre-calculation and caching of the results from such a function. The result can be represented as a pixel mapping table (PMT) which eliminates the need for re-computation and simplifies the mapping process to simple table lookups which, for reasons related to performance, is a good choice.

To further increase flexibility, the generation of a PMT can be accomplished by providing source code for a fragment shader program that calculates the PMT and stores it as a texture in video memory. This also enables the shader program to be interchanged during run-time and compiled on the fly if the video format, display type or display resolution changes.

An overview of all the filter input and output parameters is illustrated in Figure 21. Note that semi-static and dynamic data are separated for clarity.
6.4.3 Video filter processing pipeline

The data flow through the video filter processing pipeline is initiated when a set of multi-view pixel data is passed to the filter function, as illustrated in Figure 22. The multi-view pixel data is then copied to video memory using a pixel buffer object (PBO) circular buffer and in the case of multi-stream video, assembled to a tiled format directly into the PBO. Due to the logical structure of the PBO pool (as described in detail in chapter 6.4.5), a number of $M$ iterations, i.e. $M$ calls to the filter function, are required before the intended output is received. However, this will not be a problem for video playback, as the $M-1$ first output textures can simply be discarded and two dummy sets of pixel data can be inserted at the end of the video frame sequence to flush the circular buffer.

For each call to the filter function, an input texture will be made available in the video memory. An optional stack of function pointers allows pre-processing of the input texture by adding callback routines that process the input texture directly in the video memory.

The tiled input texture is then passed to a fragment shader performing spatial multiplexing by performing texture lookups in the PMT texture and rendering RGB-components from the tiled input texture to the output texture using the OpenGL off-screen frame buffer object (FBO) extension (See chapter 6.4.6 for details). The output texture is then written to another cyclic PBO buffer structure that introduces yet another delay that is linearly proportional to $K$ and asynchronous DMA transfers are yet again used to read back pixel data of the output texture to the CPU controlled memory as illustrated in Figure 22.
Figure 22: The 3D video processing filter data flow illustrating texture transfers between CPU controlled memory and GPU controlled memory.

If rendering directly to a display is intended it is possible to bypass the readback-mechanism and access the output texture in the FBO directly to avoid wasting hardware resources. This feature is used by the 3D video player component described in chapter 6.3. Note that the FBO object is required to enable rendering without any window context handle as mentioned by Song Ho Anh [38].
6.4.4 Generic multi-view texture mapping coordinates

OpenGL relies on normalized texture mapping coordinates [11] and this makes it convenient to represent all coordinates as normalized coordinates, i.e. in the range [0, 1]. A partial rectangular texture can be described by two coordinate pairs: \((u_1, v_1)\) and \((u_2, v_2)\) describing the normalized coordinates of the bottom-left corner and upper-right corner in a rectangle according to equation 6.4:

\[
0 \leq u_1 \leq u_2 \leq 1 \\
0 \leq v_1 \leq v_2 \leq 1
\]  

(6.4)

By providing a set of normalized texture coordinates, all tiled formats may be defined. For example, for a tiled format with two views that share horizontal space, this would imply a set of two coordinate pairs as defined by equation 6.4: \((0.0, 0.0), (0.5, 1.0)\) for the leftmost view and \((0.5, 0.0), (1.0, 1.0)\) for the rightmost view as illustrated in Figure 23.

![Figure 23: Texture coordinates specifying the view texture alignment within a tiled multi-view texture.](image)

Multi-stream formats on the other hand are assembled into an internal tiled format where the texture coordinates in this case describes the actual assembly or pixel mapping process, which can be generated automatically.

6.4.5 Texture transfers

Transferring a texture to and from video memory involves both loading the texture data from the source and copying it to the OpenGL controlled video memory. A conventional texture transfer method involves the CPU to firstly read the texture data from the source into
RAM and then to copy the texture data from RAM to the OpenGL controlled memory. In the 3D video filter, copying may also involve the assembly of a multi-stream video format into a tiled video format. This copy operation is illustrated in Figure 24 where $V_n$ denotes the $n$:th source texture (pixel array data) or collection of multiple view textures that constitutes the input data for video frame $n$.

![Figure 24: Conventional texture transfer from source to the graphics card.][39]

The approach illustrated in Figure 24 is simple and straightforward, but inefficient as the CPU will stall during the texture transfer from the RAM to the graphics card. The OpenGL extension pixel buffer object (PBO) enables fast texture transfers to and from the graphics card through asynchronous direct memory access (DMA) without involving CPU cycles [39] as illustrated in Figure 25.

![Figure 25: Texture transfer using pixel buffer object.][39]

To take advantage of asynchronous DMA transfers and to minimize the CPU workload, a dynamic pool architecture of PBOs is used for
uploading (a.k.a. *unpack*) and downloading (a. k. a. *pack*) pixel data to and from the video card memory. By aligning the pixel buffer objects in a circular list, asynchronous behaviour can be achieved by initiating a DMA transfer for one PBO and read or write data directly from the pixel buffer of another PBO in the list. It is noticeable is that a high number of pixel buffer objects will allocate video memory equal to the texture size of the video resolution times the number of PBO objects.

The CPU must still copy data to the OpenGL supplied PBO, but the actual transfer to the graphics card will be handled asynchronously through a DMA transfer. Hence the CPU will not stall during texture transfers and will be able to perform other computations while the texture is transferred. Run-time tests have indicated that using PBOs to transfer texture pixel data gives a texture transfer speedup of approximately 3.4 (see Appendix B) for a PBO list size of 4. The method could be improved even more by eliminating the memory copy from source to RAM, which could be achieved by implementing a call-back mechanism for the actual PBO unpack operation and by writing the source data directly into the PBO. However, this would require that the application implemented the actual copying function inside the video decoder. Direct copying to the PBO from the video decoder is left as an optimization that could be considered and implemented in future development.

### 6.4.6 Spatial multiplexing

For each pixel with coordinates \((u, v)\) in the output texture (spatially multiplexed texture), the zero-based index for the view to obtain each color component can be calculated from equation 6.5, which describes the sub-pixel mapping functions for a slanted 25-view lenticular display from LG.

\[
N_r = f_{r}(u,v) = (u \times 10 + 5 + r \times v) \mod 25 \\
N_g = f_{g}(u,v) = (u \times 10 + r \times v) \mod 25 \\
N_b = f_{b}(u,v) = (u \times 10 - 5 + r \times v) \mod 25
\]  

(6.5)

Run-time tests indicate that using a pre-computed static pixel mapping table to lookup the results of \(N_r, N_g\) and \(N_b\) during run-time is about 40 percent faster than re-calculating \(N_r, N_g\) and \(N_b\) each time the output
texture redraws. A similar technique was also proposed by E. I. Verburg [15].

By providing a constant that defines the number of available disparate views to the PMT shader program at shader compile time, it is possible for the PMT shader to decide how to distribute the different views among the stereo pairs that the display can visualize. This is only required if the number of views does not match the number of views supported by the display. Compiling the PMT shader program during application run-time also makes it possible to set other constant parameters apart from the available number of views. For example x and y screen offsets may be set if the rendering target is not aligned to the upper left corner of the screen.

The RGB-components of the precomputed PMT specifies the index of the corresponding view from which to obtain the sub-pixel components for the actual pixel as illustrated in Figure 26.

![Figure 26: A pixel mapping table (PMT) is a texture in video memory that uses RGB-components to store the index of the view pixel mappings at a sub-pixel level.](image)

Since each component of 32-bit RGBA vectors are limited to hold a floating point values in the range [0, 1], indexes have to be transformed into the range [0, 1] in the PMT-shader and then inversely transformed back to the range [0, 255] in the display shader that renders the final output texture. This is easily achieved by the normalization of indexes.
In practice this involves dividing indexes by 255 which is the maximum value of a zero-based 8-bit integer or byte.

### 6.5 Optimization details

Pixel copying routines from the CPU controlled memory to the GPU controlled PBO memory are optimized using loop-unrolling techniques for fast copying using explicit BGRA to RGB conversion. Several implementations with different numbers of byte array lengths are provided as speed will be dependent on the memory alignment and an automatic profiling function will select the most efficient memory copying function by quick run-time tests when the filter is initialized.

Pixel copying routines for texture assembly of multi-stream video formats have been optimized by copying complete horizontal lines using one function call.
7 Result

The experimental methodology and evaluation methods described in chapter 5 define a number of experiments and heuristics for evaluation of the 3D video playback solution. A summary of the results obtained from these experiments is presented in chapter 7.

7.1 Throughput as a function of video format bitrate

The graph illustrated in Figure 27 illustrates the average measured frame rate for varying video bit rates for the video files (A to L) presented in Figure 13. Multi-stream formats and tiled formats are represented by two different graphical symbols for clarity. The standard deviation $\sigma < 2.8$, was obtained for all measured video files.

![Figure 27: Throughput in terms of average number of produced frames per second for varying video bitrate (video synchronization disabled).](image)

Average number of frames per second for varying video bitrate

- Native frame rate
- Multistream format 5/6/8 views
- Tiled stereo format
- Linear regression

Bitrate [Mbit/s]

Average number of frames per second
The data represented in Figure 27 is complemented by a linear regression line plot in order to visualize the frame rate trend as the bit rate increases. According to the results obtained throughput is sufficient for all video files apart from the video file with a bit rate of 30 Mbit/s (video L, see Figure 13).

### 7.2 Throughput as a function of video resolution

The graph illustrated in Figure 28 illustrates the average measured frame rate for varying video resolution (pixel data rate) for the video files (A to L) presented in Figure 13. Multi-stream formats and tiled formats are represented by two different graphical symbols for clarity. A standard deviation $\sigma < 2.8$ frames per second was obtained for all measured video files.

![Image](image.png)

**Figure 28:** Throughput in terms of average number of produced frames per second for varying video pixel data rate (video synchronization disabled).

The data represented in Figure 28 is complemented by a linear regression line plot in order to visualize the frame rate trend as the pixel...
data rate increases. As can be seen in Figure 28 the obtained throughput in terms of frame rate is higher than the native frame rate for all videos apart from video L (see Figure 13).

### 7.3 Throughput as a function of the number of views

By means of the definition of the 3D video filter input parameter representation of disparate view texture coordinates, as described in chapter 6.4.4 and 6.4.2, the number of displays views $N$ can be varied by providing virtual coordinates to a simulated tiled format. The graph illustrated in Figure 29 illustrates the average measured frame rate for a varying number of views $N$, interpreting video file M presented in Figure 13 as a multi-view video while rendering it to a lenticular display. The standard deviation $\sigma < 3.7$ frames per second was obtained for all measured video files.

![Graph showing throughput as a function of the number of display views](image)

**Figure 29:** Throughput in terms of average number of frames per second for varying the number of display views for lenticular display rendering (Video synchronization disabled).
As can be seen in Figure 29, a second order best-fit polynomial approximation indicates a negative trend as the number of views, \(N\), increases.

### 7.4 Average frame rate in relation to native frame rate

The difference between average frame rate and native frame rate, \(\Delta f\), as defined in equation 2, is visualized in Figure 30 for video files A to L described in Figure 13. Since \(\Delta f > 0\) for all video files except video L (see Figure 13), the proposed 3D video playback solution manages to display the video at the intended frame rate for all video files regardless of whether the file is encoded as a tiled or multi-stream format.

![Figure 30: Difference between average and native frame rate.](image)

### 7.5 Load balance

The 3D video playback solution proposed in chapter 6 is based on four different threads implementing the producer/consumer design pattern. Hence, the work load balance between these threads will directly affect the performance. The graph illustrated in Figure 31 describes the
average processing time per video frame and thread for video files D, C and L described in Figure 13. The standard deviation $\sigma < 2.0$ ms was obtained for all measured video files.

![Average processing time per frame for the video player threads](image)

**Figure 31:** Average processing time per frame for the video different threads in the 3D video playback solution implementation.

The rendering and the system bus appear to be limiting factors as video processing and rendering on the GPU have the highest average processing times per frame. Video decoding and pixel conversion threads require approximately the same amount of processing time for a relatively low bit-rate and low-resolution video, but as video resolution and video bit-rate increases, the video decoding processing time appears to increase and approaches the processing time necessary in order to render video.

### 7.6 Cross-platform interoperability

The software design prototype proposed in chapter 6 is implemented in C89 and is built around Khronos Group OpenGL [11] for hardware accelerated graphics, FFmpeg [5] for video demultiplexing and decoding, the POSIX [40] standard for multithreading and GLUT [37] for window context management. FFmpeg and OpenGL was chosen as
the underlying libraries for video demultiplexing/decoding and hardware accelerated graphics based on the theoretical survey presented in chapter 2. The motivation behind this choice involves the high degree of cross-platform interoperability in addition to being distributed under an open-source license agreement. A summary of the third-party libraries and the supported platforms and hardware is presented in Table 2.

Table 2: Summary of operating systems and CPU architectures supported by the libraries and frameworks used to realize the 3D video playback prototype.

<table>
<thead>
<tr>
<th>Operating System</th>
<th>Linux</th>
<th>Apple Mac OS</th>
<th>Microsoft Windows</th>
<th>Unix/X</th>
<th>OS/2</th>
<th>BSD/OS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Software library / framework</td>
<td>Ffmpeg</td>
<td>OpenGL</td>
<td>GLUT</td>
<td>POSIX</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Linux</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Apple Mac OS</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Microsoft Windows</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes*</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Unix/X</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td></td>
<td></td>
</tr>
<tr>
<td>OS/2</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BSD/OS</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

*C = Enabled through Pthreads32 (POSIX interface towards WINAPI)

The POSIX [40] threading standard is supported by all the operating systems presented in Table 2, with the exception of Windows. However, a POSIX compliant library called Pthreads-win32 [35] is available for Windows that implements the standard POSIX interface as a wrapper against native Windows API threading functions.

### 7.7 Validation of spatially multiplexed video

The system output was verified based on pre-processed image pairs of interlaced and deinterlaced counterparts as described in chapter 5.3. The results obtained when the 3D video processing filter was fed with a deinterlaced input frame was equal to the pre-processed interlaced image which was proven by subtracting each RGB-component from the output frame by the corresponding RGB-component in the pre-processed interlaced image, which resulted in RGB triplets \( \{0, 0, 0\} \) for every pixel.
8 Conclusion

The proposed 3D video playback solution created within the scope of this thesis indicates that it is possible to build a working 3D video player relying completely upon open-source and cross-platform libraries. Compressed tiled and multi-view video formats up to resolutions of 1080p have been successfully verified to be displayed correctly at the intended frame rate. However, the results obtained from the performance measurements presented in chapter 7 indicates some interesting performance issues and technical obstacles which must be overcome in order to advance the efficiency of GPU-based real-time 3D video playback of multi-view video.

8.1 Evaluation of system throughput

According to the throughput results obtained for varying video file bit rates, illustrated in Figure 27, throughput is more or less linearly dependent on the video bit rate. Observing the average measured frame rate depending on the average pixel data rate, as illustrated in Figure 28, also indicates a linear relationship between the achieved throughput and increasing video resolution without any considerable anomalies between tiled and multi-stream formats.

The proposed 3D video playback solution has been verified as being able to display all video files except video L (see Figure 13) at the intended frame rate, regardless of whether the file is encoded as a tiled or multi-stream format, as illustrated in Figure 30. In the case of video L (see Figure 13), \( \Delta f_L = -3.066 < 0 \), indicates that the achieved throughput is insufficient, which is also confirmed by Figure 27 and Figure 28, where video L constitutes the rightmost circular symbol in the graph. The video L, has a variable bit rate with an average of 30.0 MBit/sec and fails to render at the native video frame rate of 30 frames/sec. Unfortunately, video L also has the highest video resolution which only adds complexity to the problem of determining why video L fails to reach the desired native video frame rate (throughput).
8.2 Multi-view and GPU performance

The results presented in Figure 27 and Figure 28 appear to indicate that hard drive data access time is a limiting factor for the inferior throughput of video L. However, according to Figure 31, illustrating thread load balance, it is evident that hard disc access is not a primary bottleneck for 3D video playback. At least this was not the case for the system used for the experiments, since the media container demultiplexer thread constitutes only a minor part of the total CPU-time spent on processing video. If hard disc access had been a bottleneck the processing time for media container access demultiplexing would have consumed the majority of the CPU-time waiting for data. It is also clear that the thread processing synchronization, filtering and rendering to the display is dominant and hence is the primary throughput bottleneck for video C, D as well as L.

If application profiling had been applied to a lower and more detailed level than on the thread level during the experiments, it would have been possible to further investigate which particular processing stages on the video synchronization, filtering and rendering thread actually constitutes the primary bottleneck. On the other hand, video synchronization is merely an operation of integer arithmetic on a small scale and rendering in this context only involves drawing a ready-to-display texture to the screen. Neither of these operations are considerably computationally intense, even though the rendering process is related to video resolution. Hence, the filtering processing step, i.e. the GPU, is currently most likely to be the primary bottleneck in the 3D video playback system. The reason for the inferior throughput of video L is probably a function of video resolution, limited by pixel shader performance rather than pixel data transfers on the system bus.

8.2 Multi-view and GPU performance

The GPU cache is designed to accelerate texture filtering and hence only requires to be as large as the texture filtering kernel, typically only a few texels. GPU cache formats are usually also designed for locality in two-dimensions. Considering the GPU cache format and a pixel mapping function similar to that presented in equation 5 for a lenticular display, when increasing the number of views $N$ for multi-view video, this would case the number of cache misses to increase with $N$. This is a result of the manner in which disparate view RGB sub-components are
aligned in a lenticular display. Hence, the texture lookups in the display pixel shader that perform spatial multiplexing will converge from high cache utilization for only a single view to a random memory access pattern when \( N \) is so high that no texels within the multi-view patch (see chapter 2.3.2) are common and fit into the texture cache.

Observing the average frame rate for a varying number of views \( N \) in Figure 29, the degradation of throughput appears to converge to a floor level when \( N \) reaches 24. Hence, for values of \( N \) higher than 24 this implies that random memory access patterns and the time for each texture lookup would equal the video memory access time since every texture lookup would result in a cache-miss.

The second order polynomial best-fit curve in Figure 29 appears to match the measured values almost too perfectly even though the data set is very limited. The reason for this perfect fit is most likely to be derived from the GPU cache format. Since the GPU cache is a square matrix, degradation is exponential. This would imply that the size of the square shaped texture cache of the video card is directly related to the exponential degradation of the GPU throughput that is currently limited by pixel shader performance. The minimum or floor throughput is however determined by the video memory access time for the particular video card.

However, no hardware specification is publicly available from NVIDIA to confirm any actual cache memory size for the NVidia Quadro FX 580 used for the experiments in order to confirm this hypothesis or make it possible to prove any theories presented in this conclusion mathematically.

### 8.3 Evaluation of load balance and hardware utilization

In Figure 31 the average processing time per frame is presented for the four threads processing video, presented for video C, D as well as L. As illustrated very clearly in the graph and as previously stated in section 0, the GPU is the primary bottleneck. From Figure 31 it is also evident that video decoding is the most CPU-time consuming task, whilst the media container demultiplexing takes only a small fraction of the total processing time per frame. The system appears to be very imbalanced
between threads, at least in the results obtained from the measurements for the four core processor machine used for the experimental evaluation. This low CPU efficiency is a direct result of the implementation of the static threads in the video processing pipeline as described in chapter 6.3.

In order to evaluate load balance it was decided to measure execution time for each thread on a per-frame basis, as stated in chapter 5.1. In a multithreaded environment this creates uncertainty regarding the actual effective CPU-time, since waiting times due to blocking on mutexes and conditional variables are included in the measurements. In order to increase the accuracy of future measurements a designated profiler mechanism should be used to measure the waiting time of each thread. Using such an approach, it would be possible to determine how much time is spent on actual computations and how much of the time the thread is blocked or is waiting for an asynchronous operation to complete. However, the load balance results indicate that hardware utilization is low and may potentially be improved with a more sophisticated design.

8.4 Evaluation of cross-platform interoperability

The proposed solution is built on top of several hardware and platform abstraction layers for hardware accelerated graphics, multithreading and window management. All third-party libraries are also open-source and hence support modification of the source code. As presented in Table 2, the requirements for cross-platform-interoperability are fulfilled as all the libraries used (FFmpeg, OpenGL, POSIX threads and GLUT) within the proposed solution may be compiled on at least Windows, Mac OS and Linux for x86, SPARC and PowerPC architectures. In reality some modifications to the current code might be required in order to compile the source code since the software has not yet been tested or verified on any operating system other than Windows Vista and Windows XP. Nor has it been verified for any other hardware than the system described in Appendix A, but this should merely be a question of minor modifications to the source code of the prototype implementation. Hence, the proposed solution is considered to support multiple platforms and hardware configurations.
8.5 **Evaluation of spatial multiplexing correctness**

The proposed 3D video playback solution has been verified to interlace video correctly by using interlaced and deinterlaced image pairs supplied by Mid Sweden University to compare system input and output as described in chapter 5.3. The system output has been verified to be correct objectively by sub-pixel arithmetic comparison with expected RGB-components and subjectively by observing 3D video content on the LG 42’ 3D display.

8.6 **Future work and recommendations**

The proposed solution has proven that it is possible to create a working 3D video player built on top of cross-platform, open-source libraries and frameworks, still providing sufficient throughput and configuration flexibility. However the solution proposed in this work is merely a novel video player application that might be improved in several ways.

The current solution has been optimized by loop unrolling for pixel copying routines which indicated an approximate speedup of 3 for copying routines. Since the software prototype has not yet been exposed to code review or application profiling, there might be other potential code optimizations. Using a profiler tool on a method basis to identify current bottlenecks is recommended in order to increase performance. Data locality should also be evaluated and improved in order increase CPU cache utilization.

Load balance in the proposed solution is very poor and is implemented statically so it will not scale well with hardware. The four threads are implemented logically according to typical stages in a video processing pipeline in order to even out throughput for each processing stage by using thread-safe queues as intermediate buffers. Load balance in the current solution can be improved by two strategies:

- Improved static parallelism – Parallelism may be increased by extending the producer/consumer pattern by decomposing the processing tasks into sub-tasks executed on an increased amount of threads. For example, this could be performed by decoding video streams in parallel for multi-stream media container files. However, such a solution will not scale with hardware and will
introduce additional context-switching overhead or result in low efficiency on systems where the number of processors does not match the number of threads.

- Introducing scalable parallelism – Performance and load balance may be increased by introducing scalable multithreading concepts by exploiting thread pooling techniques in the current solution. By using a thread pool with $M$ worker threads on a system with $M$ processors and dividing algorithms into coarse grained tasks, the hardware efficiency might be improved, while still retaining overhead at acceptable levels. Concepts such as work-stealing might be considered to create efficient task queuing and scheduling.

It is immaterial whether static or scalable parallelism is introduced to the current design, and it is also highly interesting to introduce buffering of spatially multiplexed video frames in video memory (texture memory). In this manner the “smoothness” of throughput might be improved, but will introduce overhead by fetching and unloading textures from the texture units in the GPU. However, such a solution would make it possible to run synchronization and filtering on a separate thread and allow the vertical synchronization call-back to only constitute a simple texture fetch and rendering call. Such a solution would also introduce synchronization overhead, since semaphores or mutexes would be required to provide thread-safety for OpenGL calls.

It is also vital to remember that the results obtained and presented within the scope of this work have been evaluated for only one system. New chipset and GPU architectures with different cache architectures might provide completely different results. Since GPU computational performance is doubled every six months, beating Moore’s law, it is recommended to also evaluate the proposed system for top-end hardware.
References


3D Video Playback - A modular cross-platform GPU-based approach for flexible multi-view 3D video rendering
Håkan Andersson

References

2010-11-25


http://www.business-sites.philips.com/3dsolutions/about/index.html
Retrieved: 2006-08-01.


Retrieved: 2009-12-01.

[18] Apple Developer Connection, “QuickTime”
3D Video Playback - A modular cross-platform GPU-based approach for flexible multi-view 3D video rendering

Håkan Andersson

[19] Microsoft, “Introduction to DirectShow (Windows)”

http://www.mplayerhq.hu/

[21] VideoLAN Project, “VLC media player”.
http://www.videolan.org/vlc/

[22] MPEG Video Group,”Overview of the MPEG-4 Standard”.
http://www.chiariglione.org/mpeg/standards/mpeg-4/mpeg-4.htm


http://ffdshow-tryout.sourceforge.net/

http://www.free-codecs.com/download/FFDshow.htm


http://en.wikipedia.org/wiki/Hard_disk_drive
3D Video Playback - A modular cross-platform GPU-based approach for flexible multi-view 3D video rendering

Håkan Andersson

References

2010-11-25

[29] Seagate. “Data Sheet Barracuda 7200.12”


http://en.wikipedia.org/wiki/Video_card

[32] NVIDIA, “GPU Gems 2 – Chapter 32”


[34] Wikipedia. “Shader”
http://en.wikipedia.org/wiki/Shader

[35] PThreadsWin32, “POSIX Threads (pthreads) for Win32”
http://sourceware.org/pthreads-win32/.

http://en.wikipedia.org/wiki/YUV

[37] OpenGL. “GLUT – The OpenGL Utility Toolkit”
http://www.opengl.org/resources/libraries/glut/
3D Video Playback - A modular cross-platform GPU-based approach for flexible multi-view 3D video rendering
Håkan Andersson


Appendix A: System specification for experiments

Hardware specification

ACPI Multiprocessor PC.
Intel ICH8R/ICH9R SATA RAID Controller.
Intel ICH10R INterface controller.
Intel 5520/550/X58 Hubs.
Intel ICH10 PCI express.
Intel Xeon E5520 @ 2.27 GHz (2266 MHz) 2.27 GHz, 2.50 (2047 MB) GB RAM.
Intel 82801 PCI Bridge - 244E.
Seagate Barracuda 7200.12 SATA 3GB/s 250 GB Hard drive 8 MB Cache (ST3250318AS).
NVidia Quadro FX 580 / PCI / SSE2 / CUDA 1.1GHz 4 processors. 511 MB.
Max uniforms: vertex 4096, pixel 2048, geometry 2048.
Max varying floats: 60. Max vertex attrs: 16.

Specification of software used throughout this project

Windows XP Pro 2002 SP3.
Bloodshed Dev C++ 4.9.9.2.
Mingw
Microsoft Visual Studio 2008 Pro, Version 9.0.21022.8
Visumotion 1.6.3.0.
Stereoscopic Player 1.4.6.
OpenGL 3.1.
GLSL 1.40 NVIDIA Cg Compiler.
Appendix B: Pixel buffer object (PBO) performance

Using a circular list of OpenGL pixel buffer objects (PBOs), the system bandwidth and GPU bandwidth utilization can be increased using asynchronous DMA transfers to decrease CPU time for texture transfers from system memory to video memory as illustrated in Figure 32.

![Figure 32: Average texture copying time from system memory to video memory for varying number of pixel buffer objects (PBOs) used.](image)

Based on the observed measured time presented in Figure 32, a speedup of 3.42 according to equation B.6 is obtained when four PBOs are used.

\[
Speedup = \frac{CopyTime(0)}{CopyTime(4)} = \frac{2.364918 \text{ ms}}{8.098820 \text{ ms}} \approx 3.42
\]  

(B.6)