Optimization of video encoding for cloud gaming

Jan Anders Bremer
Master’s Thesis Autumn 2014
Optimization of video encoding for cloud gaming

Jan Anders Bremer

1st August 2014
Abstract

This thesis contains preliminary work on optimizing video encoding of 3D environments by using information available in the rendering pipeline.

It suggests that the video encoding step known as motion estimation can be optimized or bypassed. This has been done using information which is available through keeping the state used to render the previous state of a system. This state was then used to calculate screen-space differences between the current and previous state.

Two similar methods have been proposed and tested in this thesis. One method skips motion estimation, the other method provides additional information to the motion estimation step.

This thesis also suggests several other methods that are partially based on the methods tested as potential future work.
CONTENTS

2.6.2 Accelerated Video Encoding Using Render Context Information ........................................ 44
2.6.3 Using SIMD ........................................ 45

3 Proposed method ........................................ 49
3.1 Idea ........................................ 49
  3.1.1 Use of a game state ........................................ 49
  3.1.2 Movement within a simulation rendered within OpenGL ........................................ 50
  3.1.3 Modern GPU technology ........................................ 55

4 Implementation ........................................ 57
  4.1 The system ........................................ 57
    4.1.1 Choice of components for testing ........................................ 57
    4.1.2 Changes in the components ........................................ 58
  4.2 Demo scenes ........................................ 59
    4.2.1 OpenGL-tutorials.org tutorial-based demo scene ........................................ 59
    4.2.2 ValeDecem ........................................ 59
  4.3 Implementation details ........................................ 60
    4.3.1 Standards ........................................ 60
    4.3.2 Dumping pixel data ........................................ 61
    4.3.3 Dumping motion vectors ........................................ 63
    4.3.4 Modifying the encoder ........................................ 66
  4.4 Issues ........................................ 67
    4.4.1 Motion vector value limitations ........................................ 68
    4.4.2 Transparency ........................................ 68
    4.4.3 Small objects ........................................ 69
    4.4.4 Advanced effects ........................................ 69
    4.4.5 Viewport distortion, rotation of objects, objects approaching the camera ........................................ 69
    4.4.6 HUD ........................................ 70
    4.4.7 Post-processing effects and shadows ........................................ 71
    4.4.8 Anti-cheat ........................................ 72
    4.4.9 Artifacts ........................................ 72
  4.5 Source code ........................................ 73
    4.5.1 Prototype components ........................................ 73
    4.5.2 Modified code ........................................ 74

5 Findings ........................................ 75
  5.1 Test metrics and goals ........................................ 75
  5.2 Initial inf5063-c63-prototype ........................................ 76
    5.2.1 Status of the prototype ........................................ 76
    5.2.2 Performance improvement ........................................ 76
    5.2.3 Output quality ........................................ 76
    5.2.4 File size ........................................ 77
  5.3 First x264-prototype ........................................ 78
    5.3.1 Status of the prototype ........................................ 78
    5.3.2 Performance improvement ........................................ 78
6 Conclusions

6.1 Overall interpretation of the results 81
6.2 Further work 82
   6.2.1 Macroblock size prediction 82
   6.2.2 Vertex tracking through mesh and vertex IDs 84
   6.2.3 An idea to increase responsiveness when playing games in the cloud 84
   6.2.4 3D objects encoding 85
   6.2.5 Other suggestions 85
6.3 Other observations 85
   6.3.1 OpenGL demo scene availability 86
   6.3.2 OpenGL versions in open source software 86
   6.3.3 YUV 4:2:0 vs Y’CbCr 4:2:0 86
   6.3.4 Development notes 87
6.4 Testing environment 87
   6.4.1 Hardware 88
   6.4.2 Software 88
# List of Figures

2.1 The Y’CbCr colour space for given a constant Y and variable Cb (horizontal axis) and Cr (vertical axis) for (from left to right) $Y=0$, $Y=0.5$ and $Y=1$ .................................................. 7  
2.2 A 400x300 jpeg-image showing a few seagulls on the sea with forests and boats in the distance. ................................. 8  
2.3 A 400x300 jpeg-image with easily noticeable artifacts. ....... 9  
2.4 Figure 2.3 without lossy compression, saved instead as png. 9  
2.5 Figure 2.2 compressed to 1/3rd of the size, showing local cosine-waves. ................................................................. 10  
2.6 Figure showing the search range for the Y-channel in the inf5063-c63-encoder. .......................................................... 13  
2.7 Figure showing the search range for the Chroma-channels in the inf5063-c63-encoder. ...................................................... 14  
2.8 Incidental vs. actual movement. ........................................ 15  
2.9 Sample matrix. .............................................................. 16  
2.10 Sample matrix multiplications. ........................................ 16  
2.11 The $4 \times 4$ identity-matrix, also known as $I_4$. ........ 17  
2.12 $I_4 \times v$. ...................................................................... 17  
2.13 The $4 \times 4$ general translation matrix. ............................. 17  
2.14 $T \times v$. ...................................................................... 17  
2.15 $T \times v'$. ...................................................................... 17  
2.16 The $4 \times 4$ general scaling matrix. ................................. 18  
2.17 $S \times v$. ...................................................................... 18  
2.18 Matrix multiplication, $S \times T \times v$. ............................... 18  
2.19 Matrix multiplication, $T \times S \times v$. ............................... 18  
2.20 Figure showing how position-vectors are processed using matrices, taken from opengl tutorials.net [2014]. .............. 19  
2.21 Screenshot of Battlefield 4, a game which relies on shaders to increase graphical fidelity. ............................................. 22  
2.22 Screenshot of SimpleGL – default rendering code. .......... 27  
2.23 Screenshot of SimpleGL – alternate rendering code. ........ 28  
2.24 Screenshot from a computer situated in Norway pinging a router over the local wireless network, then pinging a remote server in Germany. ................................................ 32  
2.25 Screenshot from a computer situated Norway running a traceroute against a remote host. .............................................. 33  
2.26 Screenshot of Stepmania 5’s input calibration screen. .... 38
2.27 Figure showing how delay affects performance in games, taken from Claypool and Claypool [2006].  

3.1 Two slightly different screenshots from Insurgency.  

3.2 The x and y-components of the rendered homogeneous space in OpenGL.  

3.3 Illustration of an object’s position in a 1024x768 render.  

3.4 Illustration of movement in a 1024x768 render.  

3.5 The space spanned out by the coordinates of the Y’-channel of an input frame with resolution 1024x768 with a sample point in the middle of the frame and a sample point at 511 x-units and 383 y-units.  

3.6 A 64x64 unit excerpt showing which pixel-information is used as to determine the motion vectors in each macroblock.  

4.1 Screenshot of tutorial-demo.  

4.2 Screenshot of ValeDecem.  

4.3 Screenshot of Half-Life 2 showing the insides of a moving train with windows.  

4.4 Screenshot of Half-Life 2 showing ripples and effects on the water surface which prove problematic to the proposed method.  

4.5 Screenshot of Dark Souls showing a dialogue box under the centre of the screen telling what button to press to rest at the nearby bonfire.  

4.6 Screenshot of Dark Souls showing the character interaction menu in the upper right corner.  

4.7 Screenshot of Antichamber, a game which may exhibit the same problems as many animated cartoons when being recorded and compressed using a video encoder due to high contrast.  

5.1 Comparison of noise with different search ranges after a movement towards the upper left corner. Samples taken from tutorialdemo.  

6.1 Macroblock scaling.  

6.2 Macroblock geometry prediction.  

6.3 For Y = 0.5 and Y’ = 0.5, left: Y’CbCr, right: YUV.
List of Tables

2.1 Definitions ................................................................. 6
2.2 A short overview of cloud gaming compared to traditional gaming running on a local machine in terms of requirements and benefits for a graphically demanding single-player game. 40

5.1 Sizes of files used for testing. ......................................... 79
5.2 Table showing file sizes and run-times for encoders given different parameters and input files. ................................. 80
Preface

This thesis should be regarded as a preliminary work to create specialized and better video recording and streaming software for video games and other applications. It proposes and tests two methods. These two methods are in common referred to as the “proposed method”. This thesis also suggests several more methods to be tested.

This thesis was written by me during 2013 and 2014 at Simula Research Laboratory cooperating with the Institute of Informatics at the University of Oslo. The thesis advisors were Kjetil Raaen(raakje@nith.no) and Carsten Griwodz(griff@ifi.uio.no).

The method, results and suggested future work will also be published in the form of a paper at www.simula.no before the 22nd of August 2014.

If you wish to contact me, I can be reached at admin@bo-it.org.

Thanks

There are several other people who have contributed to this thesis and my studies whom I would like to thank.

I would like to thank my parents who have supported me throughout my studies. My father also read through this thesis twice.

I would like to thank my thesis advisors for making this thesis possible and helping me along the way.

I would like to thank Kristoffer Kolbjørnsæter Bamrud for reading through an early draft of this thesis and providing feedback.

I would like to thank Øystein Dale and Kjell Wilhelmsen for reading through the near-finished draft, confirming that the document is understandable. Øystein also told me about Tikz, a package for \LaTeX which was used in the creation of several of the figures in this thesis.

Kriss-Anina Nyheim has my gratitude for proof-reading the final draft.

In addition, I would like to thank the developers of the x264-encoder, the
developers inf5063-c63 encoder, the maker of opengl-tutorials.org and the
makers of the ValeDecem OpenGL demo for providing source code which
made my tests possible. I would also like to thank the developers of
OpenGL and other libraries such as Asset Importer, glm, glew and glfw.

Finally, I would like to thank the creators of L\TeX, TeXworks and the
creators of the packages (tikz, longtable and many other) used in creating
this document.
Chapter 1

Introduction

This thesis proposes a method for optimizing video encoding for cloud gaming, video streaming and other applications through the use of information available when rendering a game in 3D.

It also discusses advantages, issues and details related to this method.

1.1 Description of environment

Recently, several new possibilities have opened up through the major improvements of not only computational power, but also Internet-speeds. We are moving towards a situation where running not only online services such as websites or game-servers, but normal user-applications such as games, text editors or even entire operating systems on a remote machine is a possibility. There are several actors in this field, such as OnLive (Onlive [2014]) or StreamMyGame (StreamMyGame [2014]).

In September 2013, Valve, the company behind the world’s largest digital distribution platform for PC-games – Steam, announced a Linux-based operating system called SteamOS (Valve-1 [2014]) with the possibility to run games on other machines on your local network and showing the output on your local machine. The streaming-possibilities and the operating system are now in working condition (Valve-2 [2014] and Valve-3 [2014]).

While the client-server architecture already exists (see X.org Foundation [2012]), running a latency-critical application where the server is not hosted on the local machine creates a new scenario with new issues.

1.2 Description of challenges within cloud gaming

There are several challenges which must be overcome in order to deploy latency-critical applications in a client-server scheme with a remote server and a local client. Unlike streaming movies or music, streaming most
1.2. Description of challenges within cloud gaming

Chapter 1. Introduction

games requires low response-times between user-input and an action happening on the screen because games are interactive. The required response times vary between different types of games – for instance: you don’t need to be able to act within a second if the game gives you infinite time to ponder your actions. However, delay in any interaction can ruin the experience of playing a game.

Multiplayer games also require that clients are more or less in synchronisation with each other and that what each client sees is as close as possible to the current situation in the game.

Game developers have been working on reducing perceived latency in multi-player games for over a decade. However: the scenario with a remote machine running the game client itself is new and therefore introduces new problems. Currently, the greatest limitations and problems are the following: network-bandwidth, network-delay and processing-delay. The two which are relevant to this thesis are network-bandwidth and processing-delay.

Due to our current Internet-bandwidths, video-encoding is necessary if you wish to view animated content in real time without frame drops. A game running at 60 frames per second, in full HD (1920 by 1080 pixels) with 8 bits per colour channel generates a staggering amount of raw image-data – approximately 356 MB per second. The current top-grade consumer-available ISP-solutions provide at most 125 MB/s download speed (Tele2 [2014] in Sweden), while the average consumer probably has around 2.5 MB/s download speed (Netindex [2014]). This problem will only get worse as screen-resolutions are increasing. If we want the content being streamed and our screens to have the same resolution, we will most likely not be able to do so without video-encoding for a long time and streaming uncompressed video will most likely not be considered viable. The idea is that the time it takes to compress and send a video or a video-stream is shorter than sending the uncompressed version.

While video-encoding drastically cuts down on the amount of data being sent – often more than a thousand-fold, it also contains several heavy operations and may also introduce artifacts\(^1\) in the video upon decoding.

The heavy operations involved in video-encoding means more processing-delay and cost and any effort to reduce this delay will therefore enhance the user experience unless other aspects of the experience degrade dramatically. For instance, developers of certain encoders spend a lot of time to optimize performance per platform (see the x86, ppc, opencl and arm-folders in the common folder in VLC [2014]).

---

\(^1\)visual glitches such as discourting or flickering effects
1.3 Goals and suggested improvement

The proposed method of this thesis suggests to use information within the applications or games being streamed to help the video encoder determine the motion in the frame. The suggestion is to augment or remove the motion estimation step. None of the other steps the video encoder performs have been touched.

This has been done using the capabilities of a system’s graphics card which in our case was also rendering the game or application.

The primary goal is to provide a proof of concept – that it is possible to implement the proposed method.

The secondary goals for the suggested improvement are that it should speed up the encoding-process while not interfering in such a way that it would cause a loss of quality of experience for the end-user. This includes causing additional delays, visual glitches and other issues.

1.4 Scope

This thesis describes possible optimizations of video encoding, specifically how one can use information calculated in a 3D rendering pipeline to simplify the step known as "motion estimation" in video-encoders when encoding the output from a game or a generic user application.

This thesis discusses the limitations, benefits and challenges in several different approaches of using and providing this information.

The purpose of this thesis is not to create a perfect solution for optimizing video-encoding of output from a 3D-environment, but rather to show that it is theoretically possible to not only use a GPU for encoding, but also to use information from a game or program to help the encoding-process.

The proposed method could give hardware manufacturers a new possibility to simplify their hardware and software developers the possibility to reduce the footprint of screen-recording programs or recorders integrated in games or other applications.

The proposed method has been implemented with several prototype components provided with full source code for further development, analysis and testing.

Even if the tests show that with the current encoders it is hard or not viable to improve them in this way, the proposed method and similar methods could be considered when designing new encoders, perhaps even an encoder made to encode the output from a specific game or application.
1.5  Approach

The approach in regards to testing the proposed method was to first create a proof of concept and then to create additional more advanced prototypes.

While creating the proof of concept, simplicity was a key factor. Therefore a simple demo scene based on an OpenGL-tutorial was created and inf5063-c63 – a simple encoder, was modified.

The advanced prototypes consisted of the x264-encoder and a demo scene called ValeDecem. The purpose of these prototypes was to see if the method is usable in complex scenarios.

Please note that both demo scenes are compatible with both video encoders.
Chapter 2

Background

This thesis combines theory from several different fields within computer science. It uses 3D graphics rendering – specifically the library known as OpenGL, video encoders and cloud computing. A basic understanding of each of these subjects is required to understand the proposed method and its scope.

2.1 Definitions

A star(*) denotes that this is not a common definition.

<table>
<thead>
<tr>
<th>Word</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>API</td>
<td>Application programming interface, used to define how programs should communicate.</td>
</tr>
<tr>
<td>Artifacts</td>
<td>Unintended side-effects or glitches in sound, video and 3D-technology.</td>
</tr>
<tr>
<td>CPU</td>
<td>Central processing Unit, executes the program(s) and operating system (if present) in a machine.</td>
</tr>
<tr>
<td>Frame</td>
<td>Within video encoders/decoders: unit containing information about an image to be encoded/decoded.</td>
</tr>
<tr>
<td></td>
<td>Within OpenGL and other rendering systems: the buffer onto which a scene is rendered.</td>
</tr>
<tr>
<td>Frame-space motion*</td>
<td>Motion as seen from the game’s camera.</td>
</tr>
<tr>
<td>GPU</td>
<td>Graphical Processing Unit, specialized hardware used to quickly draw 3D images to a buffer and display this buffer on a screen.</td>
</tr>
<tr>
<td>High-Dynamic Range</td>
<td>Used in games to adjust contrast and brightness of a scene, emulating an actual eye’s light-adjusting function. However, in certain cases this functionality can be set to hypersensitive for artistic effects.</td>
</tr>
<tr>
<td>Hit-scan [weapon]</td>
<td>Weapons in a game which hit instantaneously.</td>
</tr>
<tr>
<td>Input-To-output delay*</td>
<td>The time it takes from user-input such as a key press on a keyboard, mouse movement or controller interaction until the change is visible on the screen.</td>
</tr>
</tbody>
</table>
2.2 Video encoders

The purpose of a video encoder is to read files or streams containing video (and audio) and apply algorithms in such a way that the space requirement of the video is reduced. If we are to transfer high-resolution video from one machine to another, these technologies are needed because transmitting raw video in real time takes up more bandwidth than we in most cases have. As mentioned in the introduction, 60 frames per second in full HD is 356 MB of raw data per second.

Before describing the overall steps followed by an encoder and the encoders used in this thesis, I have described the Y’CbCr colour space and file format as it is a prerequisite of understanding certain parts of an encoder.

### 2.2.1 Y’CbCr 4:2:0

Y’CbCr is a way of encoding colour information. Y’ denotes the luma (brightness) and Cb and Cr denote the blue-difference and red-difference.
The format known as Y’CbCr 4:2:0, contains frame-by-frame Y-channel-data, Cb-channel-data and Cr-channel-data. The Cb and Cr-channel have a quarter of the resolution of the Y’-channel.

Figure 2.1: The Y’CbCr colour space for given a constant Y and variable Cb (horizontal axis) and Cr (vertical axis) for (from left to right) Y=0, Y=0.5 and Y=1

The reason for lower resolution in the Cb and Cr-channels is because the human eye possesses worse spatial sensitivity to colour compared to brightness. This conserves bandwidth by discarding 75% of the data (per channel) while losing little to no perceived visual fidelity. For more details see CDE [2014].

2.2.2 Common assumptions within a video encoder

The most common assumption within a video encoder is that your input data is natural-looking, that is: the input data is some sort of real-life recorded video or similar. To understand how these assumptions affect video quality, a description of jpeg is needed. For a complete overview of jpeg, see Wallace [1991]. Wallace [1991] details the use of a discrete cosine transform (DCT) and more to compress image-data. The relevant parts to video encoding are summarized below.

Image-data stored using jpeg is compressed at the expense of adding artifacts which easily hide within natural-looking images, but not as easily in schematics, drawings or images with high contrast and clear borders. For instance, consider figures 2.2 and 2.3. Both files were saved as jpeg-files of approximately the same size (14 KB) and identical resolution (400x300), however: artifacts are easily noticeable in figure 2.3 – there is noise almost everywhere. The circle on the right side in that figure was drawn using pure red (255, 0, 0), but upon inspection with a colour picker, it can be found that it is closer to dark red (180, 20, 20) with variations. Figure 2.4 was saved as a png and is provided for reference. It is in fact smaller than the jpeg – it takes up approximately 5 KB.

If the compression is increased, the discrete cosine transform breaks down
and yielding results such as figure 2.5. This figure also shows how jpeg-images are constructed. They consist of blocks which approximate the data by describing what cosine-waves applied on top of each other in small regions resemble the area the most – this is the DCT. Depending on the quality settings for the image, different levels of quantization are available. That is, DCT uses several different cosine waves multiplied by certain float values to approximate an area. These float values are divided by a constant from a “quantization”-table\(^1\) and then stored as integers, thus discarding some data. When reading these stored files, the values read are multiplied by the same constants from the same quantization table, thus approximating the original values. This approximation is what causes the artifacts in the compression.

Since many video encoders make the exact same assumption and use DCT, heavy artifacting may be observed in footage which is not “natural-looking”. For instance: compressing animated cartoons has proved problematic using lossy video encoders.

---

\(^1\)The values in the quantization-table depend on the quality-settings
Figure 2.3: A 400x300 jpeg-image with easily noticeable artifacts.

Figure 2.4: figure 2.3 without lossy compression, saved instead as png.
2.2. Video encoders

Chapter 2. Background

Figure 2.5: Figure 2.2 compressed to 1/3rd of the size, showing local cosine-waves.

2. For every frame read: divide every frame into macroblocks. A macroblock is a rectangle \(^2\) of image data, typically 8x8, 16x16, 32x32 or 64x64 pixels large. These macroblocks are used to find similarities between frames.

3. Then, for every frame execute one of the following:

- Compress and store all the data in the frame, macroblock by macroblock. Frames stored this way are called “reference frames”. They do not depend on any other frame and can be read without reading previous frames. Generally, reference frames are used by encoders on start of input, sometimes when there are extreme changes in a scene or a change of scene occurs and with regular intervals.

- If a reference frame is not used, the encoder performs “motion estimation” and “motion compensation”, thus storing compressed macroblocks. These macroblocks reference information in other frames and store the differences between the areas referenced in these frames and the macroblocks in the current frame. This process is detailed in section 2.2.6 using one of the encoders used in this thesis as an example.

Some steps are omitted as they are not relevant in this case.
For a complete overview, see Wang et al. [2001].

\(^2\) in most encoders a square
2.2.4 inf5063-c63

The inf5063-c63-encoder is a basic encoder used for educational purposes at the University of Oslo (see Olsen [2013]) and is not used elsewhere. It is based on a motion jpeg encoder.

The reason for choosing this encoder is that it serves as a proof of concept – the implementing the suggested method is possible.

It uses 8x8 macroblocks, it uses motion estimation that executes an exhaustive search within a 40x40 (Y-channel)/24x24 (Cb and Cr-channels) area around a macroblock. The encoder features DCT quantization. It is not optimized, it does not use any platform-specific instructions and can be executed on any machine capable of compiling C-code with little to no modification.

The encoder outputs c63-files, its internal format.

2.2.5 x264

The x264-encoder associated with the VLC media player development is a fairly advanced encoder (h264 wiki [2014]) which outputs files in the H.264/MPEG-4 format, a standardized format by both ITU.T and ISO (Wiegand et al. [2003]) used in various projects, including YouTube, Vimeo and many other programs and websites. H.264 specifies a series of different profiles and settings, which are intended for different uses – such as streaming to a phone, TV or PC.

The x264-encoder can use different sizes for the macro blocks, it has subpel-prediction\(^3\) and different types of searches available. The code-base includes files for different architectures which are used based on system information. This means that the program uses platform-specific instructions and extensions, making it run faster.

Due to this, the x264-encoder is hard to beat performance-wise (see compression.ru [2010]). It is a well-made encoder with several years of development and testing behind it, and it is therefore an ideal encoder to test against. If the proposed method performs similarly as the original x264 implementation, then this might be a sign that the method should be looked into in regards to further development of existing encoders and codecs or in the development of new ones.

\(^3\)Use of virtual pixels which are (possibly weighted) averages of 4 and 4 pixels for motion search and estimation

11
2.2. Video encoders

2.2.6 Motion estimation and compensation – detailed

Most video-encoders and the ones used in this thesis split every single frame of the video into macroblocks. These are units of typically 4x4, 8x8 or 16x16 or even larger blocks of pixels laid out in a grid-pattern. The inf5063-c63-encoder supports only 8x8 pixels per macroblock, while the x264-encoder supports several sizes, including 8x8.

The reason for using motion vectors and differences between (reference) frames is compression. It is more space-efficient to describe differences between a macroblock and the previous frame if for instance Huffman coding (see Huffman [1952]) is used on the values.

For instance, the inf5063-c63-encoder used in this thesis does the following:

1. Advance one frame

2. If there have been major changes in the scene, N frames have passed since the last reference frame or a scene change has occurred, the following happens:
   - The encoder encodes a reference frame using the raw data for the first frame, DCT and quantization is applied. A reference frame is a frame which can be read by a decoder without knowledge about what came before it as it includes all the information about the entire frame without any dependencies. Go to step 1.

3. For every macroblock in the current frame, execute a SAD against this macroblock and every 8x8 area within a certain range in the previous frame (this varies from encoder to encoder, for instance: the inf5063-c63 encoder uses 16 pixels out of the macro-block for the Y-channel and 8 pixels out of the macro-block for the Cb and Cr-channels). Find the lowest SAD and store the relative position of where it was found in the macroblock – this is the motion vector.

4. Find differences between every macroblock and the macroblock the motion vector points at as well as the motion vector itself. This is known as finding the residuals.

5. Apply a weighted discrete cosine transform to all macroblocks (which now contain the residuals) and store them channel-wise. This has been covered in section 2.2.2. Store the resulting data and the motion vectors.

6. If your encoder is not mathematically perfect, reconstruct the frame using the motion vectors and differences found in step 5 by applying

---

4 This also acts as a fail-safe in situations where some data is corrupted or lost, for instance: if you are sending someone a video in real-time (streaming) over the Internet over a lossy protocol and certain packets are lost, then video-stream will be discoloured or corrupted until the next reference frame is received.

5 Decoding an encoded frame does not yield the exact input frame.
the inverse DCT (iDCT) to the residuals. The new reconstructed frame is to be used instead of the previous frame for computing SADs for the next frame due to potential differences.

7. If there is a next frame, go to step 1, else: end.

When using coding-schemes such as Huffman coding, greater efficiency is achieved through making sure that certain values (for instance 0 and values around zero) appear more often than other (higher values) and making sure that these values get short symbols. This is why searching for good matches in the previous frame is an important aspect of a video encoder – lower differences between the macroblock being matched and the referenced area in the previous frame means more similar, low values which, which thus yield lower values after DCT and thus can be compressed efficiently.

\[ u = 1 \text{ unit in the encoder} = 1 \text{ pixel} \]

Red box: macro-block for which the encoder is trying to find a motion vector
Blue box: set of pixels tested
Green and yellow dashed boxes: the two first positions tested by the encoder.

Figure 2.6: Figure showing the search range for the Y-channel in the inf5063-c63-encoder.
2.2. Video encoders

One of the fundamental parts of a modern video encoder is finding motion vectors. Motion vectors describe where the encoder thinks a given block of pixels originated from. This may correspond to the actual motion perceived or it may be a completely different vector which coincidentally matches the block. See figure 2.8: if an object is moving across a surface along the solid line, while also rotating, the encoder might suddenly decide that a part of the background matches the object better, thus setting the motion vector to the dotted line.

The amount of calculations required to execute this operation varies from encoder to encoder, so the effect of the removal or augmentation of this step may vary performance-wise.

2.2.7 Motion vectors in video encoders

u = 1 unit in the encoder = 2 pixels
Red box: macro-block for which the encoder is trying to find a motion vector
Blue box: set of pixels tested
Green and yellow dashed boxes: the two first positions tested by the encoder.

Figure 2.7: Figure showing the search range for the Chroma-channels in the inf5063-c63-encoder.
Chapter 2. Background

2.3 3D rendering

This section describes 3D graphics theory and aspects of a commonly used 3D rendering library – OpenGL.

2.3.1 3D graphics – general

3D graphics rely on geometric algebra and thus it is important to have basic knowledge about this subject in order to understand certain aspects of this thesis. For more details, see Vince [2008]. For a general introduction to computer graphics, see Akenine-Möller et al. [2008].

Computer graphics relies heavily on the use of matrices to represent how geometry is drawn on our screen – that is: vectors describe position of geometry and other matrices describe how this geometry should be transformed. This includes positioning of our camera, effects such as skew and scaling. The base shape of geometry in most rendering systems is the triangle and multiple triangles are used to form more complex shapes such as squares or approximate a circle.

A matrix is a two dimensional ordered set of numbers of varying size. Matrices have certain defined operators, such as addition and multiplication. I have focused on the multiplication operator as it is of great importance to 3D graphics.

Please note that matrices operate with the indices on the form of \((y, x)\), \(y\) being the vertical offset or dimension and \(x\) being the horizontal offset or dimension. Offsets are calculated from the upper left corner. For instance: a 3x2 matrix is a matrix with three rows and two columns, such as figure

![Incidental vs. actual movement.](image)
2.3. 3D rendering  

Chapter 2. Background

2.9. $M_{31}$ thus refers to 5 in that matrix (highlighted in bold).

\[
M = \begin{pmatrix}
1 & 2 \\
3 & 4 \\
5 & 6
\end{pmatrix}
\]

Figure 2.9: Sample matrix.

The multiplication ($\times$) operator for matrices has the following properties:

- The output matrix has the same number of rows as the first matrix and the same number of column as the second matrix.

- It is only possible to multiply a matrix with another if the number of columns in the first matrix is equal to the number of rows in the second matrix.

- The multiplication operator is not associative ($A \times B \neq B \times A$).

- For the an output matrix $M$: column $x$, row $y$ (or $M_{yx}$) contains the sum of row $x$ in the first matrix multiplied element-wise with column $y$ in the second matrix.

\[
\begin{pmatrix}
a_1 & a_2 & a_3 \\
a_4 & a_5 & a_6
\end{pmatrix} \times \begin{pmatrix}
b_1 & b_2 \\
b_3 & b_4 \\
b_5 & b_6
\end{pmatrix} = \begin{pmatrix}
(a_1b_1 + a_2b_3 + a_3b_5 & a_1b_2 + a_2b_4 + a_3b_6) \\
(a_4b_1 + a_5b_3 + a_6b_5 & a_4b_2 + a_5b_4 + a_6b_6)
\end{pmatrix}
\]

\[
\begin{pmatrix}
b_1 & b_2 \\
b_3 & b_4 \\
b_5 & b_6
\end{pmatrix} \times \begin{pmatrix}
a_1 & a_2 & a_3 \\
a_4 & a_5 & a_6
\end{pmatrix} = \begin{pmatrix}
b_1a_1 + b_2a_4 & b_1a_2 + b_2a_5 & b_1a_3 + b_2a_6 \\
b_3a_1 + b_4a_4 & b_3a_2 + b_4a_5 & b_3a_3 + b_4a_6 \\
b_5a_1 + b_6a_4 & b_5a_2 + b_6a_5 & b_5a_3 + b_6a_6
\end{pmatrix}
\]

Figure 2.10: Sample matrix multiplications.

There are also special cases of matrices called vectors. There are two different cases of vector, $1 \times m$ matrices called row vectors and $n \times 1$ matrices called column vectors. The matrix multiplication rules also apply to these cases.

To illustrate how these matrices interact, some typical operations are described below and their matrices and their effect on a sample point located in $(2,2,2)$. This point is represented as a 4-component column vector: $v = (2,2,2,1)$. These components are named (in order): $x$, $y$, $z$, $w$. The $w$-component is used for scaling of certain operations, as we shall see in the following examples using a point at the same position, but with the last component set to 2, $v' = (2,2,2,2)$. Because a 4-component vector is used, a $4 \times 4$ matrix is also needed, such as the identity matrix seen in figure 2.11. The reason for needing $4 \times 4$ matrices and 4-component vectors is that this allows effects such as translation.
Figure 2.11: The $4 \times 4$ identity-matrix, also known as $I_4$.

The result of multiplying $I_4$ by $v$ can be seen in figure 2.12. Multiplying an identity matrix with a vector yields the vector as the result.

\[
\begin{pmatrix}
1 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 \\
0 & 0 & 1 & 0 \\
0 & 0 & 0 & 1
\end{pmatrix}
\begin{pmatrix}
2 \\
2 \\
2 \\
1
\end{pmatrix}
= 
\begin{pmatrix}
2 \\
2 \\
2 \\
1
\end{pmatrix}
\]

Figure 2.12: $I_4 \times v$.

Translation is the simplest of effects – every point is moved by $(T_x, T_y, T_z)$ if multiplied by the matrix in figure 2.13.

\[
T = 
\begin{pmatrix}
1 & 0 & 0 & T_x \\
0 & 1 & 0 & T_y \\
0 & 0 & 1 & T_z \\
0 & 0 & 0 & 1
\end{pmatrix}
\]

Figure 2.13: The $4 \times 4$ general translation matrix.

The result of multiplying $T_4$ with $v$ can be seen in figure 2.14. The result of changing the aforementioned scaling, can be seen in figure 2.15 (which doubles the amount our point is translated).

\[
\begin{pmatrix}
1 & 0 & 0 & T_x \\
0 & 1 & 0 & T_y \\
0 & 0 & 1 & T_z \\
0 & 0 & 0 & 1
\end{pmatrix}
\begin{pmatrix}
2 \\
2 \\
2 \\
1
\end{pmatrix}
= 
\begin{pmatrix}
2 + T_x \\
2 + T_y \\
2 + T_z \\
1
\end{pmatrix}
\]

Figure 2.14: $T \times v$.

\[
\begin{pmatrix}
1 & 0 & 0 & T_x \\
0 & 1 & 0 & T_y \\
0 & 0 & 1 & T_z \\
0 & 0 & 0 & 1
\end{pmatrix}
\begin{pmatrix}
2 \\
2 \\
2 \\
2
\end{pmatrix}
= 
\begin{pmatrix}
2 + 2T_x \\
2 + 2T_y \\
2 + 2T_z \\
2
\end{pmatrix}
\]

Figure 2.15: $T \times v'$.

Another simple effect is scaling, figure 2.16 scales the multiplied vector’s x-component by $S_x$, y-component by $S_y$ and z-component by $S_z$ as seen in figure 2.17.
2.3. 3D rendering

\[
S = \begin{bmatrix}
x & 0 & 0 & 0 \\
0 & y & 0 & 0 \\
0 & 0 & z & 0 \\
0 & 0 & 0 & 1 \\
\end{bmatrix}
\]

Figure 2.16: The $4 \times 4$ general scaling matrix.

\[
\begin{bmatrix}
x & 0 & 0 & 0 \\
0 & y & 0 & 0 \\
0 & 0 & z & 0 \\
0 & 0 & 0 & 1 \\
\end{bmatrix} \times \begin{bmatrix}
2 \\
2 \\
2 \\
1 \\
\end{bmatrix} = \begin{bmatrix}
2x \\
2y \\
2z \\
1 \\
\end{bmatrix}
\]

Figure 2.17: $S \times v$.

If we wish to translate $v$ by $(-2, -2, -2)$ and scale it by $(2,2,2)$, then it is important to remember that $A \times B \neq B \times A$ for matrices. The order of operation matters. See figure 2.18 and 2.19.

\[
\begin{bmatrix}
2 & 0 & 0 & 0 \\
0 & 2 & 0 & 0 \\
0 & 0 & 2 & 0 \\
0 & 0 & 0 & 1 \\
\end{bmatrix} \times \begin{bmatrix}
1 & 0 & 0 & -2 \\
0 & 1 & 0 & -2 \\
0 & 0 & 1 & -2 \\
0 & 0 & 0 & 1 \\
\end{bmatrix} \times \begin{bmatrix}
2 \\
2 \\
2 \\
1 \\
\end{bmatrix} = \begin{bmatrix}
0 \\
0 \\
0 \\
1 \\
\end{bmatrix}
\]

Figure 2.18: Matrix multiplication, $S \times T \times v$.

\[
\begin{bmatrix}
1 & 0 & 0 & -2 \\
0 & 1 & 0 & -2 \\
0 & 0 & 1 & -2 \\
0 & 0 & 0 & 1 \\
\end{bmatrix} \times \begin{bmatrix}
2 & 0 & 0 & 0 \\
0 & 2 & 0 & 0 \\
0 & 0 & 2 & 0 \\
0 & 0 & 0 & 1 \\
\end{bmatrix} \times \begin{bmatrix}
2 \\
2 \\
2 \\
1 \\
\end{bmatrix} = \begin{bmatrix}
2 \\
2 \\
2 \\
1 \\
\end{bmatrix}
\]

Figure 2.19: Matrix multiplication, $T \times S \times v$.

While these examples show how these matrices interact, it might not be that easy to understand in the context of 3D rendering. To understand how these matrices are used, see figure 2.20 and the following description.
Model coordinates (for instance the \( v \) in the previous examples) are inserted as input data in the model coordinates-field, which are processed by the model, view and projection matrices (in this order).

The model matrix describes position, rotation, scaling, skew and any other property of the model we are rendering. This means that this matrix defines where in the world the object is located.

Now that the world coordinates have been calculated, we need to calculate the camera coordinates, which describe where the given position is relative to our camera. This is done by applying the view matrix – containing the position, rotation and orientation of the camera – on all the coordinates. The best way to think about is as following: we are moving the entire world relative to the “camera” as opposed to moving the camera in the world.

Finally, the projection matrix is applied on all coordinates to create homogeneous coordinates which are used to describe where geometry should appear on our screen. The projection matrix describes the shape and size of the camera’s view. After this transformation, only geometry within a \(([-1, 1], [-1, 1], [-1, 1])\)-cube is kept, while the rest is clipped away and
not passed to the next rendering stage. Clipping makes rendering faster as there is less data to process on the subsequent stages.

This process can be optimized. The calculation in our case is $P \times V \times M \times v$. However, we can instead calculate $MVP = P \times V \times M$ on the CPU and instead do $MVP \times v$ on the GPU. This saves some processing power as the GPU only have to do one matrix multiplication per position instead of three.

For a complete instructions regarding matrices (in OpenGL), see opengl tutorials.net [2014].

There are certain steps in computer graphics which are common in various systems. For instance, a common step after applying transformations and clipping mentioned above is rasterization. Rasterization determines what each of the pixels in the output buffer should be. For instance: what happens when there’s geometry in front of other geometry. Two other common steps, fragment shading and vertex shading are described in the analysis of an OpenGL-program in section 2.3.5.

2.3.2 OpenGL – description

OpenGL is a cross-platform, open API for displaying 3D graphics on various devices (including phones, consoles and computers) and is capable of advanced effects. It is developed and standardized by Khronos Group consortium. This consortium includes corporations like AMD and Nvidia as well as others (for instance, see page 641 of Mark Segal [2013]).

For a complete introduction to OpenGL, see Woo et al. [1999]. However: please note that this book is somewhat dated. All concepts presented still apply, the most important changes are detailed in section 2.3.6.

OpenGL is a huge system – it has several hundred extensions and thousands of pages describing the various specification-versions, see OpenGL Registry [2014] for more. This introduction attempts to capture the important and relevant aspects of OpenGL. OpenGL provides advanced capabilities such as the ability to redefine how the rendering should work in terms of colours and geometry-transformations. It also provides many other features, such as efficient rendering of huge meshes by uploading data to the GPU and referencing this data as opposed to drawing it using function calls which also pass the data bit by bit.

OpenGL is developed solely as a graphics rendering API, without any other components such as audio, physics-simulations or other features. It is widely used in the industry both in commercial software (for instance Trine 2, see PC Games Hardware [2014]) and freeware applications (for
instance Blender, see Blender [2014]). Many Linux-programs, games and applications utilize OpenGL directly or through some abstraction layer (for instance SDL, see SDL [2014]).

OpenGL has been used to render 3D graphics in this thesis.

2.3.3 OpenGL – overview

OpenGL can be visualized as a huge state machine with appropriate functions for manipulation of almost any variable and setting.

This is the general process for using OpenGL to render the state of a game:

1. The game is launched and initializes the OpenGL state machine.
2. The game loads needed assets such as meshes (defines the geometry of objects), textures (defines the colours of the surface of meshes), vertex and fragment shaders (these are also compiled, detailed in section 2.3.4).
3. Game-simulation starts and the game starts processing.
4. Between steps in the processing, the game issues commands to OpenGL which uses the GPU to draw what is going on in the game. This action in general has the following steps:
   • Clear the output-buffer because it contains the last drawn frame, set the colour of the buffer to the colour set by glColor*( ).
   • Render the skybox (the background).
   • Render all visible meshes (groups of polygons), this step is the most complex one and consists of the following sub-steps for every mesh:
     (a) Update and upload the MVP matrix and other data to the GPU.
     (b) Issue a draw command or draw commands (through OpenGL) with appropriate input-data or referencing input-data which is present on the GPU.
     (c) The vertex shader processes the input-data using the MVP-matrix and other data using programmer-defined routines. All points (fragments) which are visible after this stage are passed to the fragment shader. The fragments not sent to the vertex shader are “culled” – removed and not accessible in the next step.
     (d) The fragment shader applies programmer-defined routines to the fragments received from the vertex shader and defines the colour they have on screen.
   • (optional) Apply post-processing effects. This is the act of using the output from the previous steps as input for another which
Figure 2.21: Screenshot of Battlefield 4, a game which relies on shaders to increase graphical fidelity.

changes the result seen on screen. This includes adding effects such as motion blur, bloom and high dynamic range (HDR).

As there is a lot going on in this case which may be hard to understand, a simple OpenGL program has been given and examined in section 2.3.5.

2.3.4 OpenGL – shaders

Modern graphics cards have been given the possibility of running custom shaders. Shaders are programs inserted into a part of the OpenGL rendering pipeline to allow the programmer to alter the rendering process as seen fit. Different shaders execute a specific part of the rendering process, this may be calculating the position of geometry (vertex shader), calculating colours (fragment shader) or adding new geometry (geometry shader). For instance: in section 2.3.3, the vertex shader mentioned in point c) in the mesh rendering process could be altered to flip the geometry horizontally or the vertex shader in point d) could be altered to show the inverse of the colours.

OpenGL allows uploading and downloading data to and from a graphics card and changing how colours work. Animation can be implemented using vertex shaders. Some meshes have additional information called bones. Bones are sets of equations taking variables describing how a character or object is bent. These equations are often modeled through simplifications of a character’s skeletal structure and with sufficient detail, allows detailing how most limbs including fingers are positioned and oriented, see Doug L. James [2005].

The style of the language resembles C with easy casting between data-
types.

A large portion of the increase in graphical fidelity within real-time 3D rendering can be accredited to more powerful GPUs and new shader capabilities.

### 2.3.5 OpenGL – basic setup

This section presents a basic C++ OpenGL-program using QT library as a wrapper and then discusses components within it. This mini-project is also included in one of the repositories.

SimpleGL.pro (project/build file):

```plaintext
QT += gui opengl
TARGET = SimpleGL
CONFIG = app_bundle
TEMPLATE = app
SOURCES += main.cpp
HEADERS += \openlwindow.h
OTHER_FILES += \shader.fs \shader.vs
```

`main.cpp`:

```cpp
#include <QApplication>
#include "openglwindow.h"

int main(int argc, char *argv[])
{
    QApplication a(argc, argv);
    OpenGLWindow window(NULL);
    window.show();
    return a.exec();
}
```

`shader.vs`:

```cpp
#version 330 core
```
2.3. 3D rendering

Chapter 2. Background

```cpp
layout(location = 0) in vec3 position;

layout(location = 0) out vec3 out_color;

layout(location = 0) uniform mat4 MVP;

void main() {
    // Apply the transformation uploaded from the CPU
    gl_Position = MVP * vec4(position, 1);
    // We only vary the green-channel. Colour varies from corner to corner on the triangle displayed
    out_color = vec3(0.5, float(gl_VertexID) / 2., 0.5);
}
```

```
shader.fs:

#version 330 core

layout(location = 0) in vec3 in_color;

layout(location = 0) out vec4 color;

void main() {
    // Receive color, use it and set alpha channel to 100%
    color = vec4(in_color, 1.0);
}
```

```
openglwindow.h:

#ifndef OPENGLWINDOW_H
#define OPENGLWINDOW_H
#include <QtOpenGL/qgl.h>
#include <QMainWindow>
#include <QGLShaderProgram>
#include <QGLShader>
#include <QGLFunctions>

// QGLWidget is the base OpenGL-render widget in QT
// QGLFunctions grants additional functions such as glGenBuffer.
class OpenGLWindow : public QGLWidget, protected QGLFunctions
{
    Q_OBJECT
}
```

public:
OpenGLWindow(QWidget *parent) : QGLWidget(parent) {
    mFrag = mVert = NULL;
    mShaderProgram = NULL;
    mVertexBuffer = 0;
}

~OpenGLWindow() {
    if (mFrag) {
        delete mFrag;
    }
    if (mVert) {
        delete mVert;
    }
    if (mShaderProgram) {
        delete mShaderProgram;
    }
}

QGLShaderProgram *mShaderProgram;
QGLShader *mVert, *mFrag;
QMatrix4x4 MVP;
GLuint mVertexBuffer;

protected:

void initializeGL()
{
    // Required for QGLFunctions.
    initializeGLFunctions();

    // Section 1
    // Load and compiler the shader-files.
    mVert = new QGLShader(QGLShader::Vertex);
    mVert->compileSourceFile("../SimpleGL/shader.vs");
    mFrag = new QGLShader(QGLShader::Fragment);
    mFrag->compileSourceFile("../SimpleGL/shader.fs");
    // Add shader files to the shader program
    mShaderProgram = new QGLShaderProgram(this);
    mShaderProgram->addShader(mVert);
    mShaderProgram->addShader(mFrag);
    mShaderProgram->link();
    // Activate the shader program
    mShaderProgram->bind();

    // Section 2
    // Create some vertex positions
    GLfloat vertexPos[9];
// These positions are equivalent to the ones which
// has been declared in Section 7.

// Create buffer and insert buffer data
glGenBuffers(1, &mVertexBuffer);
glBindBuffer(GL_ARRAY_BUFFER, mVertexBuffer);
glBufferData(GL_ARRAY_BUFFER, sizeof(vertexPos),
             vertexPos, GL_STATIC_DRAW);

// Section 3
// Reset matrix to identity
mMVP.setToIdentity();
// Set the field of view to 60, x:y-ratio to 1,
// near-plane to 0.2 and far-plane to 10
mMVP.perspective(60, 1.0, 0.2, 10);
// Move the camera to 2,2,2, looking at 0,0,0 with
// the up-direction being towards 0,1,0
mMVP.lookAt(QVector3D(2, 2, 2), QVector3D(0, 0, 0),
             QVector3D(0, 1, 0));

// Section 4
// Enable Z-buffer
glEnable(GL_DEPTH_TEST);
// Set the clear colour to black.
glClearColor(0.0, 0.0, 0.0, 1.0);
}

void resizeGL(int w, int h)
{
    // In case of resize
    glViewport(0,0,w,h);
}

void paintGL()
{
    // Section 5
    // Clear colour and Z-buffer
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    // Upload the MVP-matrix in case of changes
    mShaderProgram->setUniformValue("MVP", mMVP);

    // Section 6
    // Bind and draw data
    #if 1
    glEnableVertexAttribArray(0);
    
}
Chapter 2. Background  2.3. 3D rendering

```c
103  glBindBuffer(GL_ARRAY_BUFFER, mVertexBuffer);
104  glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 0, NULL);
105  glDrawArrays(GL_TRIANGLES, 0, 3);
106  glDisableVertexAttribArray(0);
107
108  // Section 7
109  // Alternative (slower, will not work properly with shader):
110  glBegin(GL_TRIANGLES);
111  glVertex3f(-1, 1, 0.0);
112  glVertex3f(-1, -1, 0.0);
113  glVertex3f(0, 1, 0.0);
114  glEnd();
115  
116  }
117  
118  
119  
```

Figure 2.22: Screenshot of SimpleGL – default rendering code.
2.3. 3D rendering

Figure 2.23: Screenshot of SimpleGL – alternate rendering code.

main.cpp is mostly irrelevant to us. It opens up a window with our widget which does some rendering.

openglwindow.h has most of the interesting code – the rendering. “The frame” refers to the current screen output buffer. The Z-buffer enables OpenGL to read and write to a buffer describing depth. This is used to determine what geometry is in front of other geometry, which in turn is used to determine if a fragment is visible or not.

The code does the following:

- Section 1 loads and compiles the shader-files, shader.fs and shader.vs. In other words: shaders are not compiled with the rest of the source code which is written in C++, but rather when the program actually starts.

- Section 2 defines three positions in 3D space which together span a triangle. Then it initializes a buffer for such position data and uploads the three positions to the GPU.

- Section 3 sets up our camera (described in section 2.3.1).

- Section 4 enables the Z-buffer (GL_DEPTH_TEST) and sets the clear colour to 0-alpha black. Even though this scenario won’t face issues regarding triangles being in front of each other as there is only one triangle, enabling the Z-buffer is a good idea in case more triangles are added later.
• Section 5 clears the frame and Z-buffer. Not clearing the Z-buffer causes issues in terms of triangles rendering inside each other despite being far from each other. This section also uploads the MVP-matrix to the vertex shader, into the variable MVP that was declared using this line in the vertex shader (shader.vs):

```glsl
layout(location = 0) uniform mat4 MVP;
```

The uniform keyword means that this variable is explicitly set through a command for setting uniform variables and does not vary when executing a drawing command.

• Section 6 binds the buffer initialized in section 2 and draws it as a triangle. The shaders loaded in section 1 are used.

• The vertex shader (shader.vs) receives the input vertices and processes them using the MVP matrix. It also outputs a colour to the fragment shader. This colour varies from corner to corner of the triangle, depending on `gl_VertexID` that is an attribute which holds the ID value of the vertex being processed. That is: the first corner has the colour (0.5, 0, 0.5), which is purple, the second corner has the colour (0.5, 0.5, 0.5) which is grey and the last corner has the colour (0.5, 1, 0.5) which is green (as seen in the screenshots). However, the points in between the corners do not jump discretely between these values, because of how the fragment shader works.

• The fragment shader receives the data from the previous step. However: the colour information is interpolated across the surface. This means that the corners have the values mentioned above, while for instance the centre of the edge between the first and the second point has the value (0.5, 0.25, 0.5). This is because every single fragment gets a colour which is based on a linear interpolation of the corner colours based the fragment’s position relative to the corners.

• Section 7 would draw a triangle on the same positions as section 6. However: the `gl_VertexID`-attribute wouldn’t vary, and thus the result is colours compared to what we observed using the section 6 rendering code. See figures 2.22 and 2.23. Drawing using `glBegin` and `glEnd` is deprecated.

### 2.3.6 OpenGL – changes in OpenGL

As noted in section 2.3, there have been changes in how OpenGL works since Woo et al. [1999] was published and I think it is a good idea to mention the most important change.

The most important change is how the model, view and projection matrices are handled. Prior to OpenGL 3.3, the programmer had to use `glMatrixMode(GL_MODELVIEW)` and `glMatrixMode(GL_PROJECTION)` to swap between the matrices and execute subsequent calls using functions such as `glLoadIdentity()`, `glMatrixMode()` and `glLoadIdentity()` as well as
other functions to modify the current matrix.

This system has been removed. Instead, you must manage the matrices yourself as variables within your program and upload them to your GPU using glUniforMatx4fv(). It is thus possible to pre-compute the MVP-matrix on the CPU so that the GPU only has to do one vector-matrix multiplication in the vertex shader.

See the OpenGL 3.3 specification (Segal and Akeley [2010]) for a complete overview of features and changes.

Please note that in the qt-implementation in section 2.3.5, setUniformValue has about 60 different signatures which call the appropriate OpenGL-function.

### 2.3.7 Other uses of GPUs

The capabilities of graphics cards have been further expanded by toolkits such as OpenCL and CUDA. These toolkits allow running C/C++-code on a GPU in a parallel fashion. This technology is intended for running non-render scientific applications such as differential equation solvers or generic user programs like desktop search.

The toolkits are highly developed and contain separate programs for debugging, profiling and more.

See AMD-1 [2014] for tools related to OpenCL and Nvidia-1 [2014] for tools related to CUDA.

### 2.3.8 Alternatives to OpenGL within 3D rendering

There are two consumer-grade alternatives to OpenGL:

- **Microsoft DirectX** – DirectX is Microsoft’s closed-source graphics rendering library (see Microsoft-1 [2014]). It is available in various versions on Microsoft’s Operating system Windows and their various Xbox\(^6\) consoles. It is considered to be simpler in use, but it is not officially cross-platform like OpenGL.

- **AMD Mantle** (see AMD Mantle-1 [2014]) – AMD Mantle (co-developed by DICE) is a new API focused on providing faster draw routines and greater control over the graphics pipeline. At the time of writing (July 2014), there are a few games which already use this technology, such as Battlefield 4, Sniper Elite 3 and Thief (see AMD Mantle-2 [2014]). The developers of this API have stated that Mantle will become an open API, however: no development materials or API have been released to the public at the time of writing.

---

\(^6\)The name “Xbox” is actually a contraction of “DirectX Box”
It is most likely possible to implement the proposed method and ideas described in this thesis using both of these rendering APIs.

2.4 Games

Games have become a commonplace source of entertainment. Most PCs, tablets, phones and other devices are capable of running various games. Game requirements differ from genre to genre. Some genres require some level of graphical fidelity, for instance: a horror game is not as scary without fog or shadows. Other games rely on audio or story-telling to immerse the player in the game’s world.

However, there is one requirement which to some degree exists in all genres: responsiveness. Responsiveness is a simple real-time requirement – an action should yield a response as fast as possible. While there are games which let the player ponder their next move indefinitely (such as turn-based games), responses following a player-action should still happen as fast as possible.

The following factors cause loss of responsiveness in an application: processing delay, network transfer delay and output delay. The most important delay-factors for this thesis are processing delay followed by network transfer delay.

2.4.1 Lag in games

When playing a game, the user is interacting with one or more devices such as a touch-screen, keyboard, mouse, controller or other devices. Such devices produce inputs which the game receives and handles in appropriate ways, however, this does not happen instantaneously. There’s a slight amount of delay from the time when the user produces an input through an input device to the moment when the game reacts and displays changes on the screen as the game has to process the input, process the game state and update the screen. This delay is barely noticeable (if at all) if the game is running smoothly.

The most common cause of local delay is when a game is not given the resources it needs to run properly. The two most common examples are that the game is run on a PC which does not meet the required performance levels for the graphical options used or other programs are using the same resources (usually CPU, GPU or memory), thus not giving enough processing resources to the game.

However, not all sources of delay are local. If you are playing a networked multi-player game, for instance, a multi-player shooter like Counter-Strike,

\[ \text{Network transfer delay consists of throughput delay (limitations imposed by bandwidth) and latency delay (limitations posed by physics, processing time and the route to the target).} \]
the situation is somewhat different. You still have the same sequence of user input to screen output as above, but you also have network delay. In many multi-player games there are dedicated servers which are hosting game-sessions which players can join. When several players are playing on the same server, the server receives data from all the players and broadcasts this data to everyone else. However: this data has to travel out of each player’s machine, over some sort of unspecified network – either the Internet or a local area network (LAN). While travel-times for wired (and sometimes wireless) LAN networks usually yield less than or approximately 1 ms travel time for packets between two PCs on the network, the Internet may apply a delay over 100 times greater (as seen in 2.24).

The following physical and technical limitations cause network lag: Propagation time of signals caused by the the limits of physics. Our global network (the Internet) is built up using several different technologies on different levels – fibre-cables, tp-cables and wireless technologies are all meshed together to create one network. However: one property is shared by all of these technologies: their propagation time cannot exceed the speed of light in vacuum and thus there is a lower bound on how fast communications can be achieved. While there is research which may yield faster-than-light communication (see Ma et al. [2012]), such technologies are far from ready for the average consumer.

Packet processing time is another factor. Whenever any device in the communication between two nodes (any device capable of processing network traffic) on the Internet receives a packet, it has to process it to determine what to do with it and this takes some time. If a device is
receiving more traffic than the processing unit of said device can handle or if communication links are overloaded, then even greater delays occur or perhaps even packet loss might occur.

Generally speaking: end-user devices (such as phones, computers or tablets) receive data which they process, and send data to other end-user devices. Switches and routers on the Internet receive data which is addressed to another device and determine where to send it next. There are certain exceptions to this: sometimes the switches and routers on the Internet receive special packets (for instance ICMP echo, see Postel [1981] – also known as ping and ICMP traceroute, see Niccolini et al. [2008]). Another common issue is congestion. If several users are trying to access the same host or utilizes the same connection on the Internet, packet drops and/or delays may occur. Some automatic re-adjustment may occur where packets have a given chance of taking one of several routes, but if no such alternatives are available, packets are most likely dropped rather than delayed.

### 2.4.2 Lag compensation in multi-player games

One method of increasing perceived responsiveness in online multi-player games is lag compensation. This method is most suitable for games with the following properties:

The game should have smooth motions – actors should have a movement delay when starting a movement through friction applied in the opposite direction of the movement. Likewise, instant full stops should not be possible unless hitting another object – one must instead decelerate. The reason for this is due to the fact that predictions are more erroneous if there are sudden player-controlled changes in a system. In a game, the players are from the game’s perspective, the most random element – it simply
cannot be predicted. The game must instead make assumptions based on what the players are currently doing. Therefore: making sure that sudden changes in how player input affects the game have small short-term (100 ms) consequences is a way to make the game behave slightly smoother. Basic rewinding capabilities within the game engine are required. The engine should keep recent game-states (for a few hundred milliseconds back in time) in order to handle packets which describe interactions that happened, but have not yet been recorded. For instance: if a player decides to shoot another player who is moving, the player should not have to lead (aim ahead of) the target depending on how much network lag is present if using a weapon which hits instantaneously. The server should know what the player saw while shooting at the other player and use that information to determine if the shot was a hit or a miss.

For actual implementation details on a working system used in several games, see Valve-4 [2014] and Bernier [2014], a simplified version has been described below, detailing why both of these points are important.

However, these lag compensation techniques are also applicable in games that have instantaneous acceleration. However, this may require more computational power or other resources – for instance through more frequent synchronizations.

The way lag compensation works in many games is through interpolation and extrapolation of player movements – the clients and the server try to predict where objects will be given the current state of inputs which is synchronized with regular intervals. For instance: if a player is pressing a key to move his character, this information is sent to every client in the system.

The server and all clients attempt to keep an up to date state of the game, but the server is considered the authority on the current game state. The game state describes what is going on in the game. The game state in our case and many multi-player games consists of the following elements:

- General global variables such as gravity, friction and optional states such as how many points a team has scored or how far a team has progressed in terms of an objective.
- Every single object of the physics-simulation in the game. This includes players, movable objects and flying bullets.
- Data describing what kind of inputs each player is giving.

Clients transmit the current input state of the user and the server uses this information to extrapolate player positions to be transferred to the clients. The extrapolated positions sent may be different from client to client depending on their average ping. The clients also extrapolate game states.
between updates based on client inputs which the server also broadcasts.

This interpolation works better with smooth movements — a simple calculation can show this.

Let us suppose that we have a system consisting of one server and multiple clients. We are running a multi-player first-person shooter with Newtonian physics. The players can move, jump, aim and fire. While the game state contains a lot of information, not everything is sent at all times because it would increase the amount of traffic and would also reduce performance. The following data from all clients is being sent periodically to the server from all clients:

- The current position and velocity of the player.
- The current input state — is the player moving, shooting, crouching, running or walking and what direction is the player aiming?

The following is being periodically sent from the server to all clients:

- The input states received from all clients.
- Extrapolated positions and velocities based on these input states.
- Corrections of the physics-simulation when a client has simulated something incorrectly.

Let’s assume the server is broadcasting this information to all clients with a one-way delay of 100 ms and 10 updates per second and that the clients are also broadcasting their actions to the server with 100 ms delay and 10 updates per second. For the sake of simplicity, let’s assume there is no variance and the updates are synchronized (sent out at the same time).

Let us suppose that we have a player moving at \( 2 \, \text{m/s} \) in direction \( \vec{x} \). This means that the other players are extrapolating based on states which are 200 ms old. Let us now suppose that the player decides to change his movement-input to match \(-\vec{x}\) instead and analyse the error in prediction after 200 ms given instant acceleration to \( 2 \, \text{m/s} \) compared to an acceleration of \( 10 \, \text{m/s}^2 \).

Please note that the values chosen are higher than in realistic scenarios.

**Instant acceleration:**

- Player is in position \( x = 0 \) and decides to change movement direction. The player accelerates to \(-2 \, \text{m/s}\) and broadcasts this change to the server.
- 100 ms pass, the update is received by the server and sent to the clients. The other clients currently see the player in position \( x = 0.2 \text{m} \), but the player is in actual position \( x = -0.2 \text{m} \). The absolute error is currently \( 0.4 \text{m} \).
• Another 100 ms pass and the player is now in position \( x = -0.4m \), all clients receive the update about the player having changed directions and update his position from \( x = 0.4m \) to \(-0.4m\). This corresponds to \(0.8m\) of error.

Acceleration of \(10m/s^2\):

• Player is in position \( x = 0m \) and decides to change movement direction. The player is still moving at \(2m/s\). The change in movement direction is sent to the server.

• 100 ms pass, the update is received by the server and sent to the clients. The other clients currently see the player in position \( x = 0.2m \). We can calculate the player’s position using \( s = v_0t + \frac{1}{2}at^2 \). We get \( x = 2 \times 0.1 + 0.5 \times -10 \times 0.1^2 = 0.15m \). The player’s movement speed is currently \(1m/s\) in the original direction.

• Another 100 ms pass and the player is now in position \( x = x \times 0.2 + 0.5 \times -10 \times 0.2^2 = 0.2m \). The player is now moving at \(0m/s\), all clients receive the update about the player having changed directions and update his position from \( x = 0.4m \) to \(0.2m\). This corresponds to \(0.2m\) of error.

A change of \(0.2m\) and \(2m/s\) is less noticeable compared to a change of \(0.8m\) and \(4m/s\).

After the error is discovered, the clients try to interpolate steps between the player being in the incorrect position and the updated position in order to avoid situations where it seems as if a player is teleporting.

The basic rewinding capability requirement is important as well. This is needed for checking if someone actually was capable of seeing an enemy and if that person managed to hit the enemy.

Consider two players, one which is moving in a wide circle around another stationary player. Let us again use a simple movement speed value of \(2m/s\), 100 ms network delay and updates every 100 ms. The shot fired is hitscan (hits immediately without any delay). The stationary player fires one perfect shot (from his current perspective) at the enemy, the server receives this event after 100 ms, rewinds back to the moment when the player shot (100 ms back in time), calculates the trajectory and registers the walking player as hit. Without rewinding, the player would have to fire his shot ahead of the moving player, approximately \(delay \times movement\_speed\) ahead, in this case this would be \(0.2 m\) – enough to make a shot miss. There are certain issues here. For instance: players feel as if they are getting shot “around corners”, because someone else shot them 200 ms ago, but that information was not available until later.

2.4.3 Why responsiveness matters

Responsiveness within a video game is important due to several reasons. The following applies to all games:
Delays in action-responses cause frustrations. Nielsen [1993] describes user-interactions in regards to web applications (but argues that these apply to all other applications). 0.1 seconds is the limit for the response time if the user is to feel as if he or she is directly manipulating objects, 1 second is the limit for how long a user should at most wait if he or she should feel as if freely navigating. Similar numbers most likely apply in video games.

It makes user-optimizations possible. Experienced users quite often do tasks so fast that they fail to react in time if inputs are ignored or executions are delayed.

There is a slight negative correlation between skill and delay (lower delay = better performance) according to Henderson [2001].

Certain types of games such as shooters, fighting-games or music-games are more sensitive to delay than other games such as turn-based strategy games or certain types of simulator-games.

There are certain things which are important in such reaction-demanding games, for instance: input variance between systems matters. It is important to create a consistent experience across different setups of TVs, audio equipment and user-input devices. Not being able to do so makes it harder to play the same game on different setups as many actions within games become hard-wired eye-hand-coordination sequences with precise timings. Because of this, music-games such as Rockband (see Rockband [2014]) or Stepmania (see figure 2.26) include settings for synchronizing these inputs and outputs.

Input and output delay matters in terms of task completion (see Claypool and Claypool [2006] in section 2.4.4). For instance, in a fighting game, any delay applied to actions undertaken by a player or rendering of the current game state makes the game harder for that player. Most attacks in these games have “wind-up” animations. These are animation cues which delay an attack by a certain amount, giving the opponent the opportunity to block, counter-attack with an even shorter attack or get away. Any player fighting with delay has less time to react to these animation cues and this might be enough to prevent said player from reliably reacting to an attack.

Frustrations from non-responsiveness also apply here. There are several effects which easily break the flow of the game if playing with high network delay. To name a few common effects:

- Rubber-banding\(^8\) (or “warping”, mentioned in Bernier [2014]) – the lag causes synchronization issues where the server moves the player to the position where it thinks the player is. This problem is easily noticeable when a player stops moving and then gets moved to the

---

\(^8\)Not to be confused with “rubber-banding AI”, a concept in many racing games where an opponent is “right behind you” at all times if you’re in the 1st place.
“actual” position. This is noticeable for other players seeing it happen and especially for the player it is happening to.

- Hitting things or getting hit around corners or behind obstacles – lag compensation in itself fixes a serious problem: not being able to hit moving targets reliably (see section 2.4.2). However, it also causes a less serious problem: a larger time-gap where you can be retro-actively hit. The less serious problem may become annoying if the network delay is high enough. This may also cause situations where the game kills a player after he has shot his opponent in the local simulation, but the shot does not count because the player was already dead on the server, but to the player’s confusion: a part of the shot-animation and sound was played and then halted upon death.

Figure 2.26: screenshot of Stepmania 5’s input calibration screen.

### 2.4.4 The 100 ms figure

Many papers state different figures related to how much delay is acceptable in a game. For instance, Pantel and Wolf [2002] write:

> Since we are considering the racing game as a worst-case system, for other games, e.g., first person shooter, such a presentation delay of 100 ms or even more may be acceptable.

While Claypool and Claypool [2006] write:

> The experiments measured the average hit fraction during two-player battles using high-precision weapons. There is a noticeable overall downward trend in performance as latency increases, with a sharp drop (about 35%) in accuracy at 100msec of latency.
A drop of 35% in accuracy in any multi-player game is an issue. This would in many cases be the difference between defeat and victory. While 100 ms ping is not a big issue in many other games, there are always some issues in regards to fairness, see figure 2.27.

![Figure 2.27: Figure showing how delay affects performance in games, taken from Claypool and Claypool [2006].](image)

There are games which are fast-paced, to name a few examples: Tribes: Ascend, Quake and Unreal Tournament. In these games, 100 ms ping is noticeable as acceleration is (close to) instantaneous in certain cases, movement speeds are extremely high and weapons are often not hit-scan. Weapons which are not hit-scan are harder to hit with in cases where the simulation is being re-adjusted frequently due to the reliance of these weapons on the player’s ability to calculate an opponent’s position ahead in time. Any delay in these systems is to a player’s disadvantage unless it is applied to all players’ interactions (see Mauve et al. [2002]).

In a fighting game, if an attack takes 300 ms to complete, having variable delay per player or perhaps even any delay may not acceptable in terms of fairness. As there is no way to compensate for delay in fighting-games in a way that would make the experience consistent between local-play and multi-player without introducing potentially unwanted lag in local-play, I would argue that there most likely is no such thing as acceptable delay in certain games genres in terms of fairness or user experience.

To illustrate how crucial timing is on a high level of play in these fighting games, I would like to reference some of the Street Fighter 2 communities’ resources on the Internet, see Shoryuken [2014]. Some of the players
2.5 Cloud gaming

Cloud gaming is running your game on one device (the server) and transferring the rendered output to another (the client). The latter device’s inputs are forwarded to the machine running the game.

The concept of cloud gaming arose around year 2001 when a company called G-cluster announced streaming of PC-games to handheld devices over WiFi at E3 (The Free Library [2001]). It allowed primitive clients (or thin-clients) to offer the service of graphically impressive games without the need for powerful hardware on the local device.

<table>
<thead>
<tr>
<th>Property</th>
<th>Cloud gaming</th>
<th>Local machine</th>
</tr>
</thead>
<tbody>
<tr>
<td>External machine requirement</td>
<td>Powerful machine</td>
<td>None</td>
</tr>
<tr>
<td>Local requirement</td>
<td>Thin-client</td>
<td>Powerful machine</td>
</tr>
<tr>
<td>Input-to-output delay</td>
<td>Network delay + processing delay</td>
<td>Processing delay</td>
</tr>
<tr>
<td>Internet connection speed</td>
<td>Up: low, down: high</td>
<td>Not applicable</td>
</tr>
<tr>
<td>Internet connection latency</td>
<td>Low</td>
<td>Not applicable</td>
</tr>
<tr>
<td>Sensitivity to network issues</td>
<td>High</td>
<td>Not applicable</td>
</tr>
<tr>
<td>Fidelity</td>
<td>Artifacts added through compression if lossy compression is used</td>
<td>Not applicable</td>
</tr>
<tr>
<td>Additional tasks on machine running game</td>
<td>Video- and audio- encoding and network-communication</td>
<td>Not applicable</td>
</tr>
</tbody>
</table>

Table 2.2: A short overview of cloud gaming compared to traditional gaming running on a local machine in terms of requirements and benefits for a graphically demanding single-player game.

---

9Period of time following an action in which a character is not able to act or execute certain actions
2.5.1 Benefits and challenges related to cloud gaming

Cloud gaming opens up new venues of business and practicalities.

The following benefits and aspects are central to cloud gaming as an idea:

- Certain games are only available on certain consoles. For users who don’t wish to purchase the console to be able to play the game, cloud gaming may make a given title available on other devices.

- Not everyone wants to get a high-end computer as they are either expensive or hard to carry around. Cloud gaming gives access to cheap performance which can be accessed from any location with a stable and sufficient Internet connection without the need for expensive hardware.

However, there are certain challenges which providers of these services must overcome:

- Availability – the providers must have data centres in multiple locations with good Internet connections to provide as low latency as possible. This is costly compared to having one big data centre to house many services which do not have the same real-time requirements.

- Licensing – renting out video-games in the current environment may be problematic due to Terms of Service (ToS) provided by game publishers.

- Audience – the intended audience must have good enough Internet connections to handle the service. In general, this service would be targeted at people who are not that concerned with the occasional technical issues (such as the ones described in 2.4.4) related to their game and who do not own a powerful home computer as that would render the service somewhat pointless. Additionally, one should consider the costs of needing more Internet-bandwidth versus purchasing a better PC as this may be a deciding factor in certain areas where high-speed connections are expensive.

- Scaling – Because availability is a problem (data centres should be local), the providers of these services may have to purchase more gear than what is required at the same time. This is because of peak usage hours – everyone is more likely to use the service within given time-slots within a time zone Henderson [2001]. In regard to this: even if your customer-base is in different time zones, one data centre might not be able to help another because of added delay. This may degrade the quality of service to a degree which some users may find unacceptable. However: there are certain exceptions in this case, turn-based strategy games may be playable with more latency.
2.5. Cloud gaming

• Architecture – the provider must create systems for managing save game files and settings. Not all games are well-made in this regard and this may require some work.

Cloud gaming relies on several of the subjects covered in sections 2.2, 2.3 and 2.4 and is affected by changes in these. For instance, network lag-related effects from 2.4 are also relevant to cloud gaming as the content being streamed and inputs are sent over a network. However: lag compensation is harder or impossible as we shall see in section 2.5.2. Cloud gaming is also affected by how well the video encoders (see 2.2) and games are optimized to meet the real-time requirement of the scenario.

2.5.2 Additional loss of responsiveness

As mentioned in the introduction, cloud gaming is a relatively new scenario. When playing a single-player game, the input-output delay consists of the round-trip time between the client and machine running the game. This delay is harder to compensate for compared to network lag in non-streamed multi-player games as described in section 2.4.2 as the client does not have an approximation of the game state available locally.

The following additional challenges are present when streaming games between a client and a remote machine running the game:

• When playing a game where it is possible to look around using the mouse or other input devices, it would not be possible to get the state after a rotation by the next frame as when playing a network multi-player game or a single-player game. The machine running the game must receive the input, process the new frame and send it back. This adds one entire network round trip time to the input-to-output delay for all actions in the game. Even small delays become noticeable as certain actions in games have slim timing windows.

• An input variance akin to the one mentioned in section 2.4.3 can be observed. Your location and network connection, e.g playing on your tablet while riding the train, playing on your laptop on a public network or playing on your desktop at home causes slightly different delays. This may be highly detrimental to the experience as one grows used to one kind of input delay.

If you are playing a multi-player game with your game client running on a remote server, the delay becomes enormous. First, there’s the delay between you and the machine running the game, then there’s the delay between the machine running the game and the server that machine is connected to. If each round-trip time is 100 ms, this is equivalent to 200 ms of lag, of which 100 ms is hard to compensate for. Considering how sensitive certain games are to delay, I would propose that some games such as fighting-games are not playable on a high level when using cloud gaming.
2.6 Alternative optimizations

Some work has been done on optimizing video encoding of games. I would like to reference a few works.

2.6.1 Game as Video: Bit Rate Reduction through Adaptive Object Encoding

Hemmati et al. [2013] write:

The key idea in our approach is to exclude less important objects from the game scene so that it takes less processing time for the server to render and encode the frames, and furthermore a lower bit rate would be achieved to stream the resulting video. (...)

In each frame of the gameplay, importance of each object is evaluated based on the current activity of the player (activity can be walking, running, aiming, shooting, etc.). (...)

Using normalized importance factors, we prioritize the objects and, depending on the mobile device’s capabilities, include a number of the most important objects in the current game scene. We have shown that removing objects that are irrelevant or less important to the current activity of the player does not significantly affect the user experience, including task accomplishment, fun, and immersion[26].

This approach is based on the assumption that the object behind the one you remove is easier to encode while also making the entire scene easier to render. While the latter is almost always true\(^{10}\), but the former is not always the case as you may very well be revealing more of an object which is harder to encode.

Detracting objects from a scene, only to add them later when they become relevant can create an abrupt experience where objects flicker in and out of existence. This can easily ruin the experience of playing a game.

If a scene can be rendered without a given object and it does not detract anything from the scene – including things like the “feeling” of the game, then it should by default be rendered without it.

It is hard to determine if an object (for instance an obstacle) is relevant. This may cause situations where a player may be running at high speeds towards an obstacle while trying to get away from danger only to hit said obstacle because it becomes visible in certain situations.

\(^{10}\)the exception is when few polygons would obscure the entire screen and these are removed
While this method may reduce the encoding complexity of the game, it increases code complexity by a fair amount at huge costs in terms of user experience. This may also give players an unfair disadvantage or advantage. In a competitive multi-player game, if an object is deemed unnecessary and removed, any player hiding behind it becomes easily visible. By extension, any game where there are objects you can hide behind is incapable of benefiting of this technique as one would be able to deduce enemy player positions by observing what objects are not being removed.

An object flickering in or out of view has a high chance of causing huge value-differences compared to the previous frame. This equates to more data to be encoded.

While the paper claims that the method works – the size of the frames is smaller, I would argue that the side-effects are too great, outweighing any gain.

Additionally, this method was not compared to the alternative, which is to simply increase compression level. I suspect compressing the same scene slightly heavier would decrease the frame size even further and affect the quality of experience slightly less.

### 2.6.2 Accelerated Video Encoding Using Render Context Information

Fechteler and Eisert [2010] propose a method similar to the method in this thesis:

In this paper, we present a method to speed up video encoding of GPU rendered 3D scenes, which is particularly suited for the efficient and low-delay encoding of 3D game output as a video stream. The main idea of our approach is to calculate motion vectors directly from the 3D scene information used during rendering of the scene. This allows the omission of the computationally expensive motion estimation search algorithms found in most of today’s video encoders. The presented method intercepts the graphics commands during runtime of 3D computer games to capture the required projection information without requiring any modification of the game executable. We demonstrate that this approach is applicable to games based on Linux/OpenGL as well as Windows/DirectX. In experimental results we show an acceleration of video encoding performance of approximately 25% with almost no degradation in image quality.

While this version has the advantage of being able to hook into any game and reducing encoder complexity, it does not provide perfect information. It would not be able to handle things like bones or other more complex state information.
While it is most likely possible to bypass some of these problems in simple games, more complex ones involving characters which are at times being rendered and at times not may prove highly problematic. If given access to the source code of the game, these issues would be trivial to bypass.

The results in this paper are highly promising. Encoding performance improvement with few to no issues.

### 2.6.3 Using SIMD

Single Instruction, Multiple Data (or SIMD) are instructions which execute the same operation on multiple data simultaneously. Some SIMD instructions were made with video encoders in mind – for instance: the two available mpsadbw-instructions (July 2014). For a complete description of mpsadbw and several other functions, see Intel-2 [2007]. These are instructions that are part of various extensions for the common x86-architecture used in most home computers and some of the current generation of consoles: Xbox One and Playstation 4 and as well as other instruction sets.

These functions are available in several different compilers and usually do in few instructions what would usually require many. For instance: the mpsadbw-“family” of instructions calculates parts of several different SADs – element-wise sums of absolute differences between two ordered sets of data. This is much faster than doing it one by one through the x86 standard instruction set. Shorter definitions can be found at Intel-1 [2014], for instance: this is the operation description for mpsadbw:

```
MPSADBW(a[127:0], b[127:0], imm[2:0]) {
  a_offset := imm[2]*32
  b_offset := imm[1:0]*32
  FOR j := 0 to 7
    o := j*16
    k := a_offset+i
    l := b_offset
  ENDFOR
  RETURN tmp[127:0]
}
dst[127:0] := MPSADBW(a[127:0], b[127:0], imm[2:0])
```

Explanations:

- Undefined variables are created with appropriate type on-the-fly.
• Brackets denote what bits are being written to or read from within the declared buffers. The format is foo[to:from]. For instance, a[7:0] = b[15:8] would be reading the 2nd byte in b and inserting this content into a.

• Data reads and writes are aligned from the first position read and written. This means that b[10:3] = a[8:1] yields the eight bits read from a in eight consecutive bits starting from position 3 in b.

• imm is an “immediate”, which describes the mode of the function (or offsets).

Please note that I have updated this definition to what the function actually does. The current definition at Intel-1 [2014] is incorrect and I have contacted Intel about fixing the error prevalent in both the 128-bit version and 256-bit definition of this function (July 21st 2014). The error in question is that line 6 is lacking and line 9 begins with tmp[i:15:i], meaning that there is overlap between the fields written to in tmp and that the last 56 bits go unused – this is not the case in the actual implementation.

To explain how this instruction is used, I have provided an example optimization. The result is that it is possible to calculate 8 consecutive SADs using as few as 32 of these special instructions (not included: loading the data and zeroing fields). This would otherwise take $8 \times 8 \times 8 = 512$ subtractions, additions and variable writes.

Original implementation (taken from inf5063-c63):

```c
void sad_block_8x8 (uint8_t *block1, uint8_t *block2,
                   int stride, int *result)
{
    int u, v;

    *result = 0;

    for (v = 0; v < 8; ++v)
    {
        for (u = 0; u < 8; ++u)
        {
            *result += abs(block2[v*stride+u] - block1[v*stride+u]);
        }
    }
}
```

Implementation using mpsadbw (called once for every 8 times the old implementation was called):

```c
static void sad_block_8x8x8(uint8_t *block1, uint8_t *block2,
                          int stride, int *result, int *blockoffset)
```
Chapter 2. Background 2.6. Alternative optimizations

```c

int v;
__m128i bl1, bl2, sad, minpos;
sad = _mm_setzero_si128();
int offset = 0;
for (v = 0; v < 8; ++v) {
    // Load one row from block1 and block2.
    bl1 = _mm_loadu_si128((__m128i*) &block1[offset]);
    bl2 = _mm_loadu_si128((__m128i*) &block2[offset]);
    // Calculate SADs for the row and add to the existing ones
    sad = _mm_adds_epu16(sad, _mm_mpsadbw_epu8(bl2, bl1, 0));
    sad = _mm_adds_epu16(sad, _mm_mpsadbw_epu8(bl2, bl1, 5));
    offset += stride;
}
// Find the lowest SAD and offset.
minpos = _mm_minpos_epu16(sad);
*result = _mm_extract_epi16(minpos, 0);
*blockoffset = _mm_extract_epi16(minpos, 1);
```

Line 4 declares the buffers used in the function. The buffers are 128 bit (or 16 byte) large and are automatically associated with special registers in the CPU, in short: this means that any data in this buffer can be accessed quickly.

Line 5 zeroes out the buffer which is used to contain the calculations.

Lines 9 and 10 load the data needed. In this case this is 16 consecutive bytes from the previous frame and from the current frame. The current frame is block1 which is loaded into bl1 (only 8 of these bytes are used).

Line 12 issues the mpsadbw-command. The mpsadbw command does the following calculation for arr_offset-values 0 to 7:

```
out[arr_offset] += abs(block2[arr_offset] - block1[0])
+ abs(block2[arr_offset + 1] - block1[1])
+ abs(block2[arr_offset + 2] - block1[2])
+ abs(block2[arr_offset + 3] - block1[3])
```

out should be considered an array of 8 2-byte ints. The results of these calculations are added to our sad-variable through the _mm_adds_epu16-instruction (epu16 means that the input is expected to be 2-byte ints, this function also outputs 2-byte ints). This result in turn is written to the sad-variable.

Line 13 executes mpsadbw with a different mode. This time, a different immediate (imm) is given which offsets the calculations. Passing an imm of 5 sets the first and third bits of the imm-argument (5 is 101 in binary),
which offsets which point the function starts reading from both block1 and block2 by 4 bytes. See lines 2 and 3 in the operation description near the beginning of this subsection.

This changes our calculations as following, with arr_offset varying in the same way as previously:

\[
\text{out}[\text{arr\_offset}] += \text{abs}(\text{block2}[\text{arr\_offset} + 4] - \text{block1}[4]) \\
+ \text{abs}(\text{block2}[\text{arr\_offset} + 5] - \text{block1}[5]) \\
+ \text{abs}(\text{block2}[\text{arr\_offset} + 6] - \text{block1}[6]) \\
+ \text{abs}(\text{block2}[\text{arr\_offset} + 7] - \text{block1}[7])
\]

This is then added to sad as previously with line 12.

This means that lines 12 and 13 together do the following:

\[
\text{out}[\text{arr\_offset}] = \text{abs}(\text{block2}[\text{arr\_offset}] - \text{block1}[0]) \\
+ ... + \\
\text{abs}(\text{block2}[\text{arr\_offset} + 7] - \text{block1}[7])
\]

This equates calculating 1/8th of the SADs between the macroblock being matched and 8 (horizontally) consecutive possible matching blocks in the previous frame.

Once this has been done 8 times at different offset-values (line 14, stride is the input video’s width), the 8 SADs are stored in the sad-variable.

Lines 17-20 find the lowest values in a buffer which is interpreted as 8 2-byte values – in this case, the 8 SAD values. The output-buffer is also 2-byte aligned and position 0 is the lowest value while position 1 is that value’s position previous position.

There are certain issues in regards to using such instructions:

- The code is harder to understand. Reading the optimized version and understanding what it does is only possible if given context and the unoptimized implementation or if you possess great knowledge about what these instructions do.

- Certain machines are not be able to execute this code due to the lack of these instructions.

- Not all search-algorithms can be optimized using mpsadbw, if one was to improve diamond search (see Shan Zhu [1997]) using the improved sad-function, one would end up calculating many not needed SAD-values. However, there are other functions one could use which could benefit diamond search and other algorithms.
Chapter 3

Proposed method

The purpose of this thesis is to provide ideas for optimization of video encoding when the input stream is rendered by a program using a 3D renderer.

3.1 Idea

This thesis focuses on one main idea:

- Calculate the scene motion by keeping the MVP-matrix (or matrices, depending on implementation), positions, rotation-states and bones or equivalent for the previous state and calculating the difference by looking at where a given object was in the previous frame. This calculation is akin to calculating motion blur\textsuperscript{1}, but using the data for other purposes.

- Insert this data into the video encoder as motion vectors and skip the entire motion estimation step.

Not skipping motion estimation and instead using the motion vectors derived from OpenGL as initial guesses for the best motion vector in the encoder has been tested. There is no doubt that the game or application will run slower with the proposed method implemented. However, the idea is that the added operations within the game will remove a sufficient amount of operations within the video encoder to such a degree that there is a performance benefit.

3.1.1 Use of a game state

There are fundamental differences between recording the real world through the use of a camera and the world within a game or application. Cameras can’t directly evaluate an object’s position in the real world (some devices can approximate this information, such as the Xbox accessory\textsuperscript{1}).

\textsuperscript{1}It can actually be used for that purpose as well
"Kinect", see Microsoft-2 [2014]), while in a game all positions are known. Therefore: keeping a state, calculating the next and checking the differences in all positions yields perfect knowledge of the movement of every single object within the game.

For instance, keeping the game-states used to render the screenshots in figure 3.1 allows determining the exact motion of all objects.

![Figure 3.1: Two slightly different screenshots from Insurgency.](image)

### 3.1.2 Movement within a simulation rendered within OpenGL

To display the state of a simulation, a renderer is needed. In this case, OpenGL has been used. Every tuple in this section contains data on the form of (x (horizontal), y (vertical)). x and horizontal, y and vertical may be used interchangeably. The prefix pix indicates that the value given is in number of pixels, the prefix u indicates that the value is given in units (described below) and the lack of a prefix indicates that it is provided in homogeneous coordinates.

When rendering a state, if the MVP matrices and position of geometry used to render the previous state was kept, then \( \text{MVP}_{\text{cur}} \times \text{pos}_{\text{cur}} - \text{MVP}_{\text{prev}} \times \text{pos}_{\text{prev}} \) yields motion vectors of visible geometry in OpenGL’s homogeneous coordinate system. The MVP matrices also contain the position of the camera and the object, thus any object and camera movement is compensated for. \( \text{pos}_{\text{cur}} \) and \( \text{pos}_{\text{prev}} \) includes calculating the position of geometry in regards to bones and other mesh-augmenting equations in the shader (except for the ones caused by the MVP matrix) based on the current and previous states. The x and y motion data is kept and the z motion data (depth) is discarded as it is not needed.

The homogeneous system in OpenGL is defined as two width-units horizontally and two height-units vertically – regardless of render size in pixels. See figure 3.2. In this system, geometry located on the right edge of the render has an x-component of -1 while on the left edge, it has a x-component of +1. Geometry located on the bottom edge of the render has a y-component of -1 and while located on the top edge, it has a y-component of +1. Geometry located in the center of the screen thus has the position of...
Figure 3.2: The x and y-components of the rendered homogeneous space in OpenGL.

For a motion vector in this system: the x-component is scaled according to render width while the y-component is scaled according to render height. This means that a movement in the homogeneous coordinate system of (0.1, 0.1) traverses the scene unequally in the x and y-directions in number of pixels if the aspect ratio of said render is not 1:1.

Let us assume that we are rendering a game at a resolution of 1024x768. Positions given are relative to the bottom left corner. Let us assume we have an object in pixel position (30, 30), see figure 3.3. The homogeneous coordinates of this pixel position are \((\frac{\text{pix}_\text{pos}_x}{\text{pix}_\text{render}_\text{width}} - 1, \frac{\text{pix}_\text{pos}_y}{\text{pix}_\text{render}_\text{height}} - 1)\) or \((-0.94140625, -0.921875)\). Let us assume that this object has moved in one frame from pixel position (30, 30) to (90, 90) through undefined changes in the MVP matrix, see 3.4. The motion in pixels is (60, 60), while the motion within OpenGL in terms of homogeneous coordinates is given by \(\text{pos}_\text{cur} - \text{pos}_\text{prev}\) or \((\frac{\text{pix}_\text{cur}_x - \text{pix}_\text{prev}_x}{\text{pix}_\text{render}_\text{width}} - \frac{\text{pix}_\text{cur}_y - \text{pix}_\text{prev}_y}{\text{pix}_\text{render}_\text{height}})\) which in this case equates to \((0.1171875, 0.15625)\).
The motion within the homogeneous coordinate system is written to a texture with two float-values \((x, y)\) per pixel in the render and then transferred to the main memory of the computer where it is accessible by the video encoder. Please note that only motion of currently visible geometry is recorded. Therefore, an object leaving the rendered area will not have any motion vectors recorded.

Before the motion vectors in OpenGL’s homogeneous coordinate system can be used, a conversion to the video encoder’s coordinate system must be applied. Most video encoders (including inf5063-c63 and x264) operate with a two-dimensional coordinate system with the origin in the upper left corner. Positive vertical direction is downwards and positive horizontal direction is to the right. A step on either axis in positive direction for a given channel will now be called a unit (or \(u\)). The coordinates
Chapter 3. Proposed method

3.1. Idea

for a given channel in the encoder spans \([(0, u_{pix\_channel\_width} - 1), [0, u_{pix\_channel\_height} - 1])\) units, see 3.5. This means that the video encoders operate with a differently scaled x-axis (pointing in the same direction) and a reversely and differently scaled y-axis compared to OpenGL.

Let us assume that Y’CbCr 4:2:0 is used to provide image-data to the video encoder. This means full resolution for the Y’-channel and quarter resolution for the Cb and Cr channels. Let us assume a render resolution of 1024x768 outputting the input frames for the video encoder and motion vectors. Let us also assume that the encoder outputs encoded video which also has a resolution of 1024x768. To convert coordinates or motion within the homogeneous coordinate system to coordinates for a given channel in the video encoders, the following operation yields the motion vectors in the encoder’s coordinate system: \((\frac{x}{2} pix\_channel\_width, \frac{y}{2} pix\_channel\_height)\)

where x and y are the horizontal and vertical component of the motion vector derived from OpenGL in the homogeneous coordinate system.

Therefore: a movement of (0.1, 0.1) in the homogeneous system corresponds to a movement of approximately (51.2, -38.4) units in the Y’-channel and (25.6, -19.2) units in the Cb and Cr-channels.

However, the encoders match macroblocks in the current frame with data in the previous frame. Therefore, the sign of the motion vectors should be changed because the encoder needs motion vectors to describe the relative origin of the geometry in the pixel rather than the motion which brought it into the pixel. Because of this, the motion vectors in the previous case become (-51.2, 38.4) for the Y’-channel and (25.6, -19.2) for the Cb and Cr-channels.
Before sending this output to the video encoder, this data should be optimized for use. It would be unnecessary to store or send data to the encoder which the encoder does not use. A way to conserve some storage space or bandwidth would be to provide a map of motion vectors with one motion vector for every macroblock instead of a map of motion vectors with one vector for every pixel. Therefore, one motion vector should be chosen per macroblock using the data describing the movement of the geometry within that block. There are several approaches to this: use the middle-point, use the average value, use the mean value or corner values of the motion pixel motion vectors. The first option is by far the simplest and computationally less expensive and has thus been used (using the 4th pixel from the top and left in each macroblock), see figure 3.6. The motion vector data is then quantized in discrete integer steps, stored as integers and sent to the video encoder along with a frame in Y’CbCr 4:2:0.

The last step of this method is to either augment or skip motion estimation. This is done by either skipping the step entirely and provide the motion vectors used in the subsequent untouched steps of the encoder or by providing an additional starting point for the motion search.

The dashed lines separate the different 8x8 macroblocks. The dot in every macroblock shows which pixel’s motion vector information is used for that macroblock.

Figure 3.6: A 64x64 unit excerpt showing which pixel-information is used as to determine the motion vectors in each macroblock.
3.1.3 Modern GPU technology

The recent innovations within GPU technology mentioned in section 2.3.7 has given us the tools needed to exploit the parallelism of a GPU to help the video-encoding process. While this may be done by using OpenCL or CUDA within an encoder, it is most likely more efficient to instead attempt to exploit these capabilities within the program running and showing the output.

In this case, we are interested in creating data which describes the motions in the scene and downloading this data from the GPU to the CPU.

The movement-calculations could be done using a function such as gluProject, but this is inefficient as this function runs on the CPU and should only be used for the debugging of a program. Instead, a custom shader has been used to output this data to a buffer which then can be downloaded from the GPU’s internal memory to CPU-accessible memory.

As a consequence, there is less overhead in swapping between what program is running on the GPU and more importantly: complete access to the state of the simulation within the application.
Chapter 4

Implementation

This chapter describes the implementation of the proposed method and related issues and details.

4.1 The system

A complete cloud gaming setup consists of three main components:

- The remote server running the game.
- The client receiving the video and audio-output of the game running on the server.
- The network connection between the server and client.

The software on the server consists of three components: the game being run and providing audio and video-input for the video encoder, a video encoder which is encoding the stream that is being sent to the client and the program that is receiving the input-information sent by the client.

The software on the client consists of two components (which are most likely one program). It consists of the software forwarding input-commands to the server and the software receiving the video stream from the server – decoding it and displaying it.

The end result is a system where a user is sitting on a relatively weak machine with high Internet bandwidth available, playing a game on a remote server with as low latency as possible.

4.1.1 Choice of components for testing

The goal and scope of the suggested method (as described in sections 1.3 and 1.4) is to implement a proof of concept. The implementation consists of two of the components mentioned in section 4.1, one of which I have chosen for testing and the other is needed to generate data for these tests. The first component is the game run on the server. However, since there are no games which I can use to directly generate the prerequisite data, I must generate this data through other means. This means either
modifying a game to create the data or an application which generates something resembling video game output – for instance a demo scene. Demo scenes are programs which render 3D graphics for no other purpose but demonstration of capabilities of rendering\(^1\) or as proof of concepts for various techniques. The advantage of using a demo scene instead is that it is simpler to modify a demo scene compared modifying a full game and yet will output comparable data.

The other component is the video encoder on the server receiving the data from the game or demo scene.

I have not implemented the entire system listed above due to the following reasons:

- Inputs from the client-machine are not a part of the system which affects how the video is encoded in the current scenario.

- The decoder on the client-machine remains the same even when the proposed method is applied on the encoder.

- Because other components in the system are unaffected, reducing resources spent on the game and the video encoder in total will not cause any side-effects on performance elsewhere. Improving the total performance of these sections will also have a net benefit on the entire system.

### 4.1.2 Changes in the components

The following changes have been made to the demo applications substituting the games:

- Retrieve pixel data from the frame buffer, convert it to the Y'CbCr colour space, dump it as Y'CbCr 4:2:0 to a file.

- Keep state information for previous frame and use this to calculate motion vectors using shaders as described in chapter 3.

- Retrieve estimated motion vectors and dump them in the appropriate way to two different files – one for each encoder. This is because the x264 encoder operates with 16x16 macroblocks and the inf5063-c63 encoder operates with 8x8 macroblocks.

The following changes have been made to both video encoders:

- Added options to skip motion estimation by reading an additional file containing motion vectors.

There are additional changes per encoder listed in sections 5.2.1 and 5.4.1.

\(^1\)or a person’s capabilities of coding and creating impressive-looking scenes
4.2 Demo scenes

I have used two different demo scenes to output data for the encoders. They vary in theme, but the output data matches the specification described in the section 4.3.1.

4.2.1 OpenGL-tutorials.org tutorial-based demo scene

This demo scene features a statically placed camera which can be rotated and simple shaders.

The camera is rotated using the mouse and no advanced interaction is possible. You end the program with ESC. In addition, I have integrated the inf5063-c63 encoder into this application, making it possible to dump the framebuffer and potentially the motion vectors as well and then encode it and write it directly to a .c63-file.

This demo scene requires the inf5063-c63 code to be available as the encoder-source is used directly.

4.2.2 ValeDecem

This demo scene features both a moving camera and lights as well as a moving and rotating object. It has more advanced shaders and features materials with specular reflections and a skybox.
There are advanced interaction features in this demo, recording can be toggled on and off using the R-button on your keyboard, D shows a special interaction menu where you can change shader-values and certain settings. It is also possible to manually move the camera around as opposed to following the demo’s program by using the mouse and arrow keys. I also fixed a few issues within the code-base which were causing some errors.

4.3 Implementation details

This section covers implementation details of the proposed method within a program rendering a 3D scene and the video encoder encoding the output from said program. It also covers the demo scenes used and limitations of the proposed method. To download the modified source code, see section 4.5.2.

4.3.1 Standards

During testing of the proposed method, some standards were set. Both demo scenes adhere to the following:

- Video-output: Y’CbCr 4:2:0, resolution: 1024x768 (the encoders do not require this resolution).

- Motion vector output: two 16-bit signed ints, x first, then y. Two different files are dumped: one which dumps the motion vector value of the pixel at the centre of every 8x8 region for the two first channels and another file containing the centre of every 16x16 region for the
two first channels (Cb and Cr can share data as the movement will be the same). The 8x8 file is used by the inf5063-c63 encoder while the 16x16 file is used by the x264-encoder. The vectors are dumped row-wise, starting from the upper left corner.

The output-format from the encoders does not matter as it isn’t used by any internal component.

### 4.3.2 Dumping pixel data

This section details how the demo scene applications write image data to a file. The code is taken from the ValeDecem demo scene and includes the most important parts.

After rendering, the following code is executed:

```cpp
sf::Vector2u size = m_window.getSize();
size_t buf_size = size.x * size.y;
GLenum *buffer = new GLubyte[buf_size * 4];

glReadPixels(0, 0, size.x, size.y, GL_RGBA,
             GL_UNSIGNED_BYTE, (GLvoid *)buffer);

uint8_t *Y, *Cb, *Cr;
Y = new GLubyte[buf_size];
Cb = new GLubyte[buf_size/4];
Cr = new GLubyte[buf_size/4];

convertToYCbCrArrays(buffer, Y, Cb, Cr, size.x, size.y );

fwrite(Y, buf_size, 1, m_col_file);
fwrite(Cb, buf_size/4, 1, m_col_file);
fwrite(Cr, buf_size/4, 1, m_col_file);
```

This retrieves all the data used to render the scene, converts it to Y’CbCr 4:2:0 and writes the contents to file.

convertToYCbCrArrays and related macros:

```cpp
#define YR 0.257
#define YG 0.504
#define YB 0.098
#define CRR 0.439
#define CRG −0.368
#define CRB −0.071
#define CBR −0.148
#define CBG −0.291
```
#define CBB 0.439
#define YS 16
#define CS 128
#define YVAL(buf, index) (buf[index] * YR + buf[index+1] * YG + buf[index+2] * YB + YS)
#define CRVAL(buf, index) (0.25 * (\n    CRSUBVAL(buf, index) + \n    CRSUBVAL(buf, index+4) +\n    CRSUBVAL(buf, index+width*4) +\n    CRSUBVAL(buf, index+width*4+4)\n))
#define CRSUBVAL(buf, index) (buf[index] * CRR + buf[index+1] * CRG + buf[index+2] * CRB + CS)
#define CBVAL(buf, index) (0.25 * (\n    CBSUBVAL(buf, index) + \n    CBSUBVAL(buf, index+4) +\n    CBSUBVAL(buf, index+width*4) +\n    CBSUBVAL(buf, index+width*4+4)\n))
#define CBSUBVAL(buf, index) (buf[index] * CBR + buf[index+1] * CBG + buf[index+2] * CBB + CS)

void convertToYCbCrArrays(GLubyte *input, uint8_t *Y, uint8_t *Cb, uint8_t *Cr, int width, int height) {
    uint current = 0;
    int i, j;
    for (i = height-1; i >= 0; i--) {
        for (j = 0; j < width; j++) {
            Y[current++] = YVAL(input, i*width*4 + j*4);
        }
    }
    current = 0;
    for (i = height-2; i >= 0; i-=2) {
        for (j = 0; j < width; j+=2) {
            Cb[current++] = CBVAL(input, i*width*4 + j*4);
        }
    }
    current = 0;
    for (i = height-2; i >= 0; i-=2) {
        for (j = 0; j < width; j+=2) {
            Cr[current++] = CRVAL(input, i*width*4 + j*4);
        }
    }
Please note that some of the macros are defined on one line in the code (a \\ is needed to define multi-line macros).

### 4.3.3 Dumping motion vectors

This code is taken from ValeDecem. Please note that in this case, buffer 2 is the last draw buffer – the one which is attached for dumping motion vectors.

Initializing the output buffer for the motion vectors:

```cpp
1. glGenTextures(1, m_out_textures);
2. glBindTexture(GL_TEXTURE_2D, m_out_textures[0]);
3. glTexImage2D(GL_TEXTURE_2D, 0, GL_RG32F, width, height, 0, GL_RG, GL_FLOAT, NULL);
```

The first two lines create the buffer (a texture), the last one declares the format and size. This provides two floats per fragment (pixel) rendered.

Defining draw-buffers and attachment-positions:

```cpp
1. glFramebufferTexture2D(...)
2. glFramebufferTexture2D(GL_FRAMEBUFFER, GL_COLOR_ATTACHMENT2, GL_TEXTURE_2D, m_out_textures[0], 0); // Call for our texture from the last step.
3. (...)
4. m_drawbuffers[0] = GL_COLOR_ATTACHMENT0;
5. m_drawbuffers[1] = GL_COLOR_ATTACHMENT1;
6. m_drawbuffers[2] = GL_COLOR_ATTACHMENT2;
```

Declaring which buffers to use:

```cpp
1. glDrawBuffers(3, m_drawbuffers);
```

Activate appropriate shaders which are listed below:

```cpp
1. m_shader.m_program.use();
```

The vertex shader:

```cpp
1. #version 330 core
2. #extension GL_ARB_explicit_uniform_location : enable
3. layout(location = 0)in vec3 Position;
4. layout(location = 1)in vec3 Normal;
5. layout(location = 2)in vec2 UV;
6. layout(location = 0)uniform mat4 Projection;
7. layout(location = 1)uniform mat4 View;
```
4.3. Implementation details

```cpp
layout(location = 2) uniform mat4 Model;
layout(location = 3) uniform mat4 OView;
layout(location = 4) uniform mat4 OModel;

out vec3 v_Normal;
out vec2 v_UV;
out vec4 v_new_pos;
out vec4 v_old_pos;

void main(void)
{
    gl_Position = Projection * View * Model * vec4(
        Position , 1.0);
    v_old_pos = Projection * OView * OModel * vec4(
        Position , 1.0);
    v_Normal = (Model * vec4(Normal, 0.0)).xyz;
    v_UV = UV;
}
```

Fragment shader:

```cpp
#version 330 core

in vec3 v_Normal;
in vec2 v_UV;
in vec4 v_new_pos;
in vec4 v_old_pos;

uniform sampler2D Diffuse;
uniform sampler2D Specular;

layout(location = 0) out vec4 Color;
layout(location = 1) out vec4 Normal;
layout(location = 2) out vec2 MovVal; // This is our output−buffer

void main(void)
{
    float spec = texture(Specular, v_UV).r;
    Color = texture(Diffuse, v_UV);
    Normal = vec4(v_Normal, spec);
    vec3 oldScrPos = (v_old_pos/v_old_pos.w).xyz;
    vec3 scrPos = (v_new_pos/v_new_pos.w).xyz;
    MovVal = (scrPos − oldScrPos).xy;
}
```
Upload the current and previous uniform data and then draw the objects. In ValeDecem, this is done this way:

```cpp
1. glUniform1f(m_texturedMeshGLSL.m_specularLocation, 10);
2. glUniformMatrix4fv(m_texturedMeshGLSL.m_modelLocation, 1, GL_FALSE, glm::value_ptr(m_modelMatrix));
3. glUniformMatrix4fv(m_texturedMeshGLSL.m_viewLocation, 1, GL_FALSE, glm::value_ptr(m_viewMatrix));
4. glUniformMatrix4fv(m_texturedMeshGLSL.m_o_modelLocation, 1, GL_FALSE, glm::value_ptr(o_modelMatrix));  // previous model matrix
5. glUniformMatrix4fv(m_texturedMeshGLSL.m_o_viewLocation, 1, GL_FALSE, glm::value_ptr(m_o_viewMatrix));  // previous view matrix
6. glUniformMatrix4fv(m_texturedMeshGLSL.m_projectionLocation, 1, GL_FALSE, glm::value_ptr(m_projectionMatrix));
7. // Draw an object
8. m_object.render();
```

And lastly, the following code is executed to download the motion vectors to CPU-accessible memory and store them in two different files – one for the inf5063-c63 encoder and one for the x264 encoder:

```cpp
sf::Vector2u win_size = m_window.getSize();
1. glBindTexture(GL_TEXTURE_2D, m_gbuffer.getOutTexture(0));
2. GLfloat *floatBuf = new GLfloat[win_size.x * win_size.y * 2];
3. glGetTexImage(GL_TEXTURE_2D, 0, GL_RG, GL_FLOAT, floatBuf);
4. size_t buf_size = 2 * win_size.x * win_size.y / (64);
5. int16_t *Y_mv = new int16_t[buf_size];
6. int16_t *UV_mv = new int16_t[buf_size / 4];
7. dumpMv(m_mv_file8, 8, floatBuf, Y_mv, UV_mv, win_size);
8. dumpMv(m_mv_file16, 16, floatBuf, Y_mv, UV_mv, win_size);
```

The `dumpMv` uses the `convertToMVArrays`-function and then dumps the contents to a file:

```cpp
void convertToMVArrays(GLfloat *input, int16_t *mv_y, int16_t *mv_cbcr, int stride, int width, int height
```
4.3. Implementation details

```c
int i, j, index, current;
current = 0;
for (i = height−stride/2 + 1; i >= 0; i -= stride) {
    for (j = stride/2 − 1; j < width; j += stride) {
        index = (i*width + j) * 2;
        mv_y[current++] = round(−input[index]*width / 2.);
        mv_y[current++] = round(input[index+1]*height / 2.);
    }
}
current = 0;
for (i = height−stride + 1; i >= 0; i -= stride * 2) {
    for (j = stride − 1; j < width; j += stride * 2) {
        index = (i*width + j) * 2;
        mv_cbcr[current++] = round(−input[index] * width / 4.);
        mv_cbcr[current++] = round(input[index+1] * height / 4.);
    }
}
```

Please note that the macros are defined on one line in the code (you need a \ to define multi-line macros).

4.3.4 Modifying the encoder

The encoders have been modified to accept the output produced in section 4.3.3.
Here are the most important parts from the inf5063-c63-encoder:

```c
/* Motion Estimation */
if (cm−>curframe−>orig−>MVY == NULL || cm−>curframe−>
    orig−>MVcbCr == NULL) {
    c63_motion_estimate(cm);
} else {
    c63_motion_read(cm);
}
```

This part skips motion estimation is motion vector data is present.

```c
static void c63_motion_read(struct c63_common *cm) {
    c63_motion_sub_read(cm, 0);
    c63_motion_sub_read(cm, 1);
    c63_motion_sub_read(cm, 2);
}
```
Chapter 4. Implementation

4.4 Issues

This part reads motion vector data from the file and makes sure that it is within bounds. int cc is in this case the channel currently operated on (0 = Y, 1 = Cb, 2 = Cr) and cm->padw[cc] is the width of the frame in pixels.

4.4 Issues

While the proposed method works great for simple scenes, there are a couple of scenarios which may prove problematic. Possible solutions for the issues are provided as well.
4.4. Issues

Chapter 4. Implementation

4.4.1 Motion vector value limitations

Many games – especially first-person shooters – typically involve the player rotating quickly (180 degree rotations in less than half a second are common). While OpenGL may be able to calculate correct motion vectors, most encoders won’t support this kind of extreme movement.

For instance: motion vectors were int8_t in the original implementation of the inf5063-c63-encoder, meaning that the value-range for the motion vectors was at most [-128, 127]. This means that there is a possibility that the output-format of the encoder needs to change as well as the internals of the encoder and decoder if such motion vectors are to be supported. Such changes may have dramatic consequences, such as slow adoption rates or the need for making an entirely new standard – this does not imply writing an encoder from the ground up as you can fork an existing implementation and make changes. To avoid these consequences, falling back to normal motion estimation or re-encoding the entire block may be a preferable alternate solution.

4.4.2 Transparency

Visible objects behind mostly transparent ones are incapable of reporting any motion vectors unless special care is taken. I would suggest using separate shaders which do not alter the motion vector buffer if the transparency of an object is under a certain threshold. This can also be done using an if-test within the shader, but this may cost slightly more processing power as the previous vertex position is calculated before the colour.

Figure 4.3: Screenshot of Half-Life 2 showing the insides of a moving train with windows.
4.4.3 Small objects

In this scene, a smaller object is passing in front of the scene, disrupting the rest of the scene by passing through the center of several macroblocks. This can be alleviated by calculating the average motion of a macro-block, rather then the center. However, this costs more processing power.

4.4.4 Advanced effects

Advanced effects such as objects changing colour or rendering your scene onto a texture, then rendering your scene through an effect may also cause issues. Such effects make it harder to make an effective motion estimation for the scene, if it is at all possible.

![Figure 4.4: Screenshot of Half-Life 2 showing ripples and effects on the water surface which prove problematic to the proposed method.](image)

Objects changing colour may also be problematic as this does not constitute a movement. Perhaps falling back to normal motion estimation is the easiest solution for these cases.

4.4.5 Viewport distortion, rotation of objects, objects approaching the camera

Rotation of objects may expose vertices which have previously not been visible. Even though the movement of these vertices is simple to calculate, the previous state does not contain any information about what these vertices look like. In other words, while the proposed method provides “correct’ 3D’ motion vectors, they may not be always useful as they point at irrelevant data in 2D space. Also, as objects move in 3D space, interpolations across textures on moving objects and objects moving away or towards the camera may exhibit patterns which may prove problematic
4.4. Issues

for the proposed method.

There are no proper workarounds within the current technology, so falling back on normal motion search may be the best solution in most cases. However, one method is proposed in section 6.2.1 to alleviate issues related to objects moving from or towards the camera.

4.4.6 HUD

z HUDs\(^2\) may also cause issues as these are often statically placed 2D objects which are rendered on top of a 3D game render. Movement occurring behind these HUD-objects may still have their motions recorded. This may prove especially problematic if the GUI-elements are not 100% opaque.

The solution for this is analogous to the one for transparency (section 4.4.2): set the motion vectors of such areas to 0 if the opaqueness is over a given threshold.

Another issue is handling temporary GUI-elements. Figure 4.5 and 4.6 shows two different screenshots from Dark Souls: Prepare to Die Edition. This game features GUI-elements which appear in certain situations. For instance: the "Rest at bonfire"-dialogue box is only visible while near a bonfire.

A solution would be to render the GUI-elements with a shader which also modifies the motion vector buffer and sets it to zero in the areas affected or even better: calculates motion – if any – in the GUI. However, in some cases the GUI-elements are drawn using other libraries than the library used for rendering the 3D graphics. If this is the case, one could solve this issue by having code that notices which elements are being displayed and declares areas these GUI-elements are covering as non-moving. This may not a perfect solution as parts of the GUI may be more translucent than not.

\(^2\)Head-up display: often used for showing vital player information such as health or ammo status
4.4.7 Post-processing effects and shadows

Effects such as bloom, reflections or shadows may also cause issues. For instance, the shadow of an object passing over a non-moving surface does not cause movement where the shadow is cast.

There are no simple solutions for moving shadows on surfaces.
4.4.8 Anti-cheat

Even if it is possible to implement this scheme for a third-party game without complete source code access using function call hooks or DLL-hooks\(^3\), one would still face issues from within the applications themselves—especially in multi-player games. Various developers have included cheat detection within their games, for instance: VAC\(^4\) and BattleEye\(^5\).

No one but the developers of these systems know what the anti-cheat systems actually do, but it is unlikely that none of the systems check against the aforementioned hooks as this is a common-place method of implementing wallhacks that allows cheating players to see and hit enemies through walls.

This problem is not easily solved. If you decide to give direct access to the game state using an API, the cheaters will be indistinguishable from other players streaming their game as it is possible to create cheats which behave as if they were a recording or streaming program.

An implication of this would be that this method needs to be implemented by the developers of the games.

4.4.9 Artifacts

As mentioned in section 2.2.2, certain images are not easily compressed using the discrete cosine transform and this may extend to games as well. If a video encoder with settings which are lossy is applied, some artifacts must be expected in any case.

However: certain games may be more prone to these issues than others. While this is not an issue directly caused by the proposed method, it is making the situation slightly worse, see table 5.1 in section 5.2.3.

\(^3\)methods for interrupting a function call to execute custom code as well as the original function call

\(^4\)Valve Anti-Cheat, an anti-cheat program used by Valve in their own games (such as Team Fortress 2 or Counter-Strike) as well as certain third party developers

\(^5\)created by Bohemia interactive for the Arma-series
4.5 Source code

To demonstrate the capabilities of the optimization proposed method third-party software has been utilized and modified.

4.5.1 Prototype components

The first demo scene (tutorialdemo) uses tutorial source code which can be found here: http://www.opengl-tutorial.org/download/
The specific tutorial used was tutorial 14.

ValaDecem can be found here: https://github.com/Mahoimi/ValeDecem

The initial encoder used was the inf5063-c63-encoder used for a course at the University of Oslo – INF5063 – Programming heterogeneous multi-core architectures.
The course-page can be found here: http://www.uio.no/studier/emner/matnat/ifi/INF5063/
The encoder-source can be found here: https://bitbucket.org/mpg_code/inf5063-codec63/

The x264-encoder can be found here: http://www.videolan.org/developers/x264.html
4.5.2 Modified code

All the code used, modified and created during the work on this thesis is available under various open source licences. To simplify downloading the code and to have a unified list of references to the code, I have made a repository with all the source code. In order to acquire the reference-list and all the code, please execute the following command:

1 $ git clone https://bitbucket.org/TZer0/mastercode.git

In order to run this commands, git is required. See README.md in the root directory for information regarding the source code.
The repository can also be found at http://tinyurl.com/odfovg.
The reason I am providing this is to give access to complete implementations as the snippets throughout this text contain only the most relevant code.
Chapter 5

Findings

This chapter describes the result from every prototype compared to each other and the original encoders. Please note that the time spent recording the demo scene used is not included in any measurements.

5.1 Test metrics and goals

I have defined certain goals in section 1.3. This section covers what achievements for each goal qualify as reaching the goal for each metric.

We are applying these metrics only to the video encoder, but with some margins to cover for additional processing time within the demo scenes generating the output. “Modified encoder” refers to the encoder with the modifications specified in chapter 3. “Original implementation” refers to the unmodified encoder.

Processing time:

- Slower: the modified encoder spends more time than the original implementation’s run-time.

- Neither slower or faster: the modified encoder spends 97.5%-102.5% of the original implementation’s run-time.

- Barely faster: the modified encoder spends about 80%-97.4% of the time the original implementation’s run-time. This is considered barely faster based on the assumption that adding the mechanisms to create the required data for the proposed method causes the game to spend slightly more time processing.

- Faster: the modified encoder spends less than 80% of the original implementation’s run-time.

Output stream size:

- Larger: the modified encoder outputs files larger than the original implementation. I have defined this as 102.5% or larger of the original stream size.
5.2. Initial inf5063-c63-prototype

The first implementation of the proposed method was made using the inf5063-c63 video encoder source code. This implementation serves as a simple proof of concept of the proposed method.

5.2.1 Status of the prototype

The following changes were done within the initial prototype:

1. Changed internal encoder data to allow longer motion vectors.
2. Ported the code to C++. This was done to make it possible to integrate the encoder in tutorialdemo.
3. Added customizable search range to create figure 5.1.

5.2.2 Performance improvement

This implementation runs much faster.

The inf5063-c63-encoder is written in pure C++ without any optimizations. Of all the operations in the program, the motion estimation takes up about 95% of the execution. Thus, replacing this operation with a single read from an array improves performance significantly. The encoder encodes several frames in the time it took to encode one.

Running tutorial-demo in live encoder mode with and without motion calculation through OpenGL easily demonstrates this.

5.2.3 Output quality

This implementation has certain quality issues. While these issues are not caused by the proposed method, the situation is made worse by the proposed method.
Output quality suffers in certain scenarios. The issue can easily be demonstrated by making a single movement and comparing the output from each of the encoders. Due to the settings chosen, the encoder outputs files which do not yield original input files when decompressed. In this scenario, the movement leaves a trail of noise which dissipates over time when using the original implementation, but not while using the motion vectors given from the OpenGL-application. The encoder’s motion estimation searches for motion vectors even while nothing is moving, finding “incorrect” motion vectors, but have lower SADs and thus they match the block better and eliminate the noise, see section 2.2.7. If the search range is decreased, the trail takes longer to dissipate as seen in figure 5.1. This effect is not as prevalent in ValeDecem because it is a more natural-looking scene compared to TutorialDemo.

![Figure 5.1: Comparison of noise with different search ranges after a movement towards the upper left corner. Samples taken from tutorialdemo.](image)

### 5.2.4 File size

The new output files are smaller than the ones generated by the original implementation. The effect was more prevalent on tutorial-demo (as seen in 5.5).
5.3 First x264-prototype

The second prototype uses the x264 source code. The purpose of this prototype was to provide a working implementation within a well-established and advanced encoder. A success in this scenario would provide possible direct improvements to the encoder when encoding a video game.
Please note that output quality is not a factor for this test as the settings used in this case give pixel-perfect quality.

5.3.1 Status of the prototype

A few issues became apparent:

1. The x264-encoder uses different kinds of frame-encodings depending on what it thinks is optimal – for instance: frames referencing both future and past frames called B-frames. Due to the scheme implemented in this thesis only reference frames and frames referencing previous frames can be used.

2. The motion vector range is limited, thus, the input vectors have been suitably constrained as well.

5.3.2 Performance improvement

There was a slight performance improvement for ValeDecem and a performance-drop for tutorial-demo. This means that the proposed method functioned better for the natural-looking scene.

5.3.3 File size

The output files were three times as large as the one the original encoder made for both demo scenes. This is a sign of the proposed method finding sub-optimal motion vectors compared to the ones the original encoder finds. If setting the encoder to be lossy, this would most likely cause a huge drop in quality. This is an indication of the proposed method functioning worse than the original implementation in terms of optimizing for output size.

5.4 Improved x264-prototype

The third prototype is enabled through a boolean in the second prototype. In other words, the second and third prototypes share code-bases. The purpose of this implementation is to attempt to improve on the results from section 5.3.
Please note that output quality is not a factor for this test as the settings used in this case give pixel-perfect quality.
5.4.1 Status of the prototype

The following additional changes were done within the improved x264-prototype:

1. Instead of preventing motion search, the motion estimation is provided with another starting point suggestion. The encoder then proceeds with the standard motion search when given the –mvf-hint parameter.

2. This addition also required some changes elsewhere in the code to prevent some errors.

The idea behind this modification is the fact that while you may know exactly how things are moving, you are not able to compensate for earlier mentioned distortion caused by moving and rotating objects in 3D space. The original x264-prototype is in the same code-base (run without the -mvf-hint parameter).

5.4.2 Performance improvement

Performance-wise, this runs approximately as fast as the original implementation of the encoder, sometimes somewhat slower and sometimes somewhat faster.

5.4.3 File size

The output-files are approximately the same size as the ones the original encoder outputs a few bytes smaller or larger for both demo scenes. This means that the addition actually has a chance of reducing the file size in certain cases. This is an indication that this version of the prototype is performing on par with the original implementation – most likely because it heavily relies on the original implementation of motion estimation.

5.5 Comparisons

This section contains data detailing run-times comparing the original encoders’ performances compared with the encoders modified with the proposed method. The runtime is an average for 5 runs. This does not include running the application which generates the input-files. The outputs from both demo scenes were generated once and then encoded by the encoders in various modes. The file sizes of the input files can be seen in table 5.1.

<table>
<thead>
<tr>
<th>Demo</th>
<th>Video file size</th>
<th>inf5063-c63 mv file size</th>
<th>x264 mv file size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tutorial-demo</td>
<td>211 MB</td>
<td>11 MB</td>
<td>2.8 MB</td>
</tr>
<tr>
<td>Valedecem</td>
<td>673 MB</td>
<td>36 MB</td>
<td>8.8 MB</td>
</tr>
</tbody>
</table>

Table 5.1: Sizes of files used for testing.
The tests were run on the machine described in section 6.4. Other programs were shut down and the tests were run using the linux program “time” on the encoders. Real time was recorded.

<table>
<thead>
<tr>
<th>Demo</th>
<th>Info</th>
<th>inf5063-c63</th>
<th></th>
<th>x264</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Mode</td>
<td>Normal</td>
<td>MV</td>
<td>Normal</td>
<td>MV</td>
</tr>
<tr>
<td>Tutorial-demo</td>
<td>Run time</td>
<td>276.9 s</td>
<td>15.86 s</td>
<td>5.8 s</td>
<td>8.3 s</td>
</tr>
<tr>
<td></td>
<td>File size</td>
<td>9.5 MB</td>
<td>6.6 MB</td>
<td>0.623 MB</td>
<td>1.7 MB</td>
</tr>
<tr>
<td>ValeDecem</td>
<td>Run time</td>
<td>901 s</td>
<td>52.75 s</td>
<td>26.5 s</td>
<td>23.7 s</td>
</tr>
<tr>
<td></td>
<td>File size</td>
<td>35 MB</td>
<td>35 MB</td>
<td>3.8 MB</td>
<td>13 MB</td>
</tr>
</tbody>
</table>

Normal – The original implementation
MV – Normal motion vector implementation (detailed in sections 5.2 and 5.3)
MV+ – Motion vectors denote where the searches for optimal motion vectors should start (detailed in section 5.4)

Table 5.2: Table showing file sizes and run-times for encoders given different parameters and input files.
Chapter 6

Conclusions

Having implemented and tested the proposed method, I reached a few conclusions and ideas for future work.

6.1 Overall interpretation of the results

The goal of the proposed method was to speed up video encoding of video output from games for use in cloud gaming. This was done through limiting the scope to the critical component – the video encoder – and modifying it to accept information from the application which was also generating the video-output being encoded. In this case, the original applications (games) were substituted with demo scenes to simplify generation of the data.

By evaluating the results from chapter 5 in regards to the goals set in section 1.3, I have reached the following conclusions about each of the tested prototypes:

• Section 5.2: the results from the initial prototype were somewhat promising. A great increase in performance and smaller file sizes were observed at the expense of the goal that failed. The specific goal failed was that the proposed method should not detract from the quality of experience of the game. Table 5.1 shows problems in regards to noise when encoding certain kind of scenes.

• Section 5.3: the results from the first x264-prototype are somewhat promising. There’s an increase in performance at the expense of a larger output size.

• Section 5.4: the last x264 prototype held some promise, but ultimately the gain was too slim. Performance and the output stream size remained mostly unaffected.

This is not to say that the proposed method is not applicable to anything. As a proof of concept, the proposed method works. The proposed method does not detract anything from the game itself and it can be implemented by a game developer in their own games as a way to make the games
available on low-end platforms through streaming.

One thing to note about the last x264-prototype is that it runs approximately as fast as the original implementation while also reading an additional file. On one side, this means that the process might be running faster despite having to read an additional file. This could imply that this version would run faster if access to all data was faster, such as reading from memory (which would be the case in a real streaming-program). However: one could also argue that the program creating the output also spends more time on this process as it has more output-data to create and write.

If the proposed method is to be used within any piece of software, I would suggest heavily modifying an encoder or building one from scratch to match the needs of the application. The x264 encoder does not rely heavily on papers to find venues of optimizations, but rather through the failures and successes of experimentation. This approach is most likely necessary for this proposed method to work efficiently.

Also, the table 5.1 shows that the modified inf5063-c63 encoder is approaching the x264-encoder performance-wise. This was in a way an unexpected development due to the fact that the x264-encoder is much more optimized in other areas.

In short, initial tests show that using the proposed method is possible, but it needs more work.

### 6.2 Further work

I would also like to suggest additional untested methods. The code complexity of these methods are higher than the proposed method, but may yield better results.

#### 6.2.1 Macroblock size prediction

I would like to suggest a new variant of subpel prediction specifically for encoding games. By extending the proposed method through adding more output-data, it is possible to calculate movement in the z direction (forward/backwards relative to the camera). This could give further possibilities by scaling the matching size of a macro-block in an encoder and using the standard subpel-technique with different weights to create virtual pixels.

If an object is moving away from the camera, the matching macroblock in that area should be scaled down (see figure 6.2.1). If the object is moving towards the camera, the matching macroblock in that area should be scaled up. A simple calculation of $MVP_{new} \times pos_{new.z} / MVP_{old} \times pos_{old.z}$ yields
Chapter 6. Conclusions

6.2. Further work

While creating the data isn’t hard – slight modifications to the demo source code would’ve made this possible, modifying an encoder to accept this data is a challenge.

![Figure 6.1: Macroblock scaling.](image)

Black: pixels in the previous frame
Red: virtual pixels for an object moving towards the camera.
Green: virtual pixels for an object moving away from the camera.

Furthermore, this proposed method could be extended by having the macroblocks approximate the shape of the geometry which enters the block. This would also remove the requirement for calculating movement towards or away from the camera as one could instead calculate the previous position of every corner of a macroblock and linearly interpolate the rest. See figure 6.2.1. Please note that the matching shape (green) does not have to be rectangle.

![Figure 6.2: Macroblock geometry prediction.](image)

Red: the macroblock being matched in the current frame
Green: predicted shape and position of the geometry in the previous frame, there is a one-to-one mapping between these points and the points in the macroblock.
Yellow: Corner points of a macro block
6.2.2 Vertex tracking through mesh and vertex IDs

Another alternative is tracking polygons based on the mesh they’re a part of and their vertex ID. This method possesses both weaknesses and strengths compared to the one used in this thesis.

For every mesh rendered, the program must keep track of a uniform variable per object rendered in a frame as you can’t trust the automatic meshIDs given by OpenGL as they rely on render order — if an object leaves or enters the scene or the render order changes, the meshIDs given may no longer match.

The output in this case is not a map of motion vectors, but a map of which vertex is being rendered at a given position. Calculating the motion vectors given this map is a post-render calculation which utilizes the CPU.

While this method may provide additional capabilities such as detection of new objects or the skybox, it is still far more complex than the method tested in this thesis and thus harder to implement. Also, if a new object appears, setting a (0,0) motion vector for it is not an unreasonable solution. If the appearing object distorts the macroblock too much, the encoder may choose to encode the entire block from the ground up instead of encoding the difference.

In addition, all of the problems detailed in section 4.4 also apply to this technique as it is similar.

6.2.3 An idea to increase responsiveness when playing games in the cloud

I would like to suggest one feature to increase responsiveness in games where cursor-manipulation is a central part of the game mechanics.

Render the scene server-side without a mouse cursor, transfer the cursor to the client and render it on top of the rest of the game scene on the client-side. This could be applicable in many turn-based strategy games, point-and-click adventures and perhaps even real-time strategy games.

Additionally, I would like to suggest an even more advanced approach. Server-side: render (and encode) all 3D scenes and run the game engine. Client-side: receive encoded 3D scenes from server and render GUI and mouse cursor on top of them. This may increase perceived responsiveness in games, however: the code complexity increases. As a side-effect, this solves the issue described in section 4.4.6 as there are no GUI-components on top of the 3D-render on the server-side.
6.2.4 3D objects encoding

Another alternative would be to encode the stream as actual 3D objects with vertices, indices and textures (with mipmapping) transferred to the client for rendering. Perhaps only objects which are close up should be transferred in this manner.

The coding complexity of this approach is immense. It would require handling two different data streams on top of each other and it would require compression of the data as transferring an entire object – with vertex data, indice data and texture data is too much to handle with current Internet speeds. The advantage of this approach is bandwidth-conservation once most of the data is transferred. The disadvantage would be the fact that some data may be generated when objects are moving (this casts shadows, thus altering the “texture” of the object) or the scenery is changing.

This approach seems too complex and inefficient to be viable. Forwarding an OpenGL program through the X window manager to another machine does something similar to this. However, it directly executes the OpenGL-calls on the machine receiving the output instead of rendering it locally.

6.2.5 Other suggestions

I would like to add some other suggestions as well:

- Adding cartoon video encoders – since some games have very “clean” aesthetics similar to many cartoons (such as AntiChamber, seen in figure 4.7), adding cartoon video encoders for the purpose of encoding video games should be considered.

- Let the application creating the output control encoder modes. If a cartoon video encoder is implemented, it would be a useful feature if the application rendering could make the video encoder switch modes and start encoding parts of the screen using other techniques. Realistic-looking games could benefit as well: GUI-components could be encoded using the cartoon-encoders as such elements often have uniform colours and high-contrast borders between them.

- Allow the encoder to control parts of the encoding-process – this could be used to make the video encoder either fall back on motion search or re-encode certain blocks which could be useful when recording effects such as water or when an object enters the scene or pops into existence.

6.3 Other observations

While working on this thesis, I made certain observations which I think are worth sharing.
6.3. Other observations

6.3.1 OpenGL demo scene availability

While there are plenty of encoders with complete source code available for download on the Internet, there’s an acute lack of open-source demo scenes. While finding tutorials is simple, finding a complete demo scene like ValeDecem is actually a small undertaking. I incidentally came across ValeDecem through footage on YouTube with links to the source code after searching for a week through various search engines. Demo scenes written in OpenGL are not rare. Demo scenes freely available with complete source code are quite scarce.

I would urge anyone reading this: if you are in possession of a somewhat nice-looking demo scene in modern OpenGL or DirectX, please publish it on the Internet as it may make demonstration of research easier.

6.3.2 OpenGL versions in open source software

OpenGL suffers somewhat from the fact that it is harder to write code which works for multiple versions due to deprecation of components through replacement (see section 2.3.6). Because of this, a lot of open source games utilize OpenGL 2 which works for a wider range of graphics cards, but does not utilize some of the capabilities of newer OpenGL versions.

6.3.3 YUV 4:2:0 vs Y’CbCr 4:2:0

There is some confusion regarding YUV and Y’CbCr. These colour spaces and encodings are very similar, but not equivalent. Please observe figure 6.3.3. The file formats YUV 4:2:0 and Y’CbCr 4:2:0 are serialized in the same way, for each frame: Y-channel data followed by quarter-resolution U or Cb-channel data followed by quarter-resolution V or Cr-channel data.

Figure 6.3: For $Y = 0.5$ and $Y’ = 0.5$, left: Y’CbCr, right: YUV.

No direct issues (except for possible sub-par encoding) arise within the encoding process as long as the data-format is presumed to be the same before and after the decoding. An encoder that is capable of compressing
YUV 4:2:0 files is also capable of compressing Y’CbCr 4:2:0 files. However, this requires the decoder and the application that generates the output which the encoder reads to have agreed upon one standard. If you encode YUV 4:2:0 footage and read it as Y’CbCr 4:2:0 footage (or vice versa), some discolouring occurs.

6.3.4 Development notes

The complexity of implementing these kind of algorithms does not arise from the need to understand single systems, but combining multiple systems and the translations between them (as seen in chapter 3). While translating between these systems, it is easy to make mistakes which are hard to detect.

I would therefore like to provide a list of mistakes and general hints. Some of the mistakes happened during development.

- Motion vector scaling relative to channel size. In Y’CbCr 4:2:0, the Cb and Cr-channels have less data per pixel, thus, a motion vector of (4, 2) would only traverse (2, 1) units in the Cb and Cr-channels. To test if your implementation is correct: dump the prediction buffer and see if you get discolouring. There is a switch for this in the inf5063-c63encoder, search c63dec.cpp for #if 1. Dumping the prediction buffer can also help you determine if you are reading the motion vectors correctly.

- Do not assume that the implementation you’ve pulled from the internet is without bugs. See last part of section 4.2.2.

- When creating or modifying an OpenGL-program, make sure to use something akin to this function almost everywhere in debug-mode:

```cpp
void checkError(std::string msg) {
    GLenum error = glGetError();
    if (error != GL_NO_ERROR) {
        std::cout << msg << "::" << glewGetErrorString(error) << std::endl;
    }
}
```

This may help you pinpoint the command causing the error. However, the source of an error may still be elsewhere.

- When porting from C to C++, new is not a replacement for calloc. Remember to zero out your buffer.

6.4 Testing environment

This section details important parts of the testing environment used in chapter 5. It is important to provide this information as the readers of this
text may encounter issues when running the code provided in this thesis on the basis of outdated hardware, misconfiguration or other issues.

6.4.1 Hardware

The code was tested on the following machine:

- RAM: 2x1 GB RAM, 800 MHz, DDR2
- CPU: Intel Core 2 Duo E6750 @ 2.66 GHz
- Graphics card: GeForce GTS 450 (1024 MB memory)
- Hard drive: Seagate Barracuda ST3320620AS, 7200 RPM, 320 GB, 16 MB buffer

At the moment of writing (July 2014), this is somewhat dated hardware except for the graphics card which is almost four years old.

6.4.2 Software

- Linux Mint 16 64-bit
- Nvidia Driver version 319.32
- GCC compiler version 4.8.1
Bibliography


Netindex. Netindex.com showing measured average speeds of approximately 20 Mbit/s globally by Ookla, the host of speedtest.net - a website


Preben N. Olsen. Codec63 demo, September 2013. URL http://www.uio.no/studier/emner/matnat/ifi/INF5063/h13/Resources/inf5063-pra-c63.pdf. These are slides describing some parts of the inf5063-c63 encoder.


