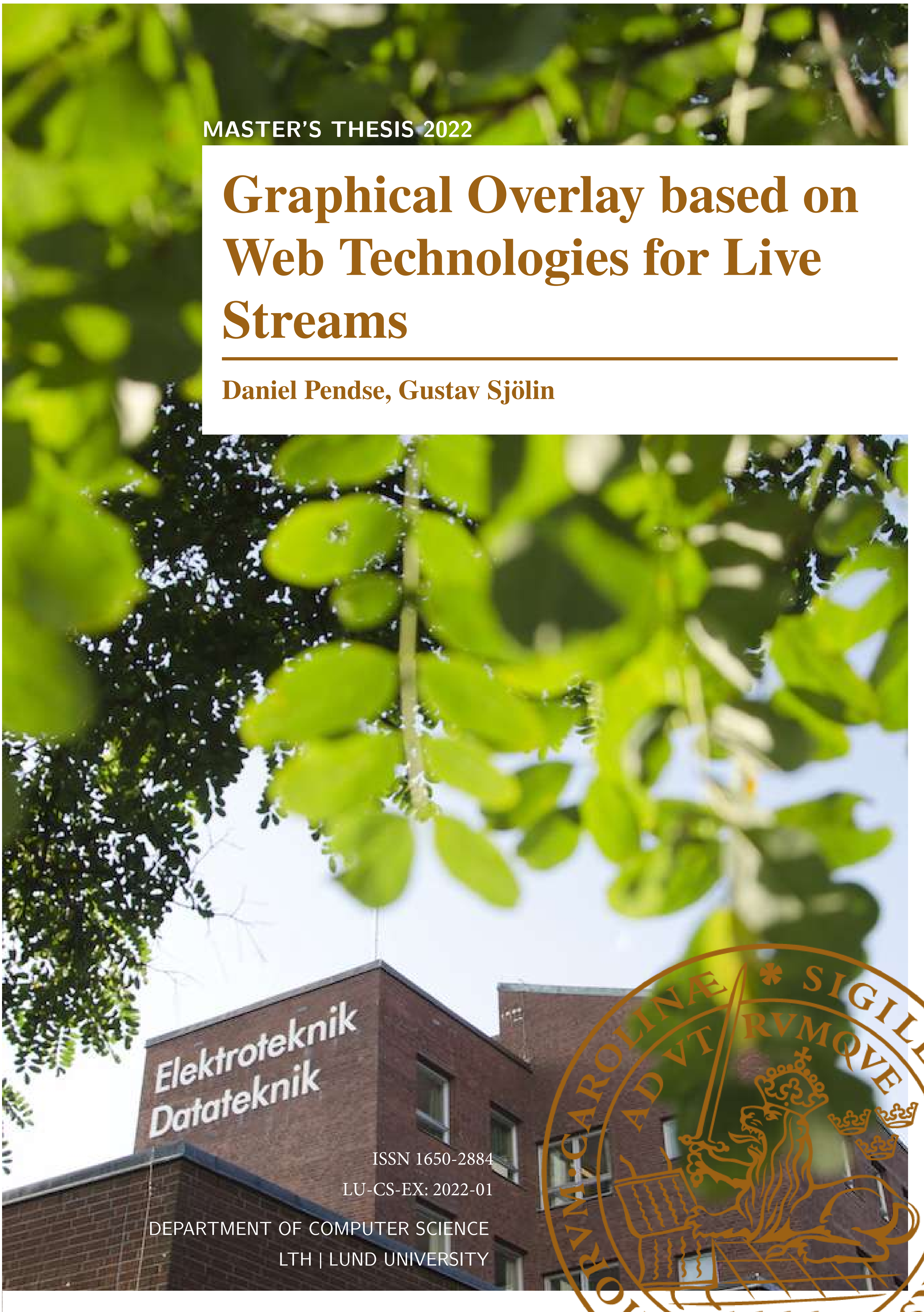


MASTER'S THESIS 2022

Graphical Overlay based on Web Technologies for Live Streams

Daniel Pendse, Gustav Sjölin

ISSN 1650-2884
LU-CS-EX: 2022-01

DEPARTMENT OF COMPUTER SCIENCE
LTH | LUND UNIVERSITY

EXAMENSARBETE
Datavetenskap

LU-CS-EX: 2022-01

Graphical Overlay based on Web Technologies for Live Streams

Grafisk Overlay Baserad på Webteknologier för live stream

Daniel Pendse, Gustav Sjölin

Graphical Overlay based on Web Technologies for Live Streams

Daniel Pendse
ast15dpe@student.lu.se

Gustav Sjölin
kem13gsj@student.lu.se

January 6, 2022

Master's thesis work carried out at Spiideo AB.

Supervisors: Erik Zivkovic, erik.zivkovic@spiideo.com
Michael Doggett, michael.doggett@cs.lth.se

Examiner: Flavius Gruian, Flavius.Gruian@cs.lth.se

Abstract

In order to enhance the engagement and viewing experience of live sports broadcasting, it is essential to engage the audience with informative graphics and interaction. Live sport streaming and analyzing company Spiideo provides an automatic live broadcasting service, Spiideo Play. The current solution for creating a graphical overlay is hard to customize and lacks the ability to generate animations.

The goal of this project was to create a feature rich and fast graphical overlay based on web technologies and compare to the current solution in aspect of speed, cost, customizability and features.

A solution was created as a hardware accelerated FFmpeg filter. When comparing the solutions the new solution had a shorter total processing time and was more cost-efficient. It was also far more customizable and capable of more features.

Keywords: MSc, Graphics, Overlay, Live stream, FFmpeg

Acknowledgements

We would like to express our gratitude towards supervisor Michael Doggett, associate professor in the graphics group, for support and help throughout the project.

We would also like to thank Spiideo and our supervisor Erik Zivkovic. Thank you Erik for all your time, engagement and expertise in the project and area. You have helped us and given us support throughout the project which we are grateful for. We are also grateful to Carl Larsson at Spiideo for this opportunity and given us all necessary resources.

Thank you!

Daniel and Gustav, Lund 2021.

Contents

Definitions

In this section commonly used concepts and abbreviations are explained

- **Frontend** - Everything the user sees on a website or in a software program.
- **Backend** - Part of the website or software program which the user does not see. Ex accessing data, processing request and uploading/downloading files.
- **GPU** - Graphics Processing Unit.
- **CPU** - Central Processing Unit.
- **SVG** - Scalable Vector Graphics.
- **Graphical Overlay** - A frame displaying graphics in a layer over a frame.
- **Rendering** - Calculations of the pixel values in a picture. Also commonly associated with the actual creation of the picture.
- **Headless rendering** - When the instance does not have a screen to generate the calculated picture to.
- **FFmpeg** - A multimedia tool.
- **AWS** - Amazon Web Services which is a service where it is possible to rent computers and other services in the cloud.
- **AWS EC2** - AWS Elastic Compute Cloud provides scalable computing capacity in the cloud.
- **AWS Lambda** - AWS Lambda is a event-based serverless compute service that runs code on events and automatically manages the machines to run the code.
- **WOS** - Web Overlay Solution.
- **SWOS** - Standard Web Overlay Solution.

- **GAWOS** - GPU Accelerated Web Overlay Solution.
- **WOG** - Web Overlay Generator.

Chapter 1
Introduction

In order to enhance the engagement and viewing experience of live sports broadcasting the stream needs to be engaging with info-graphics and interaction. As the viewer demand to get real time data and statistics about the game, and as the different sports clubs and customers to the streaming services desire a high customization, a fast and dynamic graphical overlay solution is essential. This thesis aims to explore a new feature rich graphic overlay that is rendered to the live stream in the backend, by using common web technologies. This thesis is implemented towards the case company Spiideo, which is a Malmö-based company that provides an automatic live broadcasting service for sport clubs, Spiideo Play.

1.1 Overview

The thesis will be implemented towards the case company's setup but the main focus is on how to implement a web based overlay to live streams. The case company currently has an overlay to the live stream displaying game information. It is currently based on scalable vector graphics (SVG). The overlay displays a scoreboard featuring real-time events of the game's time, score, and other relevant information. An example of a scoreboard is seen in Figure 1.1. Their SVG solution has limits on customization's, a feature which the market desires, and thus they want to examine other options. Instead of using a SVG solution, this

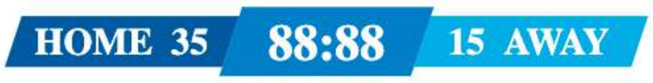


Figure 1.1: SVG Scoreboard

thesis will study an alternative overlay based on common web technologies such as HTML, CSS and JavaScript that can be rendered efficiently with a input video stream. The idea is to render the web resources as an overlay and burn the overlay in to the video source file. Using this approach, only the videos data bytes are sent for distribution allowing the video to be streamed on any device that has internet connection such as smartphones, computers and smart TVs. The general system setup for the case company is seen in Figure 1.2 which roughly illustrates the data flow and the main idea of how the overlay will be inserted in to the video. The case company uses a segmentation of the video, and the video is sent through different segments to the server. The idea is then to integrate with a database to fetch game data which is overlayed using web technologies before being sent to distribution.

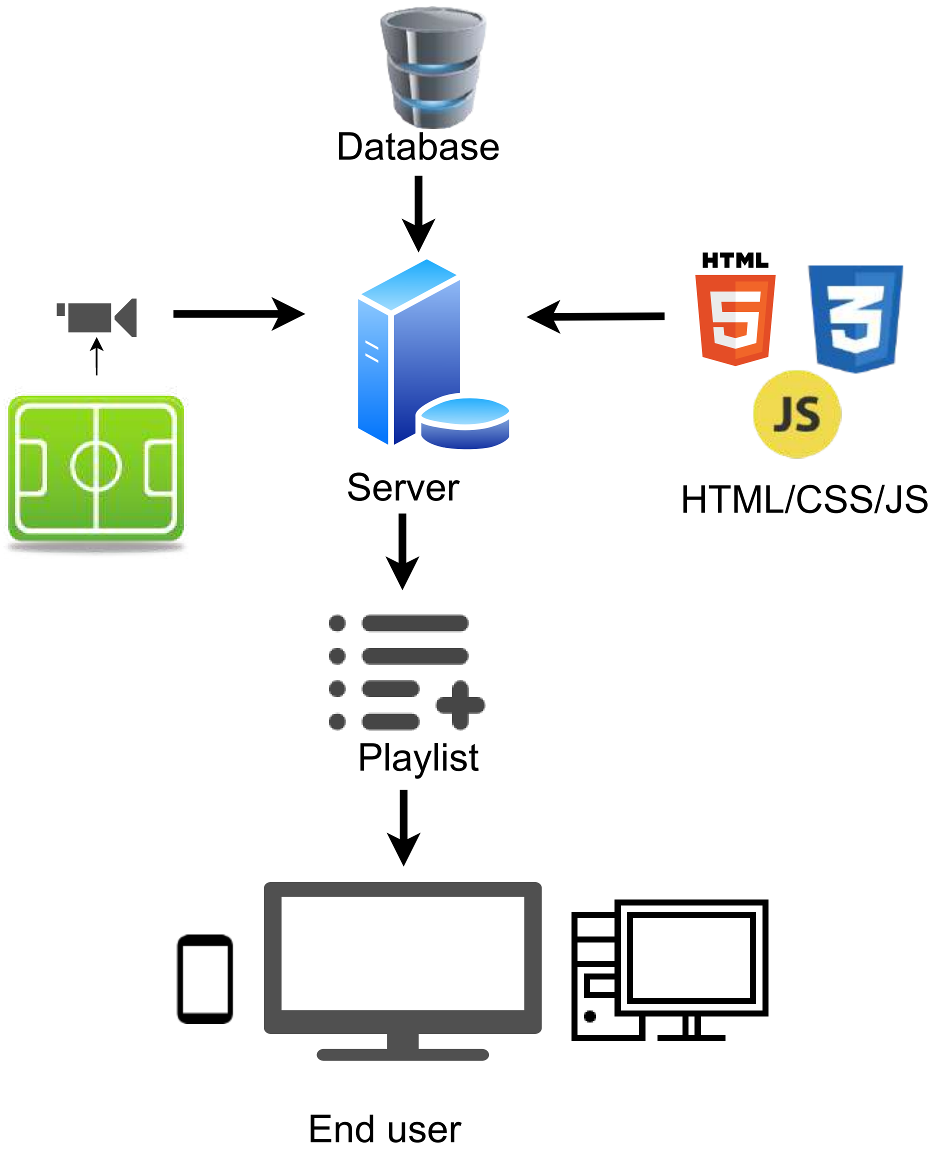


Figure 1.2: System setup

1.2 Problem Definition

The goal of this thesis is to investigate and implement a web-based overlay on a live-stream and the use of GPU for rendering. This is to reduce the latency time by improving the rendering speed and to achieve higher customization on overlays. The web based overlay will benchmark towards the case company's current SVG overlay.

1.3 Research Questions

This thesis aims to explore new features for graphical overlays and the use of the GPU for rendering. The thesis will therefore investigate if the web based overlay has more features compared to the current SVG overlay according to research question 1 (RQ1). Another vision for this thesis is to create a high customization on the overlay. This question is investigated according to RQ2.

RQ1 Has the web based overlay more features compared to the current SVG overlay?

RQ2 How can a customization be accomplished using a web based overlay?

In context of live streaming an important aspect is to reduce the latency time for the viewing experience. Which is why it is important to reduce the rendering time for creating the overlay to the video. When GPU accelerated it is likely that the rendering and process times is faster but it might also be a more expensive solution in terms of cost. That is a comparison this thesis aims to explore with RQ3 and RQ4.

RQ3 What is the difference in rendering time for the overlay solutions?

RQ4 Which of the overlay solutions is the most cost efficient?

1.4 Contributions

The outcome of this thesis aims to create a new efficient and customizable overlay based on the use of web technologies in the back-end allowing for the overlay to be stitched together with the input frame. This thesis contributes with insight on integrating web rendering technologies to produce HTML overlays to video output. Furthermore this thesis explores how to GPU accelerate the rendering of a video with a overlay using FFmpeg and also proposes how a overlay can be synchronized with the input stream.

1.5 Related Work

There are several solutions for creating an overlay on live streams. Some existing companies that are creating overlays for live sports are Singular.live and LIGR. Singular.live is one of the world's most advanced digital overlay platform and the overlays are generated using a cloud-based Control Application which outputs a HTML layer that sits on top of the video [37] [1]. This means that the rendering of the overlay is done on the viewers device and not burnt into the video. The same approach for creating overlays are also used by LIGR.

There are also company's working with creating overlays for the gaming industry like strexm. Strexm is a web based overlay service where the overlays are crafted using HTML and CSS [40]. Unfortunately none of these solutions are open sourced or have publications about the technologies underlying the software.

The techniques used in the previously mentioned softwares add overlays that will be generated on the client side. There are no articles or softwares found that imprints web based overlays in the backend directly on the source video.

A free open source software for video recording and live streaming is Open Broadcast Software (OBS) Studio [31]. When working in OBS Studios it is possible to add overlays, for example as external HTML files or URLs to a live stream. The process is the same as for the other mentioned products where the overlay is generated on the client side.

Creating a live overlay from web technologies such as HTML/CSS/JS is a current topic for the moment. No later than 6th of April 2021 did Amazon Web Solutions announce that their system now supports HTML 5 motion graphics overlays for their MediaLive service [6]. The reason for this update comes from that creating HTML based overlays has grown in popularity [44]. The rise in popularity is argued to be due to the fact that the technique is based of standard web technologies which eliminates the use of proprietary authoring or rendering tools for the content creators.

1.6 Division of work

The work has been distributed equally between the two authors. Whereas some parts have been done in pair and some parts have been divided by the two authors. The individual contributions of this thesis, both in implementation and the writing of the report, is presented chapter by chapter as:

- **Introduction** - This was done in pair but Daniel did most of the writing.
- **Background** - Gustav investigated CPU and GPU rendering and the FFmpeg overview. Daniel investigated the live sport streaming section, web rendering overview and graphics programming. The section on system overview and SVG overlay was done in pair.
- **Approach** - The method was determined in pair and also the screening process of choosing web rendering framework. The implementation of the overlay was done iterative in pairs. The design of the scoreboard was done by Daniel and the animation implementation by Gustav.
- **Evaluation** - The overlay features where presented by Daniel and Gustav performed the comparisons of the overlay solutions. Both extracted data for the measurements

on a remote server. Daniel did the discussion section.

- **Conclusion** - This part was done by Gustav.

1.7 Outline

Firstly a section about live streaming is presented in the background chapter 2 to give further context to live streaming and the important aspects requested by the market. In the background chapter there is also an overview of the case company's current SVG solution and how it operates. This of value since the web based overlay needs to be benchmarked towards it. This is also followed by a background of the different technologies used in this thesis. That is mainly an overview of different rendering techniques, what GPU accelerated rendering means compared to CPU rendering, rendering of web resources and the use of the multimedia tool FFmpeg.

The approach to solve the project is described in chapter 3 along with the methodology and implementation. The evaluation and discussion of the result is presented in chapter 4 and the conclusion is presented in chapter 5.

Chapter 2
Background

In this section theoretical information is explained with the interest to understand the background of the thesis and further motivation of the research questions. Firstly this section aims to give background about live video streaming and the importance of latency and the rising demands for customization. It will also be further explained how the case companys's current SVG overlay operates. These parts gives further context to the thesis and are necessary for understanding the motivation of the thesis and the evaluation towards the current SVG overlay solution. Necessary background information of the different rendering techniques is also presented in order to understand the method and implementation of the web based graphical overlay. The necessary information about the multimedia tool FFmpeg is lastly presented for the purpose of understanding how the video processing is handled in the thesis.

2.1 Live Video Streaming

Video streaming has risen in use significantly across the last two decades, especially in the modern workflow and entertainment industry [15]. Although there have been technological improvements in the video streaming area, there are big challenges for video streaming companies. In particular, there are bandwidth limitations, latency limitations and device compatibility challenges that restrain the viewers from receiving picture perfect and seamless video. Furthermore with innovations rising across devices, with ability to display 4k resolution, augmented reality and 360 videos, it has become a consumer demand for high quality video with low latency [15].

2.1.1 Latency

For live video streaming the challenge is to minimize loss in encoding, transport and decoding with as low latency possible. The end-to-end latency also known as glass-to-glass latency denotes the time for a captured frame by the camera glass lens to reach the glass screen of the end user [29]. Achieving low latency is of importance for the viewing experience and especially in the live sports context, as the users wants to receive the results as fast as possible and not have them revealed by other sources. Whereas a high latency in a live video context can destroy the viewing experience and engagement [29]. The latency number ranges between 3-12 for traditional broadcasting including satellite-, cable- and IP TV, for Over-The-Top (OTT) streaming the latency range between 30-45 seconds [36]. What is considered to be low latency is under 5 seconds glass-to-glass latency [29]. One approach to reduce the glass-to-glass latency without compromising picture quality is by choosing the right encoder and decoder and the right transportation protocol [29]. For this project the key to reduce latency time for the overlay, is to render the overlay to the video as fast as possible. The general video flow and the latency impact is illustrated in Figure 2.1 [29] where the camera source file is first encoded and then transported to the network for processing. The output is then transported to users devices where they are decoded.

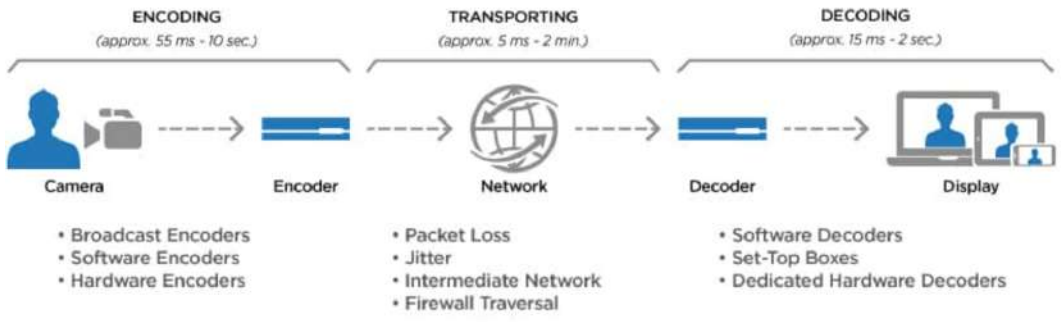


Figure 2.1: Overview of latency in video production workflow [29].

2.1.2 Automated Video Production

The case company's Spiideo Play service is an automated OTT video production which is fully cloudbased. The only installation required is the camera on the arenas and then the video content can be accessed trough any device. The owner of the play product can distribute its content and monetize on the streaming [20]. Traditional live sports productions and paid TV-subscriptions are faced with challenges by other OTT and streaming solutions. The market shares for streaming media has increased significantly with market driving services like Amazon Prime, Netflix and Hulu. Also as the demand for OTT services has overtaken pay-TV subscriptions for the first time in the US in the pandemic year of 2020 [32]. Through a study conducted by PWC, they concluded that live sports is the major reason why some consumers are still stuck with pay-TV subscriptions [35] [38]. From the study, 82 percent would stop paying for the subscriptions if they didn't consume the sports content [20]. Moreover as the demand increases for OTT services, where constant access is given to the content through any device, the same report focuses on the future of live sports streaming as the last mile for large OTT streaming platforms to have in their portfolio [33] [16]. Additionally the demand for OTT services by sports clubs is increasing. A third of the top 25 football clubs, and six out of the top ten largest leagues now offer OTT services where they can monetize on their content [16].

By using OTT live streaming for sports, viewers can access the content from around the world on any platform. And the technology is well suited for the market as users request a high level of entertainment and fan engagement, and the content distributors request a high level of customization [16] [2]. Another driving factor for live sports is the desire to increase the engagement with interactive graphics. OTT is predicted to drive Augmented Reality (AR) and Virtual Reality (VR) innovations [2]. As the technology can package the entertainment experience, e.g. as the viewer is sitting at the stadium watching live on site, and at the same time satisfying other streaming demands as 4k resolution [2].

A vision for this thesis from the case company is to provide a high level of customization on the overlay graphics, in order to satisfy request from third party media companies. And to increase the engagement, a overlay with the ability to display seamless info-graphics can be key, and also to drive revenue from ads to allow third parties and clubs to monetize on the content.

2.2 System Overview

A general system overview of case company's Spiideo play is seen in Figure 2.2. The system fetches the camera video source files from the pitch and decodes the inputs. The input source files are firstly decoded and then there is processing of the two segments to produce a rendered output frame. To the output frame an overlay is generated and blended with the frame. The video frames are encoded and uploaded to produce a segment ready for distribution. In this setup it is assumed that there are video segments from two cameras that are processed but in some cases there is one or three cameras, however, the process is similar. The idea is that the overlay is inserted in this pipeline to the computed output frame as seen in the Figure 2.2.

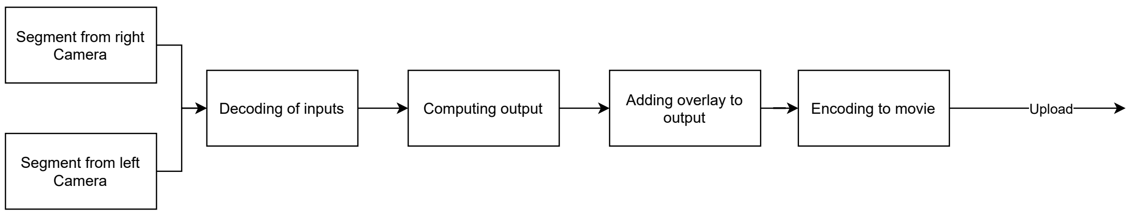


Figure 2.2: System overview of a generic setup.

2.3 SVG Overlay

In this subsection the setup for the case company's current SVG solution is described. This is relevant to understand the evaluation of the developed web based overlay towards the current solution.

2.3.1 SVG Overlay Setup

The system is set up to run on AWS where the setup utilizes both AWS EC2 instances and AWS lambda [4] [7]. A simplified overview of the series of processing steps is seen in figure 2.3. The SVG overlay setup inputs the video source files from the camera but also inputs the SVG overlay which is generated in AWS lambda. The overlay segment is stitched together with the output frame using FFmpeg commands. The overlayed segment is then uploaded for distribution. The downloading of the input video source files, SVG overlay, processing and the uploading is all generated on a AWS EC2 CPU instance. This overview is essential to understand the experimental setup when comparing the rendering time and cost for the solutions.

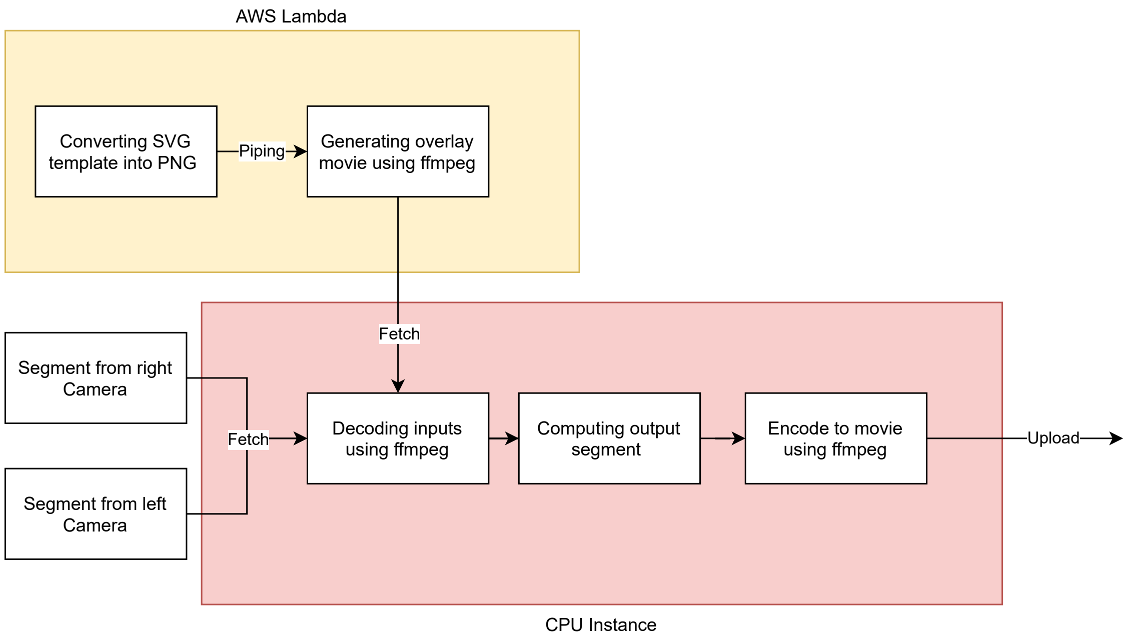
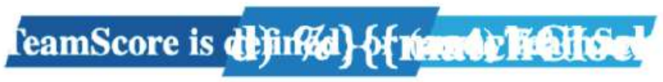
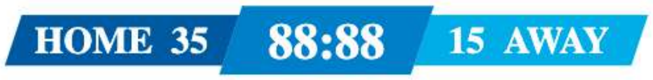

AWS Lambda
Converting SVG template into PNG
Piping
Generating overlay movie using ffmpeg
Fetch
Segment from right Camera
Segment from left Camera
Fetch
Decoding inputs using ffmpeg
Computing output segment
Encode to movie using ffmpeg
Upload
CPU Instance

Figure 2.3: The setup for the SVG Overlay Solution.

SVG Overlay Generation

The generation of the SVG overlay is based on a template engine which enables the use of placeholders to set certain values in the template. The template inputs an SVG file that displays the graphics of the score board as seen in Figure 2.4. The Figure first displays the scoreboard without any inserted variables. It is created using the template engine Tera which is a Rust framework based on jinja and the template language Django [25][22][10]. The game data is fetched for the active game and the variables are then inserted into the placeholders to then be rendered to the template which is seen in the Figure 2.4. The rendered SVG file is converted to a PNG file format before being processed to a video. The overlay feeds data bytes from the PNG files to a FFmpeg pipe, which converts the data bytes into frames and then to a video segment. An FFmpeg pipe is basically a specification on what FFmpeg will consume as standard input, and in this case data bytes from a four channel PNG, to produce a video. The SVG overlay acts as a standalone application and produces overlay video segments on the AWS Lambda and does not input any video segment. Therefore the generation of the overlay segment does not do any decoding.


(a) Tera template without data



(b) Tera template with data

Figure 2.4: SVG template.

2.4 Rendering

Rendering is the process in which an image is generated. When rendering a frame the computer calculates how the frame is supposed to look. Based of a model with given attributes for lighting, texture, geometry, shading and viewport the computer is able to generate on output image. There are two categories when it comes to rendering, Real-time rendering and Pre-rendering [41]. The difference being the speed of the output images. Real-time rendering is used when a fast output of images is needed, typically used in gaming and streaming industry. The stream of the case company is today 25 frames per second and this will be our minimum goal for the thesis. Pre-rendering is used when the output needs to be of highest quality. The computations are long and demanding and speed is not of importance. Whenever rendering is used further in this thesis it is referring to real-time rendering.

2.4.1 Graphics Programming

OpenGL

OpenGL, short for Open Graphics Library, is a low-level cross-platform graphics API that provides a large set of functions to manipulate graphics and images [18]. OpenGL acts as a specification list with functions that are used by the CPU to interact with the GPU. GPU rendering use the GPU's many different cores to quickly process the data. The processing cores on the GPU run small specific programs which are called shaders. Some of the shaders are configurable for programming in the OpenGL Shading Language (GLSL) [24].

The graphics setup for the web based overlay solution will use OpenGL and its thereby necessary to understand the fundamentals of how OpenGL works. The graphics pipeline takes a input as a number of 3D coordinates and transforms these to colored 2D pixels to the screen [24]. This rendering pipeline is a series of processes OpenGL takes when rendering an object. There are usually nine steps in the process where most of them are optional and many programmable which transform data input to fragments and pixels. The different steps of the rendering pipeline are seen in figure 2.5. However, for the purpose of this project only the vertex specification, vertex shader and fragment shader step are examined since only these steps are programmed in the solution.

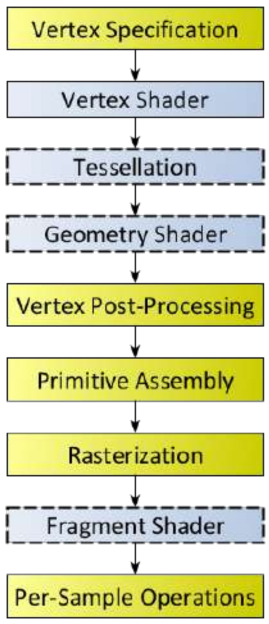


Figure 2.5: OpenGL Rendering Steps [19].

The first big part is building the model using vertex data which specifies coordinates and vertices to the OpenGL context. This is done in the vertex specification. It inputs an array of vertices that describe how the object is visualized in the OpenGL context. There are different kinds of objects to store and define the vertex data. The Vertex Array Object (VAO) and Vertex Buffer Object (VBO) work together. The VAO defines the actual data of each vertex and the VBO stores the actual vertex data. There is also Element Buffer Object (EBO) that can be specified to reuse the same vertices for drawing objects. The vertex specification data is sent to the vertex shader which calculates the coordinates to its display. Further in the rendering pipeline is the fragment shader which is responsible for calculating the color output for each pixel. The visual representation of the graphics pipeline for a simple triangle is demonstrated in Figure 2.6 [23]. Textures are used in this project and can be used to bind image data to an object. This is done by specifying coordinates for the texture which are defined in the vertex specification. In Figure 2.7 a simple polygon is generated and a texture with texture coordinates to fully display on the polygon [21]. By binding pixel data to the texture, pixel transfers from CPU memory are allowed to be uploaded to the GPU memory. The data is finally displayed in the frame buffer [3].

CUDA

CUDA is a parallel computing platform and API model for general computing created by Nvidia corporation to use on their own graphics processing units [30]. CUDA enables more general computing for programmers than OpenGL since OpenGL is mainly

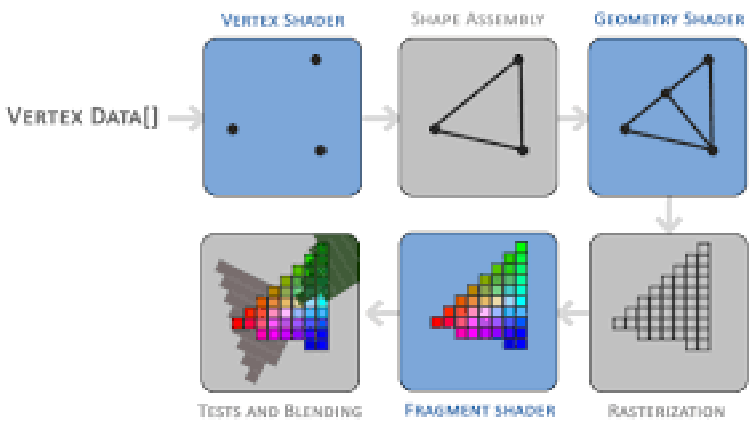


Figure 2.6: OpenGL Vertex Data [23].

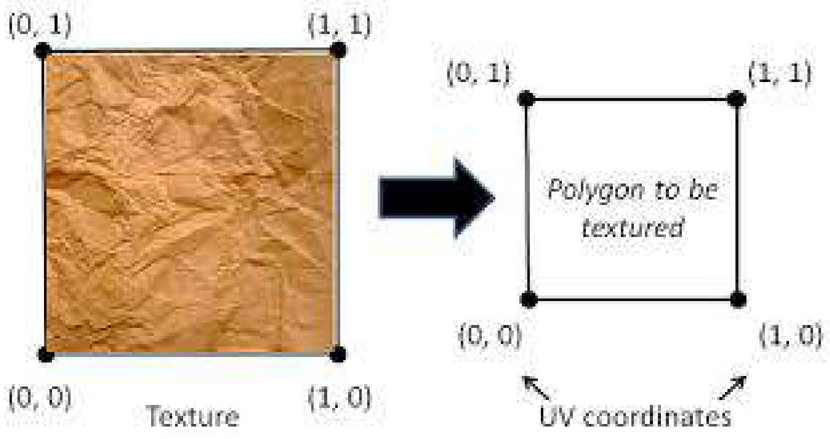


Figure 2.7: OpenGL Textures [21].

for graphics. CUDA is only used for converting frames from different formats in this project and is not a major part of the implementation. CUDA is used in combination with OpenGL because FFmpeg has the ability to hardware accelerate encoding and decoding using CUDA frames [11].

2.4.2 Web-rendering

A web browser is a software program that translates web resources and generates a visual representation. Common web browsers are Google Chrome, Apple's Safari, Mozilla's Firefox, Microsoft's Edge and plenty more [12]. To understand which components of a web browser that is necessary to use in the solution, the components of a web browser is further examined. They all share the same basic high level architecture and the main components of a web browser are [12]:

1. **The user interface**: The whole graphical user interface (GUI) of the web browser. Containing address bars, refresh buttons etc.

2. **The browser engine**: The browser engine is responsible for handling rendering

requests between the GUI and the rendering engine.

3. **The rendering engine**: The main part of the web browser which is responsible for rendering all requests of the web page to the user interface.
4. **Networking**: Responsible for managing all network calls using standard protocols such as HTTP requests.
5. **UI backend**: This part uses the operating system user interface methods and is mainly used for drawing widgets.
6. **JavaScript interpreter**: This part parses and executes the JavaScript code that is embedded in the browser and then forwards interpreted results to the rendering engine.
7. **Data storage/persistence**: A persistent layer needed for the browser to store data locally, for example cookies.

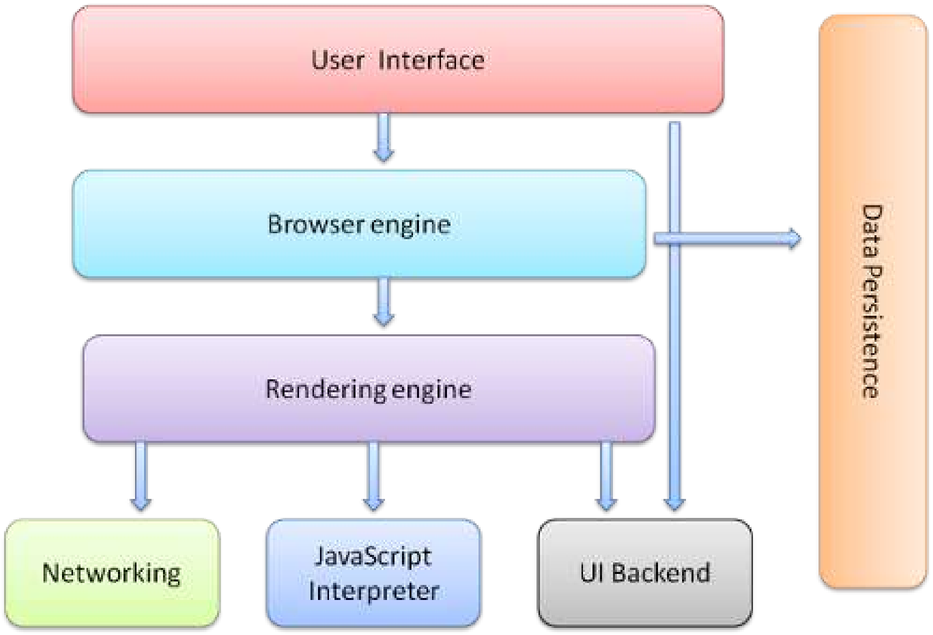


Figure 2.8: High level structure of a web browser [12].

These are also seen in Figure 2.8. It is the rendering engine which is primarily responsible for parsing the HTML and CSS to render-able objects and then also for painting. However there are other wanted components such as networking layer if the software should be able to parse URL links and a JavaScript engine if the software should parse JavaScript calls [39]. Many of these web browsers often share resources such as some have the same JavaScript engine and some have the same rendering engine [12]. There are several different software programs that can be used to render web content, and this clarification helps to narrow the alternatives. For this project a web rendering engine is needed to parse all the web resources and is also required to be embedded in an application. Since the web content will be displayed in a custom environment, it should also be able to render to use in a custom graphics context.

2.5 FFmpeg

FFmpeg is open-source and the leading framework for working with all types of multimedia. The framework is able to decode, encode, transcode, mux, demux, stream, filter and play media. The way to use FFmpeg is with the command line interface, where it is possible to use a broad variation of different commands and filters to accomplish the desired goal. In this case the libraries were used natively to create our own customized filter. The framework has support to run on all major operating systems.

The native libraries have a broad range of API:s to manipulate the video. For instance there are ways to extract a movie frame by frame and get information, such as FPS and pixel format. FFmpeg is also supported for hardware accelerated encoding and decoding [9]. Hardware accelerated encoding and decoding is a way to greatly increase speed due to the use of the GPU to do computations rather than the CPU.

Chapter 3
Approach

The approach for this thesis is very applied and to understand the different choices for the setup, the method section 3.1 will go through the motivation behind the decisions. Thereafter the implementation is explained to further understand the solution.

3.1 Method

The method is to first choose a software that can render the web resources. The rendered content from the software is set up to be displayed in a window system and to be GPU programmable with OpenGL API. The rendered content is then integrated with a custom FFmpeg filter to form an overlay upon an input video stream. The filter is also configured to receive CUDA frames as input source to be hardware accelerated. Furthermore it is implemented to operate in a cloud environment on a headless Linux computer. The overlay is then refined to be synchronized to the live stream and also to perform animations across different 4 second segments, which the input is segmented to. The focus is then to perform different measurements to measure performance when using the GPU for rendering and also to measure the qualitative features of the overlay compared to the current SVG overlay.

3.1.1 Choosing web rendering framework

As mentioned in the background chapter there are several web rendering framework alternatives on the market. The set base requirements is that the web rendering framework needs to be able to render HTML, CSS and JavaScript but it also needs to be embeddedable for the purpose of render the content in the backend an be used in a custom application. Furthermore the framework needs to have capabilities to be hardware accelerated by the GPU to reduce the latency time. The solution will use the the programming language Rust. This is because of several reasons. Apart from that the language is fast, memory safe and have performance advantages, it is primarily because the case company has adopted Rust in their project and it was an opportunity for the authors to learn an exiting programming language [14]. Because of this decision, the framework is also required to have bindings to integrate with Rust. The methodology of selecting the right framework is based on Ulrich and Eppinger's Product Design and Development concept screening method [42]. Several frameworks were investigated and a high level selection was performed to filter out those who did not meet the essential requirements. All frameworks were found by scanning the web for plausible solutions. Based on the requirements for choosing framework, a few frameworks were selected to be further investigated and evaluated against each other in a concept scoring matrix. This allows the frameworks to be objectively compared before continuing with further implementation and the results from the concept screening matrix is seen in Table 3.1. The web rendering framework of choice, based on the screening process, is Ultralight.

Table 3.1: Scoring matrix for the web rendering engines.

	Concept Variants				
Selection Criteria	**Ultralight**	**Web-view**	**Sciter**	**RmlUi**	**Electron**
Documentation	0	-	-	-	0
Ease of use	+	+	+	-	0
Community	+	-	+	+	+
Size	+	0	+	0	-
Speed	+	0	0	0	+
Bindings	+	+	0	-	+
Maintained	+	0	+	0	+
Cost	-	+	-	+	+
Compatability	+	-	0	-	0
Score	6	0	2	-2	4

Ultralight

Ultralight is a fast and lightweight HTML renderer which serves the purpose of the project. It is a cross-platform solution and has the possibility to both render to a pixel buffer by the CPU to then be used in a custom GPU context, and to be accelerated by low-level commands directly to the GPU [43]. Ultralight uses a fork of Safari's WebKit as web rendering engine and pulls changes upstream and the same for the JavaScript engine which is Safari's JavaScriptCore module [43]. Furthermore by studying the example projects, it is relatively easy to use and is implemented as the rendering framework to produce the overlay for the final solution.

3.1.2 Graphics setup

The solution is based on OpenGL and the use of textures. The solution displays to the screen but in an off screen context since it is rendering to frames for a movie output file. The approach to setting up the graphics is that the pixel buffer for the frames on the input video will be rendered to a texture in the fragment shader. This is done by using uniform variables in the fragment shader which allows for pixel data to be attached to the GPU memory and the fragment shader renders each pixel output. Uniform variables are used to declare a variable in the fragment shader and to later assign a texture to the uniform variable. By enabling blending operations in the OpenGL setup, the ultralight overlay also binds to the same texture and is rendered on top of the frame. The output is then captured for the rendered image by extracting its pixel data from the framebuffer.

3.1.3 FFmpeg filter

The approach for designing the FFmpeg filter is to create a filter that extracts the pixel data for each frame of the input video and then does processing. The general process is to upload each frame to the OpenGL context and then apply the overlay as described in the graphics setup 3.1.2. When using this methodology the overlay is synced directly to the input video and logic for calculating the time is retrieved by using the FPS for the input video. For instance, assume a 4 second input video that is processed for 2 seconds with 25 FPS. This will generate a total of $4 * 25 = 100$ frames, where 25 frames corresponds to 1 second. The overlay then uses the number of frames based on the FPS to to generate a internal clock to synchronize the overlay to display in the same timeline as the input video.

3.2 Implementation

The implementation is explained further in this section. Many of the design choices for the setup that affect the thesis outcome are motivated. The goal is to create a customizable, automated graphical overlay and to render it with the video as fast as possible. The implementation for the logic of the overlay and its abilities are presented and the different setups on how to render the video using an FFmpeg hardware accelerated filter.

3.2.1 Overlay Implementation

One of the main goal for the overlay is to allow for customization. This can be done by either allowing user input overlays as public accessible URL address, by providing different theme templates that users can select from or fully personalized overlays developed for each user. The solution needs to thereby handle custom HTML files but also to function with the constraints that the overlay is rendered in a filter with varying rendering speeds and also with the synchronization problem of the overlay and live stream to display in the same timeline. To clarify, when rendering on the CPU, the processing is significantly slower than when rendering GPU accelerated and also these renderings speeds will differentiate by different hardware and iterations. The different rendering times will affect the animation since CSS animations are time based on a master clock in the web rendering engine. Meaning that if a segment is rendered slower than real time the animation will animate much faster than anticipated in the output video. Assume a 4 second segment that contains a animation that should translate a element 100 pixels for 2 seconds and the rendering time for the segment is 8 seconds. Then the desired output would be that the element is translated to move 100 pixels for 2 seconds in the output video which would correspond to that the element is translated for 4 seconds in the rendering step. However, since CSS animations is controlled by its master clock it will render for the set 2 seconds making the output not as expected. Also because of the fact that the frames are rendered as fast as possible leading to different rendering times, causing very uneven animations. The approach for solving this is that the overlay is animated using JavaScript animations and that the animated element is moved any (X) pixels for each frame. And for the synchronization problem the suggested solution is to provide a certain logic in the filter whereas HTML files need to follow a certain specification to use this logic. Because in order to do JavaScript animations the HTML elements that needs to be animated is required to have known ID-tags when they are called upon. There is also an issue in creating seamless animations between the different video segments since the input video is segmented into 4 seconds. Consider an animation that starts 3 seconds into a video segment and should animate for 2 seconds. Then the animation will animate for 1 second to the current input video and should then continue to animate for 1 second on to the next video segment. The different steps of setting up and implementing the overlay with animations and time synchronization is further explained in this section.

Ultralight setup

Since ultralight provides a C/C++ API, a Rust bindings API is used to link towards the C API [34]. The first steps to integrate ultralight is to properly link the libraries and resources provided by ultralight, to the program. Once it is possible to use the ultralight API, necessary default settings are generated. Ultralight is setup to use its own default renderer to generate a view with certain dimensions where the HTML content is rendered. After the basic setup, the ultralight view can either input an external URL address, that is accessible via a public IP address, or HTML files from the set file system. Ultralight can integrate to the OpenGL context in one of two ways [43];

1. **Bitmap API** - Render on the CPU to a pixel buffer
2. **GPUDriver API** - Render on the GPU using low-level commands

The GPUDriver API allows for directly rendering to most graphics API and allow the rendering to be controlled in GPU memory directly as illustrated in Figure 3.1. In the figure Direct3D (D3D) is Windows native graphics API and Metal is macOS native graphics API. However, the current implementation uses the bitmap API and renders on the CPU. The

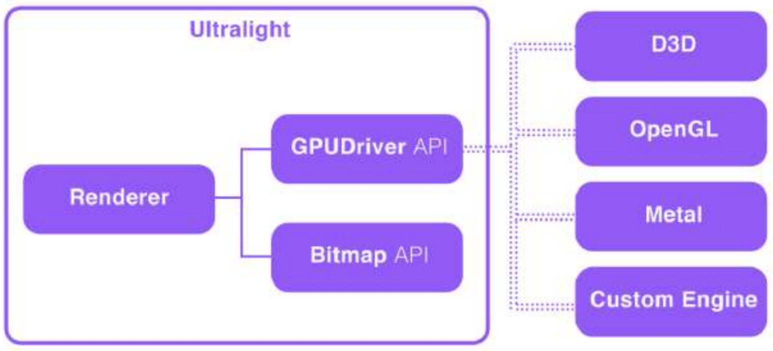


Figure 3.1: Ultralight GPU Rendering [43]

rendered content is extracted by locking the pixel data from the view which return a CPU allocated memory address for the pixel data. The pixel data is then moved to the GPU memory to the custom OpenGL context as it loads the memory address to the texture.

HTML overlay

The graphics of the overlay in this setup is a custom HTML file with CSS styling and a JavaScript. For this solution a simple scoreboard is created to display the game's participants, score and time. An imagined sponsor logo is also generated and both the scoreboard and sponsor logo is animated. The background of the overlay is transparent to be blended to the input frame and the result is seen in figure 3.2

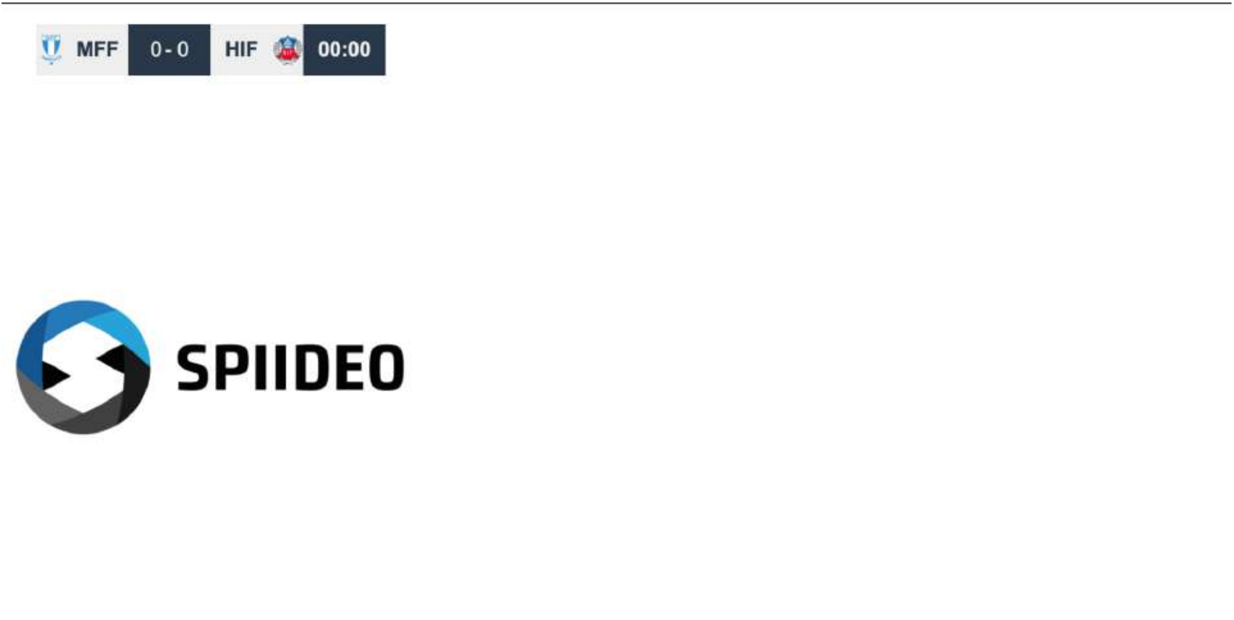


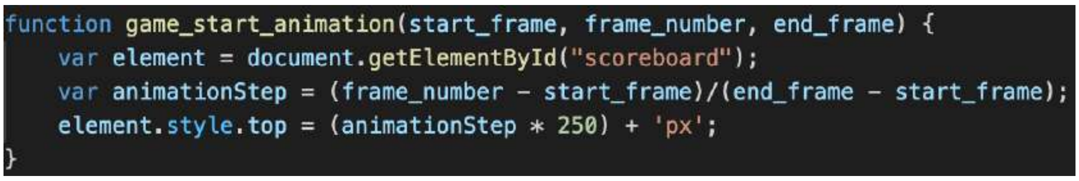
Figure 3.2: Ultralight HTML Overlay

Overlay logic

The overlay is constructed so it reacts on certain events. One can have custom overlays for different use cases but in the case of the case company, all events related to the game is generated by use of API's to extract game events and are located in a database. These events determine the logic if the match clock is running, stopped and if a team has scored and on what time the score took place. All of these events are fetched from a database and controlled if any logic needs to trigger. The events are iterated and if they match any triggers e.g. home-team-score-2, then JavaScript code runs to change the score inside the home-team-score div tag inside the HTML using DOM-manipulation. Because of this logic the HTML file is required to have certain specified div tag names to manipulate its HTML content.

Animations

The same logic is also proposed for an automated animation triggering. E.g. if the game should start then a predefined animation movement is triggered. The animation is constructed so it contains data for what type of animation and its start time respectively end time which corresponds to when the animation should trigger in context to the game's actual time. The start- and end time are extrapolated to corresponding frame numbers for the 4 second segment. If an animation is active between 1-3 seconds then for a 30 FPS input movie it is active between frame number 30-90. And also if the animation should move 250 pixels to a certain direction in a linear speed then the element can be DOM-manipulated to change its position for each frame by using the current frame number. The code block in figure 3.3 shows a JavaScript animation on how to move the scoreboard 250 pixels from the top border in linear speed using DOM-manipulation. The element variable is corresponding to the entire scoreboard div tag and the animationStep is the interpolation of the current frame number in relation to the start- and end frame between values 0-1.



```
function game_start_animation(start_frame, frame_number, end_frame) {
    var element = document.getElementById("scoreboard");
    var animationStep = (frame_number - start_frame)/(end_frame - start_frame);
    element.style.top = (animationStep * 250) + 'px';
}
```

Figure 3.3: JavaScript Animation

To create seamless animation between the 4 second segments a set of internal and global clocks are implemented. Timestamped animations are created from tags generated by events in the game. Then an internal clock compares its timestamp for each frame with the timestamp for the animation. If the timestamps match the animation, it will be executed with a progress value (0-100%). Thus the animations will run independent of the length of the segment.

3.2.2 Graphical Setup

OpenGL

The graphical setup is implemented using OpenGL and the model is created with vertices and indices to instruct which of the vertices are drawn to create two triangles forming a quad to fully cover the window. Texture coordinates are also defined to fully display a texture on the quad.

The vertices are bound to a Vertex Buffer Object and the indices are bound to an Element Buffer Object. These are both wrapped in a Vertex Array Object which stores all the states of the buffers in the GPU memory. The vertex shader gets the vertex data and the texture coordinates and maps them to the output which is received by the fragment shader. The fragment shader is implemented with a uniform texture input variable which enables CPU bitmap data to be passed to the GPU memory. The overlay data and the input frame data is sent to the uniform variable. The fragment shader then calculates the color for all pixels and maps them to the defined texture to be rendered in a off-screen window since we are rendering on a server.

Window System

Before OpenGL can render anything, a library is required to create an OpenGL context and a window for OpenGL to render to. For this solution GLFW [13], an OpenGL library, is used to render to the OpenGL context on local macOS machines for testing. For the headless Linux machine instance in the cloud, EGL, an OpenGL API [17], is used. EGL is used since it removes the need for running a X server for headless rendering [28]. X server is the most popular windowing system for Linux. It manages the graphic displays and user input [45].

3.2.3 Web Overlay Solution (WOS)

The web overlay solution is constructed by creating a custom FFmpeg filter. The solution calls on functions in the Rust Code to receive the input frames. The frames are then converted into textures and used as described in the graphical setup. Also the solution receives the output frame after it is processed in the graphical setup with the overlay.

The solution is able to compile and run towards two run cases. The first is using the CPU for decoding and encoding, this one was named "Standard Web Overlay Solution (SWOS)". The other is using the GPU for decoding and encoding and was named "GPU Accelerated Web Overlay Solution (GAWOS)". The setup to use SWOS is seen in Figure 3.4 where the purple area denotes the SWOS. To use SWOS in production in the case company it should input the projected output segment.

The GAWOS is seen in Figure 3.5. In the figure it also displays how it should be used in production for the case company. The projection, which also is a FFmpeg filter, can be GPU accelerated for decoding and encoding. This outputs CUDA frames before it is encoded and the idea is to insert the GAWOS to directly receive the CUDA frames.

However, the SWOS still needs a GPU because the graphical setup requires the OpenGL context which is configured to be hardware based. There is an option of using the software based OpenGL application by using the Mesa 3D graphics library which software emulates OpenGL and uses a software renderer for calculating the pixel values [27]. Because of time constraints this full software solution was not continued to develop.
To implement this, the solution can compile with two different features; either with CUDA frames or without. And it can also compile against the two different windowing backend systems EGL or GLFW. The latter solution, which processes using CUDA frames, requires an NVIDIA graphics card.

3.2.4 Web Overlay Generator (WOG)

The part of the code that is similar for the two WOS:s is named the "Web Overlay Generator (WOG)" and can be seen as the dotted boxes in Figure 3.4 and 3.5. It is the part

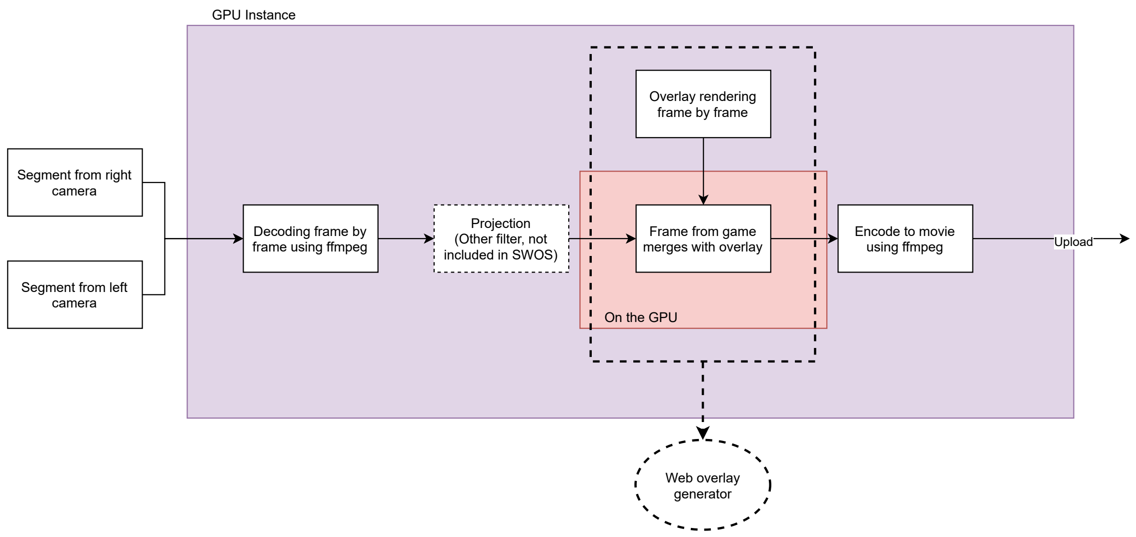

GPU Instance
Overlay rendering frame by frame
Segment from right camera
Segment from left camera
Decoding frame by frame using ffmpeg
Projection (Other filter, not included in SWOS)
Frame from game merges with overlay
Encode to movie using ffmpeg
Upload
On the GPU
Web overlay generator

Figure 3.4: Setup to use the standard web overlay solution (SWOS) in production.

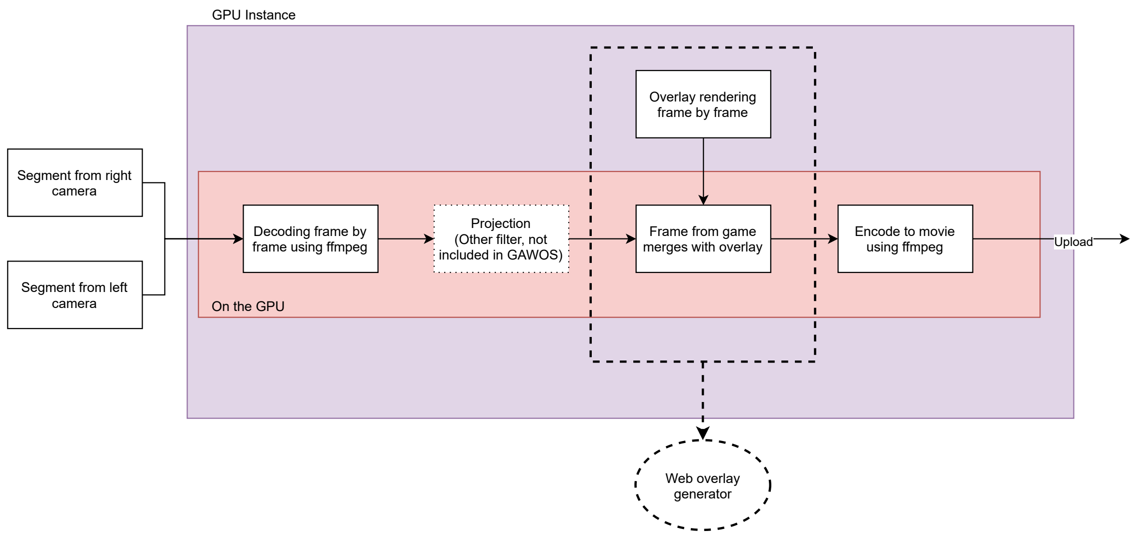

GPU Instance
Overlay rendering frame by frame
Segment from right camera
Segment from left camera
Decoding frame by frame using ffmpeg
Projection (Other filter, not included in GAWOS)
Frame from game merges with overlay
Encode to movie using ffmpeg
Upload
On the GPU
Web overlay generator

Figure 3.5: Setup to use the GPU accelerated web overlay solution (GAWOS) in production.

of the code that creates the overlay and adds in on top of the input frame. The implemented code for this part is almost identical for both solutions. The code structure of the WOG is seen in Figure 3.6 which can be divided into two parts that interact. The custom FFmpeg filter which is the C-code part and the Rust-code part. The filter is implemented to setup specified input and output formats and also to do processing of the data which calls on functions in Rust. When the filter runs it initializes the **Renderer** object in Rust. The **Renderer** initializes the **Model**, **Web Overlay Renderer** and **Backend** object. In the **Backend** the OpenGL is initialized to render to either the window system GLFW or EGL. In the **Model** the graphical setup is created with vertices, Vertex Array Object, Vertex Buffers Objects and Shader programs. This is pure OpenGL setup. In the **Web Overlay Renderer** the engine for rendering the web overlay is initialized, in this case it is Ultralight.

After everything has been set up properly the rendering of frames begin. First there is a check that there is any input frames, if there are then a function named render_frame in **Renderer** is called. This function is divided into three parts: upload frame, upload overlay and download. In the "upload frame" a single frame from the input is uploaded as a texture to the GPU with OpenGL. Then the frame is drawn using OpenGL commands on a quad that is covering a whole window. Next part is "upload overlay" where the overlay is generated and drawn. But before the overlay is uploaded as a texture it has to be computed. The way this is done is explained with pseudo code in Algorithm 1. The script run in the function "RunJavaScript" in Algorithm 1 is only intended to do style changes to the elements. The code run in the script should follow some predetermined rules to prevent the system from crashing. The code displayed in Algorithm 1 is where the overlay is generated and uploaded to a texture and thus the most relevant to understand for the thesis. After the steps run in the algorithm the overlay is uploaded and drawn on the window in the same way as before, the overlay is drawn on top of the other frame. The third part is the

Algorithm 1 Upload web overlay texture

```
ActivateTexture();
BindTexture();
for animation in animations do
    if animation.shouldRun(internalClock) then
        RunJavaScript(animation.getScript());
    end if
end for
SendToTexture(LockAndGetPixels());
UnlockPixels();
```

downloading of the frame, this is also done with OpenGL which takes the window where the frame and overlay have been drawn and saves this image as a texture. This texture is saved to the output in the previously specified output format. After the download phase the code iterates and checks for more frames, if there are more frames, the process runs again. When there are no more frames the web overlay generator is done and then the encoding takes over to generate the output as a movie.

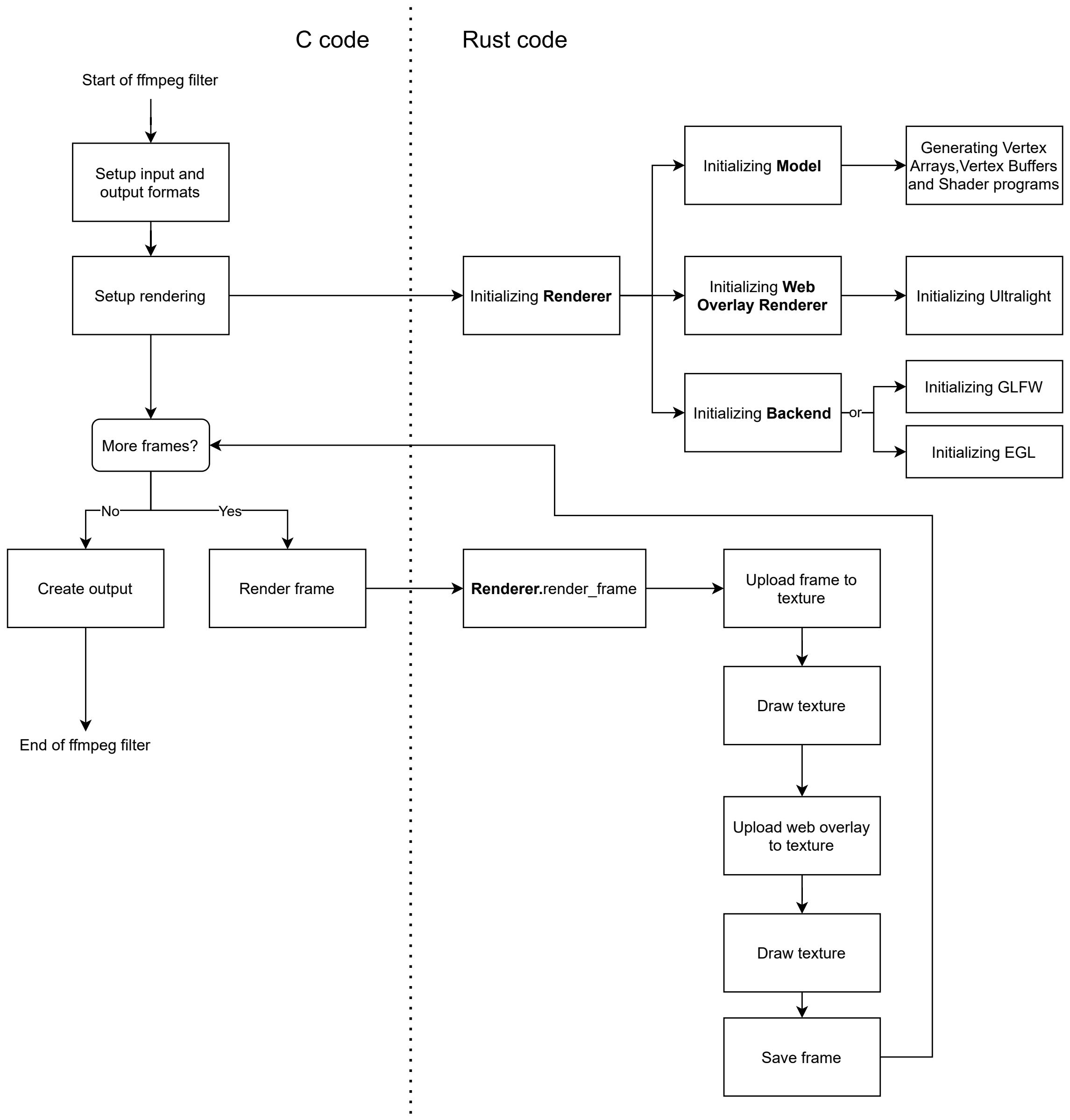


Figure 3.6: Code structure for the web overlay generator.

3.2.5 Overview of the solution

The WOS can either encode and decode the video with the CPU or use an NVIDIA GPU to hardware accelerate using CUDA frames. The WOG is almost identical for both the two solutions and therefore a graphic card is necessary for the OpenGL API, however they require different inputs. Both overlay solutions support logic for autonomous live scoring updates and can present time synchronized animations when called upon. The solutions is able to fetch any public accessible URL address and render as an overlay. They can both operate on a headless cloud machine.

Both of the WOS:s requires an AWS EC2 GPU instance whereas the SVG overlay requires an AWS EC2 CPU instance for processing and utilizes AWS lambda for generating the SVG overlay.

Both the SWOS and GAWOS inputs two video source files from the camera and uploads a segment, but the processing chain in between is different.

Chapter 4
Evaluation

4.1 Experimental setup

The experimental setup is performed by doing several iterations and measurements of rendering times and to compare the results for the different setups. The different tests run on different AWS EC2 instance types for headless rendering. The different machines used are seen in Table 4.1 and they can be divided into two main categories. The C group and the G group where the difference is that the G group is attached with a Nvidia Tesla T4 graphics card, enabling GPU rendering. These are several different instance types and groups but theses groups where chosen since the C-group is already adapted by the case company and is used for compute optimized tasks using the CPU and the G group is used for hardware accelerating using the GPU [4]. The rendering time of a segment is measured for ten iterations and the average is calculated. The measurements use the same input file which is a 25 FPS movie with 3840 x 2160 pixel resolution which is the standard video format for the case company. Both the resolution and the FPS affect the rendering time which is why the same input format is used for all the measurements. By studying the measurement data when performing iterations, the standard deviation is small and the average of ten iterations is therefor considered to give sufficient accuracy. The different machines have different hardware specifications which also leads to different pricing. The pricing is based on an on-demand pricing per hour scheme and is set by amazon [5]. The data for the pricing was gathered in May 2021 for the instances on the AWS Ireland data center.

The rendering setup for Spiideo Play consists of a series of steps to produce a video output film with the overlay. The setup for the SVG solution is seen in the background Chapter 2 in Figure 2.3. The SVG setup uses both AWS EC2 CPU instance and AWS lambda. To fairly compare the total rendering times and costs, this whole pipeline is investigated and measured for the SVG solution. The same goes for the web overlay solutions where the setup instead utilizes AWS EC2 GPU instance. This is necessary because the SVG overlay

Table 4.1: Data for different AWS instances [5].

Instance Name	Pricing per hour	vCPU	Memory	GPU
c4.xlarge	$0.226	4	7.5 GiB	No
c5.xlarge	$0.192	4	8 GiB	No
g4dn.xlarge	$0.587	4	16 GiB	Yes
g4dn.2xlarge	$0.838	8	32 GiB	Yes

Table 4.2: Pricing of AWS Lambda [8].

Memory.	Price per 1ms
3072	$0.0000000500

generates a video file as a standalone procedure and does not do any decoding of a video. And also since the solutions use different AWS services the pricing for each step is different. This approach would then consider the total latency time and their absolute costs. The pipeline consists of a series of processing and rendering steps. To be used in Spiideo Play, the two solutions would first download the source files from the cameras and the projected output is rendered from the source files. Then an overlay is generated and added to the output file before being uploaded for distribution. The pipeline differs for the two overlay solutions as the SVG overlay uses a CPU instance for downloading, projection and uploading and a AWS Lambda service for generating the SVG overlay whereas the web based overlay uses a GPU instance for the whole pipeline. Also for the GAWOS solution it can be used together with the projection step because of the possibility to use multiple filters after each other. This saves one pair of decoding/encoding as the projection step feeds a frame to the GAWOS without having to encode it. The setup of measuring the whole pipeline is set to answer RQ3 and RQ4 which compares the rendering times of the solutions and the costs. For the experimental setup the rendering times for the web based overlay solutions are performed. The rendering times were measured by extracting the execution time through the command line interface by using *time* command as the segments were processed. Data points for rendering the SVG overlay and the projection part is provided by the case company since they already have data for the measurements. Data for the upload and download process is also provided by the case company and uses a data storage service in amazon making the network activity stable.

The rendering steps and denotations in the pipeline is further presented in a list.

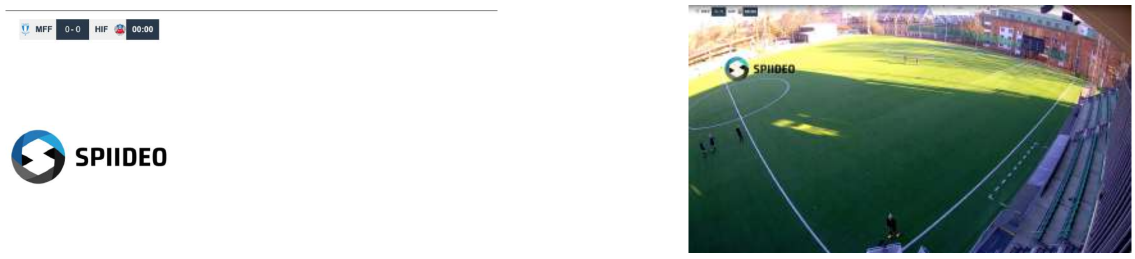
- **Projection** - The process of rendering the projected output frame from the input source files.
- **Download and Upload** - The process of downloading input source files and upload refers to uploading segment for distribution.
- **PUD** - The combined process of projection, upload and download.
- **SVG Overlay** - The process of generating the SVG overlay segment on AWS lambda.

4.2 Results

The results are divided to firstly answer the qualitative features of the overlay and later the rendering speed comparison.

4.2.1 Overlay features

The graphical overlay displaying a scoreboard in a web browser and also rendered as a overlay to a video is illustrated in Figure 4.1. The purpose of this thesis is not to implement any particular graphics design, it is more to highlight the possibilities and limitations of this solution.



(a) Overlay in browser **(b)** Overlay on video

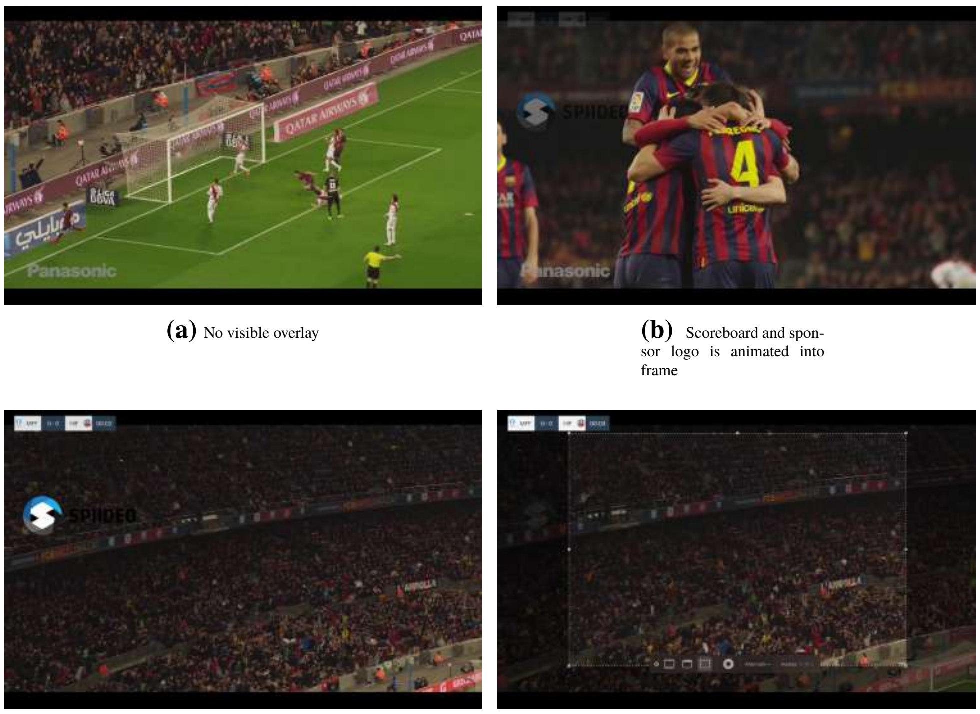
Figure 4.1: Graphical Overlay on Web technologies

Animations

The overlay can display animations using both CSS and JavaScript animations, however to synchronize with CSS animations the rendering time per frame must be synchronized with the CSS master clock as described in overlay implementation section 3.2.1. The CSS animation feature is therefore excluded as a feature. Using JavaScript animations one can change the style properties of the elements and do different types of animations. CSS animation are broadly used for animations and are easy to implement. However when using JavaScript animations, more advanced effects are possible to achieve such as making elements bounce, stop, pause and slow down [26]. In this thesis, simpler experimental animations are performed to establish a proof of concept. The scoreboard is animated to move in to visible area from hidden area, and with different logos representing sponsor logos or product logos where the opacity, i.e. the transparency, is gradually altered. An animation sequence where both the scoreboard and logo is animated by changing its opacity value is seen in Figure 4.2. In this animation the scoreboard is animated in to the video, and the logo is first rendered in to the video and then out. Since the whole service is autonomous, the solution also implements an automated animation triggering as described in the implementation.

Customization

By using this graphical overlay setup, any web resources or public accessible web addresses can be rendered as an overlay. Theoretically third parties can create any custom


(a) No visible overlay

(b) Scoreboard and sponsor logo is animated into frame

(c) Scoreboard and sponsor logo is overlayed to the frame

(d) Sponsor logo is animated out of the frame

Figure 4.2: Opacity animation

overlay without limitations. But there is a issue of how the overlay is synchronized with the video segment, so they display to the same timeline. It is of importance that the game clock featuring in the overlay is synced with the live stream. Since there is a latency of the live stream, there is a time-delay that the overlay must handle. Furthermore the overlay is rendered with the source files in a rendering speed that differs from real-time, and differs slightly for each frame and iteration. However, with the use of the predetermined logic, synchronization is handled but there are then requirements on the HTML elements to follow a certain specification list with certain tag names. Another approach to receive high customization for customers is that the case company develops different theme-based overlays which enables the customer to choose from a set of different overlay graphics.

4.2.2 Rendering speed

The following result displays rendering time for the three solutions; the case company's SVG solution, the standard based web overlay solution (SWOS) and the GPU accelerated web overlay solution (GAWOS) using CUDA frames.

SVG

The SVG solution renders fully on the CPU and is currently rendered in 5 FPS for Spiideo Play as a higher framerate is not necessary to produce the overlay since it does not generate any animations. In the case company's production the SVG overlay is running on AWS lambda and the times for rendering the overlay is provided by the case company and seen in Table 4.3. The case company's development team also has an upgraded solution for the SVG that has half the rendering time of the current solution in production.

Table 4.3: Rendering time in seconds for the SVG Overlay on AWS lambda in production and development.

Phase	SVG Overlay (s)
Production	3.825
Development	1.707

The processing time for projection of the input source videos, uploading and downloading of segments was generated by the case company and displayed in Figure 4.3. The m4.xlarge is another AWS EC2 instance but is not used anymore. Since the process time differentiates for different jobs, the y-axis in this diagram displays the distribution of the machines for different processing times. The blue line in the diagram is the combined processing time for the instances and is not of importance. The average processing time for a c4.xlarge instance is 21.4 seconds and for the c5.xlarge is 21.2.

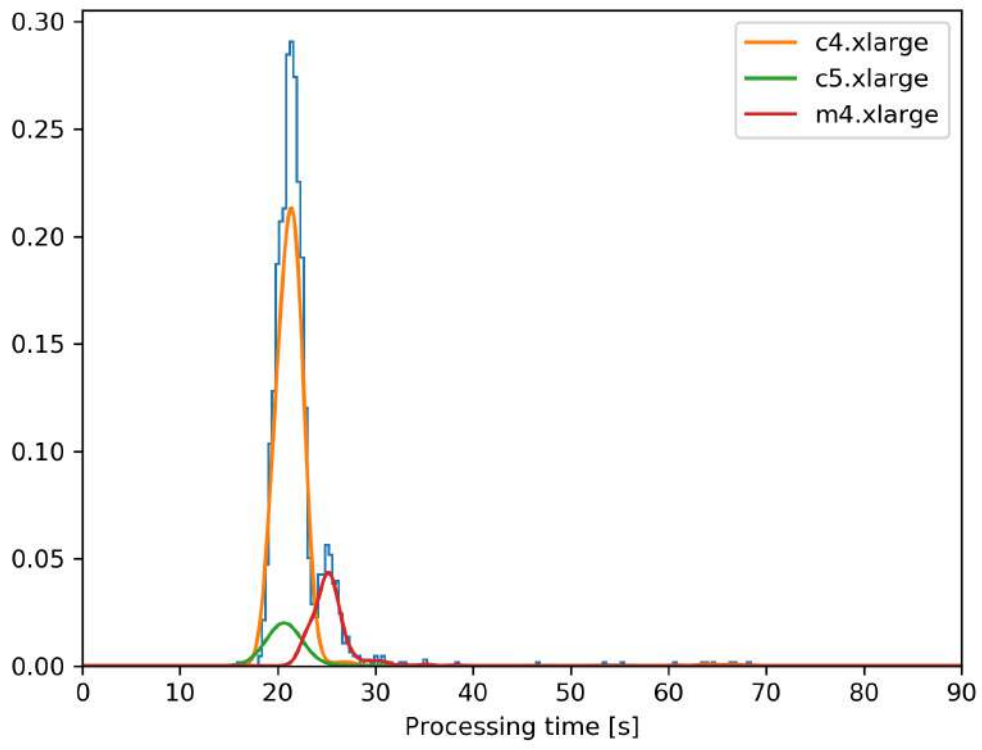


Figure 4.3: Total processing time c4.xlarge and c5.xlarge

The projection rendering process of the SVG also requires the SVG overlay to be a input. Since the processes does not run in parallel, the total rendering time for the SVG solution is therefore the combined time for projection, upload, download and the time for rendering the SVG overlay. The overall rendering time for the SVG solution in production and development is seen in Table 4.4 which is the combined time for the average processing time and generation of the SVG overlay.

Table 4.4: Processing time in seconds for the SVG overlay in production and development for different instances.

Instance	Phase	PUD (s)	SVG Overlay (s)	Total (s)
c4.xlarge	Production	21.4	3.825	25.225
c5.xlarge	Production	21.2	3.825	25.025
c4.xlarge	Development	21.4	1.707	23.107
c5.xlarge	Development	21.2	1.707	22.907

Standard Web overlay solution (SWOS)

The standard web overlay solution is able to render on most computers with a graphics card. The SWOS currently inputs a video segment and is placed after the projection step for use in the case company's production. Since the SWOS does not require the projection step to be rendered with the GPU it can either input the projected output segment from a GPU instance or a modified version of the CPU projection. The CPU projection would then be the similar to the one used in the SVG setup but would need modifications to only input two input files from the camera, whereas it currently also requires an overlay file. However the contribution of SWOS would be the same for both cases. The results from rendering the video with the overlay are presented in Table 4.5. The overlay result is the processing time of SWOS, which includes decoding, WOG and encoding. From the result, the processing times of adding the overlay to the input are significantly longer than the input video segment. On the g4dn.xlarge instance the processing time is almost six times the input video and it is also clear that the g4dn.2xlarge instance significantly improves the full processing speed to almost four times the input video.

Table 4.5: Rendering time in seconds for a 4 second segment.

Instance	SWOS (s)
g4dn.xlarge	23.0095
g4dn.2xlarge	16.8739

GPU accelerated Web Overlay Solution (GAWOS)

The GPU accelerated web overlay solution needs an NVIDIA graphics card to be able to render. The solution is built as an extension from the SWOS. The GAWOS aims to be inserted after the projection of a segment. It is possible to use multiple FFmpeg filter after each other with the advantage of only doing one set of decoding/encoding. The GAWOS is built as a custom filter where it can be used in a FFmpeg pipeline. Since it is hardware-accelerated it needs to run on a GPU, where the input frames are expected as textures. When decoding the input video, FFmpeg's built in CUDA decoder is used to generate CUDA frames.

The rendering time to project a segment when GPU accelerated is provided by the case company to 2.60 seconds. This includes decoding, computing the projection and encoding of a 4 second segment. There is also the process of downloading the two input video files from the cameras and also the process of uploading the rendered segment for distribution. The download and upload process takes approximately 2 seconds, this data is also provided by the case company. Making the process time of PUD to 4.6 seconds.

To measure the GAWOS's contribution to the total rendering time in the pipeline the idea is to run the GAWOS with and without the WOG. This is because the decoding and encoding is included in the rendering time of GAWOS, but the decoding is unnecessary in the pipeline since it can directly input CUDA frames from the projection step. The same goes for the projection step as the encoding is unnecessary as it feeds the CUDA frames directly to WOG. Therefor a pair of encoding and decoding needs to be removed for the total rendering time.

The approach is to first run the GAWOS with the WOG, which includes decoding, adding overlay and encoding, and measure the total rendering time. The same input file is then processed without the WOG which only involves decoding and encoding with CUDA frames. This time is then subtracted from the rendering time with the GAWOS to get the true contribution to rendering time of the solution.

The WOGs contribution is seen in Table 4.6 and the total rendering time for the GPU accelerated setup is seen in Table 4.7. There is a small difference in speed between the g4dn.xlarge and g4dn.2xlarge, where the g4dn.2xlarge is a bit faster but it is assumed that the difference in rendering time is because of few iterations, since the graphics card is the same for both instances.

Table 4.6: Rendering time in seconds for a 4 second segment.

Instance	With WOG (s)	Without WOG (s)	Only WOG (s)
g4dn.xlarge	4.3637	1.7635	2.6002
g4dn.2xlarge	4.3516	1.7755	2.5761

Table 4.7: Processing times in seconds of a 4 second segment by the GPU accelerated web overlay solution on different instances.

Instance	PUD (s)	Overlay (s)	Total (s)
g4dn.xlarge	4.6	2.60	7.20
g4dn.2xlarge	4.6	2.58	7.18

Web Overlay Overhead

The overlay from ultralight affects the rendering speed for the segment. This is measured by the rendering time of the solution with the overlay and without. To run without the overlay, the web overlay renderer is not initialized and there are not any uploads of web overlays to the texture. The result from the measurements are displayed in Table 4.8. In the table the denotations are:

- **With Ultralight** - Running the solutions with WOG as usual.
- **Without Ultralight** - Running the solutions but removing the web overlay to WOG i.e. the initializing of the web overlay renderer.
- **Only Ultralight** - The difference of running with and without the overlay renderer. Net rendering time contribution, of the overlay.

From the data the overlay adds 2.4434 seconds to the total rendering time on the GAWOS for a g4dn.xlarge machine.

Table 4.8: Rendering times in seconds with and without the web rendering engine ultralight for g4dn.xlarge.

Instance	With Ultralight (s)	Without Ultralight (s)	Only Ultralight (s)
g4dn.xlarge	4.3637	1.9203	2.4434

In order to reduce ultralights overhead cost, the ultralight's rendering view is experimented into smaller views as seen in Table 4.9. These views are in smaller scales that can be scaled up to display the full quad. The table displays rendering times for the GAWOS on a g4dn.xlarge instance with and without the WOG.

Table 4.9: Rendering time in seconds of a 4 second segment with modified scale ratio, run on a g4dn.xlarge instance.

Ratio	With WOG (s)	Without WOG (s)	Only WOG (s)
1:1	4.3637	1.7635	2.6002
1:2	2.6602	1.7635	0.8967
1:4	2.2142	1.7635	0.4507

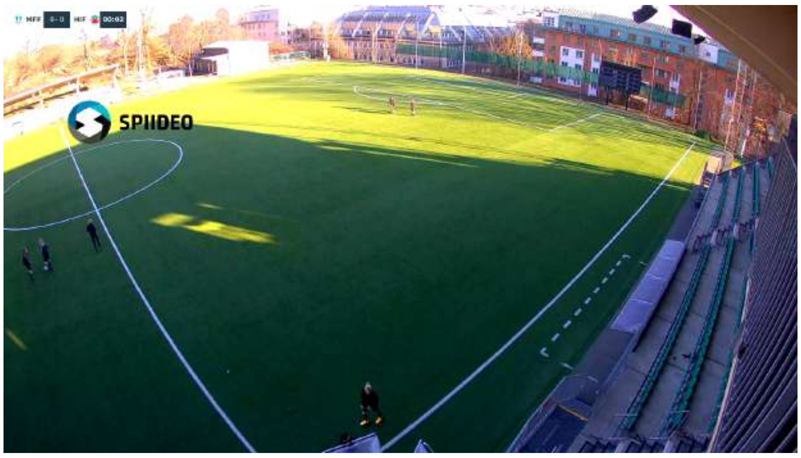
The rendered view's bitmap binds to the texture which displays on the quad. Therefor even if the view is rendered in smaller sizes it still displays the overlay on full display. However its content is amplified in size since the GPU rasterizes the pixels multiple times when scaled up. The overlay for the different scale is seen in Figure 4.4a, 4.4b and 4.4c.

4.2.3 Comparison (all overlays)

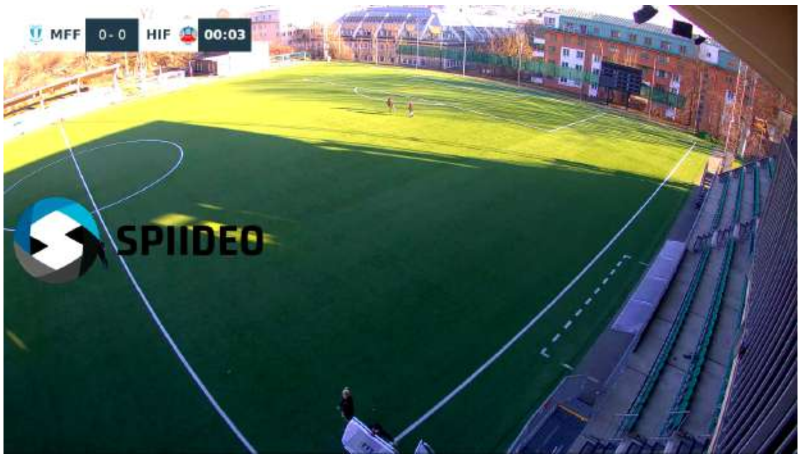
This section will compare the SVG overlay solution to the web based overlay solutions to further answer the research questions.

Rendering speed

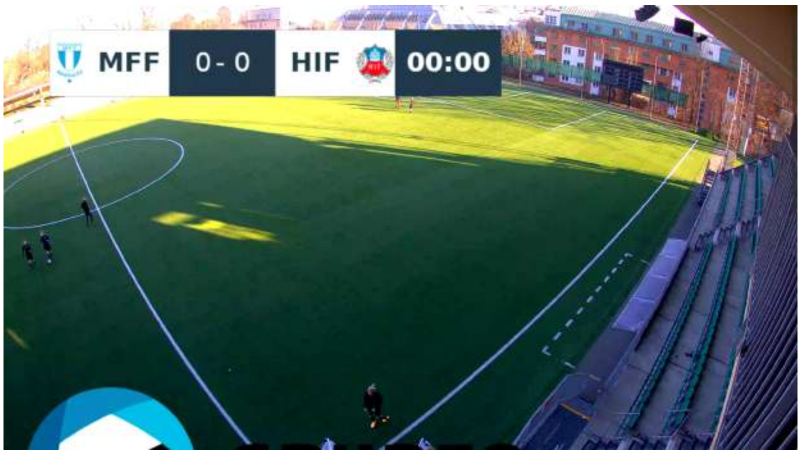
By directly studying the rendering speed it is clear that SWOS is inadequate since the total rendering time is greater than the input video time. This solution could be rendered in parallel and render more segments at the same time. But as the SWOS still requires a instance



(a) Ratio 1:1



(b) Ratio 1:2



(c) Ratio 1:4

Figure 4.4: The output result when changing the ratio of the HTML view.

with a graphic card it provides the same functionality, but with a greater rendering speed and cost. In consequence the SWOS is discarded.

The total processing time for the GAWOS and the SVG solution is visually displayed in Figure 4.5. By observing the total process times, it is clear that the GAWOS is much faster than the SVG both in production and development. The time for rendering the overlay is almost the same. The SVG in development is actually the fastest followed by the GAWOS and lastly the SVG in production. The big reason for the GAWOS being faster is due to it having faster projection, uploading and downloading speed in comparison to the SVG solution. From this data is it is clear that the rendering time is almost three times faster for the GAWOS compared to the SVG in production which answers RQ3; what is the difference in rendering time between the solutions.

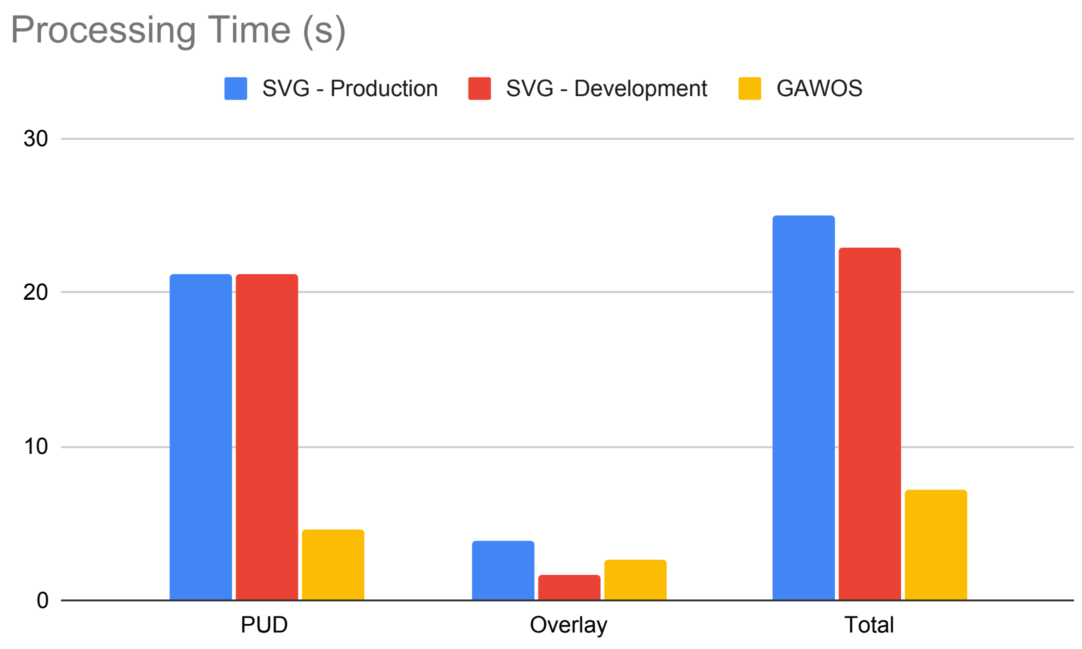


Figure 4.5: Total process time for generating a 4 second segment. The both SVG solutions are running on a c5.xlarge for the processing and a AWS Lambda for the overlay. The GAWOS is running on a g4dn.xlarge since it has more or less the same time as the g4dn.2xlarge.

Cost

The pricing is based on the total processing time for the different solutions multiplied by the AWS-pricing, converted to USD per second, for the instance they run on. The price for the SVG solution have both an instance and an AWS Lambda. The cost for running AWS EC2 instances and AWS lambda is seen in Table 4.1 and 4.2. The cost for the SVG solution on different instances in both development and production is seen in Table 4.10. The cost is calculated by multiplying the rendering time with the cost for the AWS services, displaying the total processing cost in $ per segment. From the data the c5.xlarge is the most cost efficient instance for the SVG solution in both production and development.

Table 4.10: Processing cost in $ per segment for the SVG in production and development for different instances.

Instance	Phase	PUD ($)	Overlay ($)	Total ($)
c4.xlarge	Production	0.00134	0.0001913	0.00153
c5.xlarge	Production	0.00113	0.0001913	0.00132
c4.xlarge	Development	0.00134	0.0000854	0.00143
c5.xlarge	Development	0.00113	0.0000854	0.00122

The cost for GAWOS for instances g4dn.xlarge and g4dn.2xlarge is seen in Table 4.11. The cost is calculated by multiplying the rendering time with the cost for the AWS EC2 instances, displaying the total processing cost in $ per segment. From the data the most cost efficient instance for GAWOS is the g4dn.xlarge.

Table 4.11: Processing cost in $ per segment for the GPU accelerated Web Overlay Solution.

Instance	PUD ($)	Overlay ($)	Total ($)
g4dn.xlarge	0.00075	0.00042	0.00117
g4dn.2xlarge	0.00107	0.00060	0.00167

The cost for the solutions are compared to each other for the most cost-effective instances. A visual representation of the comparison is seen in Figure 4.6. GAWOS is the most cost efficient followed by SVG solution in development and the the SVG solution in production, which answers RQ4 regarding which overlay solution is the most cost efficient.

Features

The SVG overlay can display info-graphics for the user in high resolutions. With the current setup it is possible to create any type of static graphics which is SVG based, however with the current setup it is not possible to do animations. The GAWOS is able to manipulate the HTML using JavaScript while rendering. This way it is possible to manipulate an element frame by frame and create animations. Since the GAWOS is based on HTML, CSS and JavaScript, the many users in the large frontend community are well versed within these languages. Other additional features to the web based overlay is the possibility to use JavaScript. This opens the possibility to make the overlay more interactive as JS code can fetch data from different API's to directly display in the HTML overlay. The web based overlay is able to do anything that the current SVG overlay can do but also more, such as animations and possibilities to use JavaScript making it more feature rich than the SVG overlay, answering RQ1.

Customization

The SVG overlay is developed by the case company and can create any overlay graphic with still images. However the technologies of creating a SVG overlay by using the Tera template and creating SVG files is not as common as using standard HTML. The web based overlay has its main advantage of being developed in common languages and also

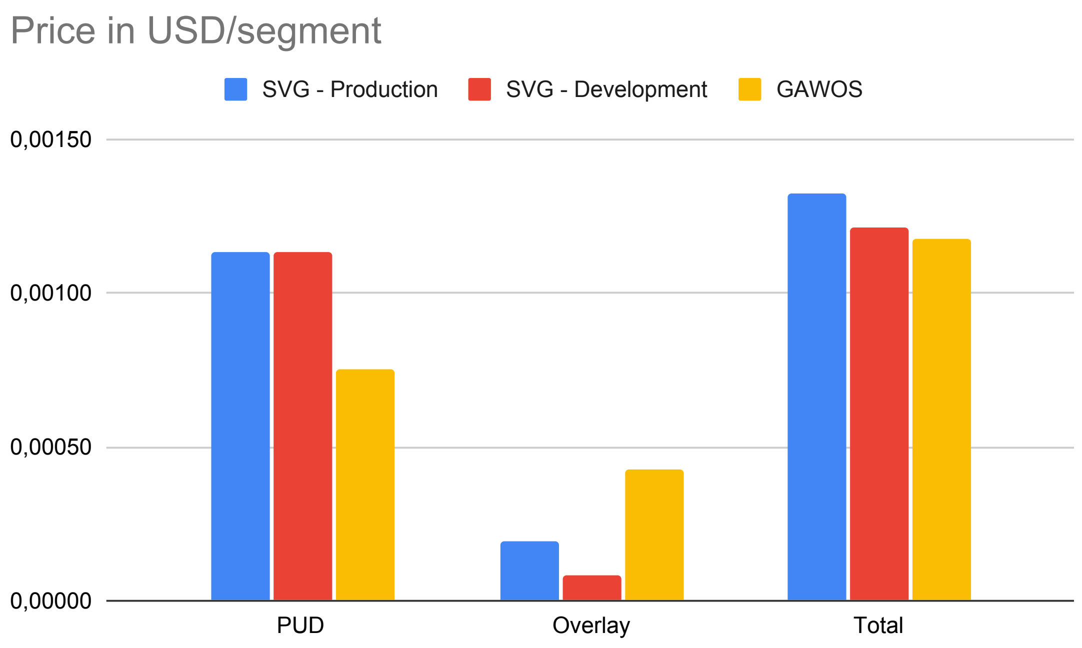


Figure 4.6: Price in US dollars for generating a 4 second segment. The both SVG solutions are running on a c5.xlarge and the GAWOS is running on the g4dn.xlarge.

to be able to render any public accessible website by its URL address. Although to use the presented overlay logic for the case company, the implementation is similar with using a template as the ID names need to be specified similarly to the use of placeholders. This gives the conclusion that the web based overlay has a higher grade of customization than the SVG overlay since the technologies to create a website is well established and that it can render any accessible website. This answers RQ2 on how customization can be accomplished using the web based overlay solution.

4.3 Discussion

The new presented solution, the GPU accelerated web overlay solution, resulted in both higher performance and is more cost efficient. This section involves a discussion of the results and future improvements on the solution. It also covers other application areas that the solution might be of value to. A discussion on the different assumptions made for the experimental setup and any disclaimers that might be of relevance is presented.

4.3.1 Overlay Quality

The overlay is differentiated by its two main design decision. The first being that the overlay should be built on web-technologies. The purpose of why the overlay is built on web technologies is mainly because the possibilities of what one can create. These technologies are well versed within in the programming community and one can display anything that one can see on a website. Thereby allowing the solution to have a high customization and flexibility of the content and that the overlays can be created by a large community. For the case company, this addresses the issues of allowing full customization's from its different customers such as sports clubs and other media companies. However for the overlay to fulfill these needs, an API needs to be created in order to use the functionality of the overlay concerning that its rendering faster than real-time. This is to get the time synchronized and despite this, overlays can be created more easily with this solution in-house than the current SVG solution. These types of possibilities adds value to the product and increases its quality. Furthermore the overlay can increase engagement and increase ad-revenue for customers with its possibility to display motion graphics.

The second design decision is that the overlay should be burned into the video file before distribution instead of letting the overlay to be rendered client side on the viewers device. This decision comes with both its advantages and disadvantages. The main idea to this decision lies in that the output video file is not dependent on having any rendering possibilities and networking calls for the overlay on the users device. By using this approach the overlay increases its interoperability so it can be streamed and viewed on any media playing platform without having to render HTML client side. However, since OTT live streaming already requires a solid network connection and the fact that most streaming platforms are able to render HTML, it doesn't come with that many advantages in the present. But with the ability to only send data bytes and also display a overlay allows for distribution to different IP-TV solutions or smart-TV apps which otherwise wouldn't be possible. Despite this, one can argue that client side rendering probably would be more performance efficient. This is because the client side overlay rendering wouldn't have to decode any video but rather simply render a view on top of the playing video. But with any product development there needs to be decisions and trade-offs. This trade-off favors scale-ability and operability to other platforms and it also presents a general solution for burnt in overlays using HTML for any video, which was not discovered how to achieve in the preliminary investigation to this thesis.

4.3.2 Reducing latency time

One of the main ideas to perform this thesis is to investigate how the overlay can be generated in a GPU accelerated environment with the idea to reduce the rendering speed. As described earlier in the background section 2 the glass-to-glass latency is crucial for the viewing experience and it is of essence that all different steps in a streaming platform are optimized to minimize the processing time. This solution, presents and quantifies, and perhaps to no-ones surprise, that hardware accelerated rendering does offer great performance boosts and also that it doesn't come at a higher cost. As it is clear that the rendering speed is greater than the current solution it would also be interesting to produce it as a standalone video overlay with no decoding of a input video. If this would run in sequence, as the current SVG overlay does, this would increase the rendering time. But this modular approach could also run in a parallel sequence to the other rendering process of stitching the source files from the camera and could then compare whether the total processing time would be affected.

4.3.3 Other Application Areas

As the solution satisfies the function of rendering an UI to a frame, there are other application areas that this could be interesting for usage.

Game Engine UI

One can argue that half of this project is describing the building of game engine UI with HTML graphics. The process of building a UI on top of some graphics display for i.e. a game can be achieved by the using the methodology of the graphics setup. The only addition would be to display the games pixel data on the same texture. When developing the web overlay solution, it was firstly developed to display the UI to screen in a custom OpenGL setup. For this setup the FPS was measured and it could render in speeds up to 300fps on local Macbook Pro machines from 2015, making it feasible for a game engine UI.

Video Overlay Graphics

While generating the overlay for this project, it is necessary to achieve fast rendering speed in order to reduce latency. But another interesting aspect is that both the SWOS and GAWOS are able to input any movie source to add HTML graphics on top of a video. Therefor making it useful for other applications where the user wants to create after effects to display overlay graphics with very few limitations in a easy way, with free open source software. Although Ultralight has a perpetual license fee for businesses, it can be used cost-free for private users. And as this thesis describes the general implementation it can be implemented with other cost-free web rendering frameworks which there are plenty of. And for this use case, the rendering speed is not the same priority, in which the SWOS could be a feasible solution.

4.3.4 Future improvements

The created solution is still in a development phase and there are lots of improvements that could be done. In this subsection some of them are presented.

CSS Animations

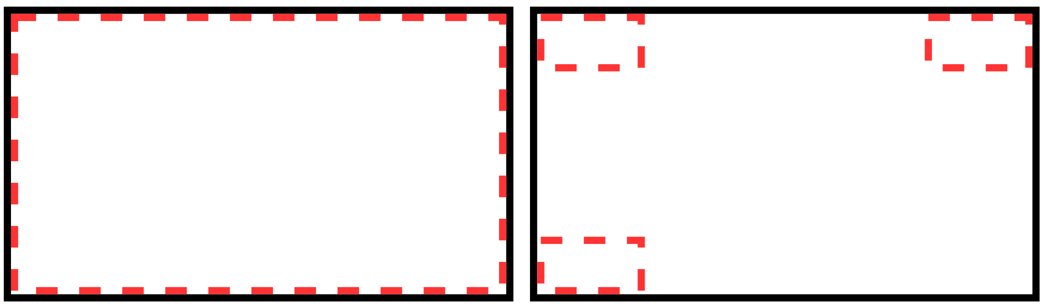
With the current setup it is not possible to rely on CSS animations. To use CSS animations the rendering time per frame must be synchronized with the CSS master clock in the rendering engine. The CSS animation feature can be enabled by overriding and controlling the CSS master clock.

Improve rendering speed

The ultralights web rendering engine is for the moment rendering on the CPU to extract a bitmap and copies the data to GPU memory. This is likely to account for the major difference in speed when running with and without the overlay. To improve the rendering speed it is possible to set up a custom GPU driver to directly render using the GPU. Not only will it improve the rendering speed for the UI but probably also for the whole process since this would reduce context switches between the CPU and GPU. To create a custom GPU driver is a more complex task, and it would require some effort to implement it.

Another way to possible improve the rendering speed would be to run on a more powerful GPU. A drawback could then be the more powerful GPU will be less cost-efficient.

The current WOG is drawing the overlay texture over the entire screen. Since only small portions of the video is necessary to cover with overlay another improvement would be to draw textures on predefined areas to lower the computations.



(a) Current usage of texture

(b) New idea to use the textures

Figure 4.7: The current texture setup and a new idea for doing it. The black frame represent one video frame and the red dotted rectangles represent the textures.

From the web overlay overhead section 4.2.2 it is obvious that the rendering size of the web overlay view matters a lot. By changing the ratio, the speed of the solution improved a lot. Further work would be to find the optimal size, with lowest rendering speed possible without losing too much in picture quality. A combination of lowering the ratio and changing the texture size would be a great improvement, and could be accomplished with a few modifications.

Modularity

Another important aspect in software engineering is using modular designs to maximize developer productivity. The SWOS is limited to requiring a graphic card and the GAWOS requires graphic card from NVIDIA. The solution also requires a video input file which adds a little to the complexity. Future improvement would be to implement the SWOS as fully software based and also add a feature to produce a video with just the overlay, similarly to how the SVG solution works. The approach of producing an overlay video standalone, might be more efficient in terms of rendering times.

OpenGL software emulator

The new software based web overlay solution currently uses OpenGL to communicate with the GPU driver. As previously brought up in the implementation section it is possible to use a software emulator for OpenGL using the Mesa3D graphics library and a software based rasterizer. This approach would be interesting to proceed because it would give the same features as the GAWOS but could instead run on a CPU instance. The CPU instances are more cost efficient and this could both add to modularity, flexibility and perhaps cost. Meaning a fully software based SWOS could be inserted directly in the current SVG setup as well as with a GPU accelerated processing. The SWOS could input any video segment and simply render the overlay using a CPU. This setup would allow for further measurements in terms of latency and cost because it can be tested with both CPU and GPU processing. Furthermore it would be able to run the solution on any machine making it more flexible for developing and testing the overlay.

4.3.5 Assumptions and Disclaimers

As the Spiideo Play system uses multiple instances which are delegated rendering jobs, they are not utilized 100% of the capacity in production. For this thesis it is assumed that the EC2 instances are used 100% of the capacity. And as the GPU instances cost more in comparison to the CPU instances, this may effect the overall cost result.

It is assumed that the processing times for the SVG setup and the GPU setup, inputs two video source files.

For the Spiideo Play, they use several different instance types for rendering jobs. The different instances used in production and the distribution of jobs is seen in figure 4.8. This thesis compares with the most common running instance type c4.xlarge and the c5.xlarge. These were measured in the thesis, because the data for processing the segment with two input video source is known for these instances. As the diagram displaying the distribution of jobs does shows the processing time for the other instances, it displays the average processing time when there could be one, two or three video input files. As the data could not be filtered out to only display rendering times for two inputs, the other instances are not included in the results and in this thesis.

4.3.6 Research Questions

To sum up the questions that has led this thesis forward a short summarize of the answers is provided.

RQ1 Has the web based overlay more features compared to the current SVG overlay?

The web based overlay is able to do anything that the current SVG overlay can do but also more, such as animations and possibilities to use JavaScript making it more feature rich than the SVG overlay.

RQ2 How can a customization be accomplished using a web based overlay?

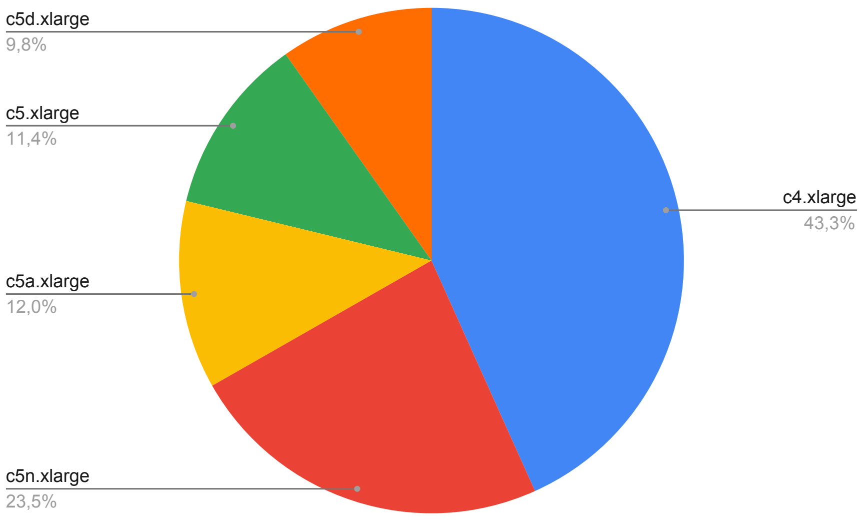
Customization can be accomplished by using templates and passing custom user data in different placeholder values, just as the current SVG overlay solution. However, because the web based overlay solution also has an embedded rendering engine with full networking capability it is also possible to fetch and render any public accessible content over the internet.

RQ3 What is the difference in rendering time for the overlay solutions?

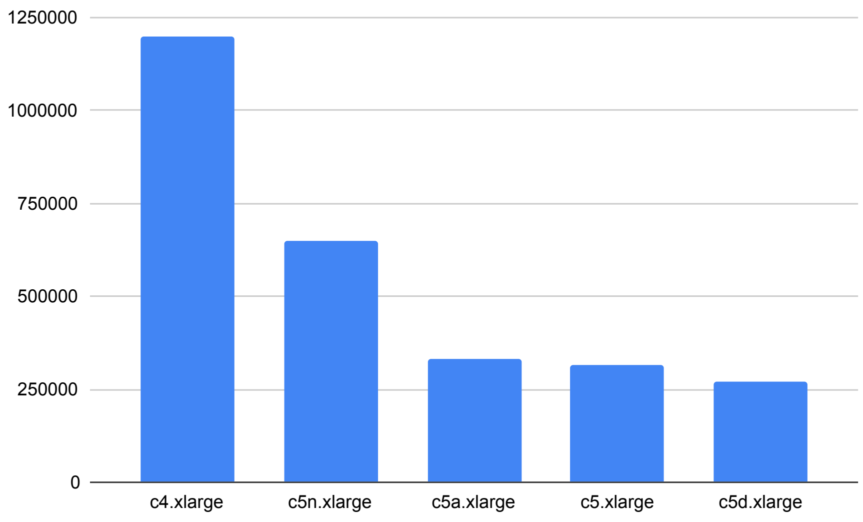
After some testing done for the different solutions it stood clear that the GAWOS was almost three times faster than the SVG.

RQ4 Which of the overlay solutions is the most cost efficient?

After comparisons between the different solutions the result prove that the GAWOS was the most cost efficient.



(a) Current usage of instances



(b) Distribution of jobs

Figure 4.8: Running instances in Spiideo Play production and distribution of jobs.

Chapter 5
Conclusions

The goal was to find a solution to use web-technologies to create overlays for a live stream of a sport event. A web overlay solution was created as a FFmpeg filter with an HTML/CSS/JS engine in the form of Ultralight. An extension of the web overlay solution was constructed with GPU accelerated encoding and decoding. The solution is able to take an input movie and display the input movie with an overlay as output. Based on the results in Chapter 4 it is also clear that running the solution hardware accelerated on a graphics card makes the whole process much faster.

The case company's current solution is using a SVG overlay and run it on the CPU. When comparing the new GAWOS with the SVG solution in production it was clear that the entire process of the new solution was both faster and cheaper than the one now used. Even when updated with a faster overlay rendering time, as the SVG solution in development, the GAWOS was faster and cheaper. Much of the reason for the better speed is due to the big difference in processing time before/after the actual overlay generation which also is a contributing factor why the whole process is cheaper. When only inspecting the generating of the overlay it is actually the SVG in development that is the fastest. Since the SVG is run on a AWS Lambda and not a instance with GPU it has a lower cost per second and together with the faster rendering it gives a lower price. The GAWOS is the most expensive when it comes to only generating the overlay. Since the solutions have different setups it is not fair to account the single steps of the processes too much, but instead look at the process as a whole. All in all, with what was expected, the new solution should be the way to continue develop.

Further work would be to improve the speed. Some possible ideas would be to accelerate the HTML/CSS/JS engine by implementing a GPU driver for it. Just by changing the ratio of the overlay a great speed increase was made and it would be interesting to continue testing how small the overlay rendering can be without losing quality.

References

[1] Platform output. https://www.singular.live/platform/output.

[2] Kumar Ahir. How can ott trump the next way of entertainment with virtual reality?, May 2021. https://medium.datadriveninvestor.com/how-can-ott-trump-the-next-way-of-entertainment-with-virtual-reality-6a8d1e86a8c.

[3] Song Ho Ahn. Opengl rendering pipeline, May 2021. http://www.songho.ca/opengl/gl_pipeline.html.

[4] Amazon. Amazon ec2, June 2021. https://aws.amazon.com/ec2/?ec2-whats-new.sort-by=item.additionalFields.postDateTime&ec2-whats-new.sort-order=desc.

[5] Amazon. Amazon ec2 on-demand pricing, May 2021. https://aws.amazon.com/ec2/pricing/on-demand/.

[6] Amazon. Aws elemental medialive now supports html 5 motion graphics overlay, May 2021. https://aws.amazon.com/about-aws/whats-new/2021/04/aws-elemental-medialive-now-supports-html-5-motion-graphics-overlay/.

[7] Amazon. Aws lambda, June 2021. https://aws.amazon.com/lambda/.

[8] Amazon. Aws lambda pricing, May 2021. https://aws.amazon.com/lambda/pricing/.

[9] NVIDEA Developer. Ffmpeg, May 2021. https://developer.nvidia.com/ffmpeg.

[10] Django, June 2021. https://www.djangoproject.com/.

[11] FFmpeg, May 2021. https://www.ffmpeg.org/.

[12] Tali Garsiel and Paul Irish. How browsers work: Behind the scenes of modern web browsers - html5 rocks, May 2021. https://www.html5rocks.com/en/tutorials/internals/howbrowserswork/.

[13] GLFW. An opengl library, May 2021. https://www.glfw.org/.

[14] Jake Goulding. What is rust and why is it so popular?, May 2021. https://stackoverflow.blog/2020/01/20/what-is-rust-and-why-is-it-so-popular/.

[15] Scott Grizzle. The key streaming video challenges the industry faces today, May 2021. https://www.streamingmedia.com/Articles/Editorial/Featured-Articles/The-Key-Streaming-Video-Challenges-the-Industry-Faces-Today-133406.aspx?utm_source=related_articles&utm_medium=gutenberg&utm_campaign=editors_selection.

[16] BS Group. Ott and sports = a match made in heaven?, May 2021. https://www.bsgroup.eu/ott-and-sports-a-match-made-in-heaven/.

[17] Khronos Group. Egl - native platform interface, May 2021. https://www.khronos.org/egl.

[18] Khronos Group. The industry's foundation for high performance graphics, May 2021. https://www.opengl.org/about/.

[19] Khronos Group. Rendering pipeline overview, May 2021. https://www.khronos.org/opengl/wiki/Rendering_Pipeline_Overview.

[20] Isabelle Hahn. Why live sports are the next big thing in streaming video, May 2021. https://vimeo.com/blog/post/2019-is-the-year-for-live-streaming-sports/.

[21] Kholodov Igor. Textures in opengl, May 2021. http://www.c-jump.com/bcc/common/Talk3/OpenGL/Wk07_texture/Wk07_texture.html.

[22] Jinja, June 2021. https://jinja.palletsprojects.com/en/3.0.x/.

[23] De Vries Joey. Hello triangle, May 2021. https://learnopengl.com/Getting-started/Hello-Triangle.

[24] De Vries Joey. Opengl, May 2021. https://learnopengl.com/Getting-started/OpenGL.

[25] Keats. Tera, June 2021. https://github.com/Keats/tera.

[26] Paul Lewis and Sam Thorogood. Css versus javascript animations | web fundamentals, May 2021. https://developers.google.com/web/fundamentals/design-and-ux/animations/css-vs-javascript.

[27] Mesa3D. Introduction, May 2021. https://docs.mesa3d.org/.

[28] Peter Messmer. Egl eye: Opengl visualization without an x server, May 2021. https://developer.nvidia.com/blog/egl-eye-opengl-visualization-without-x-server/.

[29] Lina Nikols. Video encoding basics: What is latency and why does it matter?, May 2021. https://www.haivision.com/blog/all/video-encoding-basics-video-latency/.

[30] NVIDIA. Cuda zone, May 2021. https://developer.nvidia.com/cuda-zone.

[31] OBS. Home. https://obsproject.com/wiki/Home.

[32] Paul Ojuederie. 8 trends and predictions for ott sports in 2021: Mpp global blog, May 2021. https://www.mppglobal.com/news/blog/8-trends-and-predictions-for-ott-sports-2021/.

[33] Michael Proman. The future of sports tech: Here's where investors are placing their bets, May 2021. https://techcrunch.com/2019/10/01/the-future-of-sports-tech-heres-where-investors-are-placing-their-bets/.

[34] Psychonautwiki. psychonautwiki/rust-ul-sys, May 2021. https://github.com/psychonautwiki/rust-ul-sys.

[35] PWC. Video streaming shakeup: Survey of consumer attitudes and preferences, May 2021. https://www.pwc.com/us/en/services/consulting/library/consumer-intelligence-series/consumer-video-streaming-behavior.html.

[36] Quortex. The truth about low latency in ott, Jan 2020. https://www.quortex.io/post/the-truth-about-low-latency-in-ott.

[37] realitychecksystems. Singular.live. https://www.realitychecksystems.com/singular-live/.

[38] Felix Richter. Infographic: Sports broadcasts are tv's last stronghold, May 2021. https://www.statista.com/chart/13113/sports-on-us-tv/.

[39] Bibek Shah. How do web browsers work?, May 2021. https://medium.com/@bibekshah09/how-do-web-browsers-work-1245d5b06c51.

[40] Strexm. What is strexm. https://strexm.tv/.

[41] Techopedia. Rendering, June 2021. techopedia.com/definition/9163/rendering.

[42] K. T. Ulrich and S. D. Eppinger. *Product Design and Development 5th ed.* McGraw-Hill/Irwin, 2011.

[43] Ultralight. Using a custom gpudriver, May 2021. https://docs.ultralig.ht/docs/using-a-custom-gpudriver.

[44] Christer Whitehorn. Enhance your live stream with html5 motion graphics, May 2021. https://aws.amazon.com/blogs/media/awse-enhance-live-stream-html5-motion-graphics/.

[45] xorg. Xserver, May 2021. https://www.x.org/releases/X11R7.7/doc/man/man1/Xserver.1.xhtml#heading3.

INSTITUTIONEN FÖR DATAVETENSKAP | LUNDS TEKNISKA HÖGSKOLA | PRESENTERAD 2021-06-10

EXAMENSARBETE Dynamic and Feature-Rich Graphical Overlay based on Web Technologies for Live Streams
STUDENTER Daniel Pendse, Gustav Sjölin
HANDLEDARE Michael Doggett (LTH), Erik Zivkovic (Spiideo AB)
EXAMINATOR Flavius Gruian (LTH)

Snabb och enkel grafisk overlay för filmer och liveströmmning

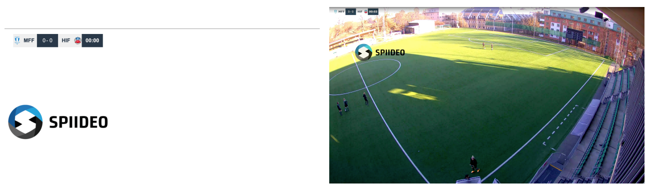
POPULÄRVETENSKAPLIG SAMMANFATTNING **Daniel Pendse, Gustav Sjölin**

För att förbättra engagemanget och tittarupplevelsen för sportsändningar är det viktigt att engagera tittaren med informativ grafik och interaktion. Detta arbete presenterar en hårdvaruaccelererad lösning som generar overlay grafik baserat på kända webbteknologier på liveströmmar.

Spiideo är ett Malmöbaserat företag som tillhandahåller en automatisk livesändningsservice för sportklubbar, Spiideo Play. I kombination med att tittarna önskar mer interaktiv realtids information och att kunderna till tjänsten önskar egen designade eller personliga grafiska overlays, växte detta arbetet fram för att hitta en snabb och dynamisk grafisk overlay. Det gör det möjligt att skapa personliga grafiska overlays för kunderna att distribuera ut till deras tittare genom Spiideo.

Detta resulterade i en lösning som använder sig av webbteknologier för att generera en grafisk overlay ovanpå livesändningen. Genom att använda webbteknologier är det enkelt att skapa olika overlays och möjligt att göra animationer för att höja tittarupplevelsen. Spiideo har redan en lösning som bygger på vektor grafik, som har en högre komplexitet och är svårare att göra anpassningsbara overlays samt saknar möjligheten att göra animationer.

Vi har skapat en lösning som gör det möjligt att använda sig av vanliga webbresurser som, HTML, CSS och JavaScript, för att skapa grafiska overlays på video segment. Vi skapade en lösning med en integrerad webb-renderingsmotor. Denna lösning utvecklades till att rendera hårdvaruaccelererat, med hjälp av en dators grafikkort för att minimera renderingstiden. I bilden syns dels en resultattavla baserad på webbresurser öppnad i en webbläsare samt en bild som visar när resultattavlan ligger ovanpå en videoström.



(a) HTML code (b) Output

Resultatet kan användas till att producera overlays till live strömmar men även för att skapa overlays på videos i efterhand. Arbetet presenterar en hårdvaruaccelererad lösning som har jämförts gentemot den nuvarande lösningen i Spiideo's system med avseende på renderingstid, kostnad och funktionalitet. Det visade sig att renderingstiden för den nya lösningen är ungefär 70% mindre med en minskad kostnad på ungefär 15%. Funktionaliteten är betydligt bättre för den nya lösningen.