A dynamic method for rendering overlay on live video

With motion detection and masking using DirectX 11 and HLSL
Abstract

The digitalization leads to that many physical solutions are replaced by digital once. Especially in the surveillance and security business have humans been replaced by cameras which are monitored from a remote location. As the power of the computers has increased, live video can be analyzed to inform the controller about anomalies which a human eye could have missed. SAAB – Air Traffic Management has a digital solution to provide Air Traffic Service which is called Remote Tower. This project will come up with a recommendation on how SAAB can dynamically render overlay based on live video, which marks the runways and taxiways on the airfield.
Acknowledgments

I want to thank the people at SAAB – Air Traffic Management, I learned much and got valuable experience working in TEAM7. I would especially like to thank Daniel Nilsson for helping and introducing me to the graphical programming and Mattias Johansson for his good feedback when analyzing the different approaches and settings. I would also want to thank my supervisor Jesper Andersson at Linnaeus University, for informative seminars and input during this project.
# Table of content

Abstract

Acknowledgments

Table of content

List of Figures

## 1 Introduction

1.1 Background

1.2 Expected Results

1.3 Method

## 2 Theory and Technologies

2.1 Graphics Programming

2.1.1 Video Encoding and Decoding

2.1.2 DirectX

2.2 Color Theory

2.2.1 RGB

2.2.1 HSL

## 3 Implementation

3.1 Dynamic Overlay

3.1.1 Analysis Program and Dynamic Overlay Algorithm

3.1.2 Implementation into the video decoder

3.2 Motion Detection

## 4 Results and Discussion

4.1 Dynamic Overlay

4.2 Motion masking

4.3 Limits and Areas of Improvement

4.3.1 Lightness Offset
4.3.2 Masking Shadows 19
4.3.3 Trail from Slowly Moving Objects 19
4.3.4 Masking objects 20

5 Conclusion ____________________________________________ 21

References _____________________________________________ 22
List of Figures

Figure 1.1 Remote Tower Module with two positions [1].................................1
Figure 1.2 Mast with camera housing [1]..........................................................2
Figure 2.1 Y’CbCr frame to the left followed by the representations of Y’, Cb and Cr respectively .....................................................................................................4
Figure 2.2 Direct3D programmable graphics pipeline [13].................................6
Figure 2.3 RGB Color Cube [15]........................................................................7
Figure 2.4 HSL color cylinder [20]....................................................................7
Figure 2.5 HSL swatches [19]...........................................................................7
Figure 3.1 Overlay marking the edges of the runway and taxiway ....................9
Figure 3.2 Overlay with the filled runway and taxiway .....................................9
Figure 3.3 Lightness dependent overlay without offset, night scenario ..........11
Figure 3.4 Lightness dependent overlay with offset, night scenario ..............11
Figure 3.5 Linear HSL Overlay .........................................................................12
Figure 3.6 Sinusoidal HSL Overlay ..................................................................12
Figure 3.7 The dynamic overlay algorithm in the pixel shader .......................13
Figure 3.8 Difference without unit step .............................................................15
Figure 3.9 Difference with unit step using 0.05 offset .....................................15
Figure 3.10 The Dynamic Overlay Algorithm with Motion Masking ..........15
Figure 4.1 Dynamic overlay in the three different scenarios, snow, day and night. .........................................................................................................................16
Figure 4.2 Alpha blending to the left and dynamic monochrome to the right....17
Figure 4.3 Dynamic monochrome overlay without motion masking to the left and with motion masking to the right ..............................................................18
Figure 4.4 Overlay marking the edges, alpha blending to the left and dynamic monochrome with motion masking to the right .............................................18
Figure 4.5 Motion masking shadows from clouds ...........................................19
Figure 4.6 Motion masking with trail from a slowly moving object. ..................20

Figure 4.7 Partly masked moving object. ....................................................20
1 Introduction

The digitalization brings many benefits to the security and surveillance business. The use of cameras and remote-control centers allows a more flexible and cost-effective solution. Live video data from cameras can be analyzed and used in algorithms to inform the user of anomalies, which decreases the risk of human error.

This bachelor thesis is written as a part of the Computer Engineering program at Linnaeus University and ordered by SAAB – Air Traffic Management – Remote Tower, in Växjö. This report aims to come up with a recommendation on how to dynamically, based on live video, render overlay which marks the airfield runways without losing the information of the background video.

This chapter will explain the background of the Remote Tower, the problem description, and method which were used to solve the problem.

1.1 Background

The initial idea for the remote tower came from the increasing concern regarding the cost of operating the airports, especially small airports with few movements. SAABs solution is to digitally manage airports with a camera house at the airport and supply the Air Traffic Service (ATS) from a Remote Tower Module (RTM). The camera housing is placed on a high mast or building containing up to 14 vertically positioned cameras covering 360 degrees horizontal and +/- 23 degrees vertical view, representing the Out of Tower Window view (OTW), which is the same as the ordinary towers. On top of the camera house, there is a Pan Tilt Zoom (PTZ) camera which lets the Air Traffic Controller (ATC) zoom in anomalies on the airfield. The RTM represents the OTW with 14 monitors (one camera per monitor) in a flexible formation, and have a dashboard containing radar, weather, PTZ controls and other essential tools needed for ATS.

![Figure 1.1 Remote Tower Module with two positions](image)
This digital approach enables SAAB to use overlays to clarify details and display additional information on the monitors. When this report is published, overlays are used for marking the airfield runways, showing the name of the taxiway, the runway number, and the point of the compass. The problems with SAABs approach are as follows:

- It’s hard to find a good balance between the visibility of the overlay and the underlying video stream. Either the overlay is clearly visible and covering the underlying video, or the underlying video is clearly visible and the overlay barely visible.
- The method is static and requires that the ATC manually changes the opacity of the overlay to find a good balance between the visibility of the overlay and the underlying video. The ATC must make this change multiple times during the day as the brightness of the video changes. It requires less opacity to have a visible overlay when the video is dark than when the video is bright, which results that a pleasant setting for overlay in daylight scenario can appear disturbing in night scenario since the contrast between overlay and video is significant.

1.2 Expected Results

The expected results of this report are to come up with a recommendation on how to solve the problems described above. To address these problems, SAAB wants a recommendation on two things: The first part is a dynamic overlay method that automatically gives a good balance between visible and not disturbing overlay, and in the same time allows the underlying video to be visible. The second part is to detect moving objects and choose not to render overlay where there is motion.

1.3 Method

To come up with the recommendation, different theories and technologies were studied to get an understanding of the programming languages, graphics programming and color theory. A program was built to try different methods
and algorithms, the program renders overlay on a single picture and has adjustable in-parameters which enable analysis of small changes.

When the results of the analysis program were satisfying, the next step was to try the dynamic overlay on live video. To get access to live video, SAABs research decoder was studied to get knowledge of how the dynamic overlay method could be designed and implemented into the decoder. With access to render the dynamic overlay on video, further tests, and analysis were made. After the dynamic overlay was done, motion detection functionality was implemented and tested. During the process of building these features, different methods and calibrations of the tests were represented and analyzed together with SAAB.
2 Theory and Technologies

This chapter will give an overview of the theory and technologies that are relevant to this project. The two main areas that will be described are graphics programming and color theory. The graphics programming will focus on, video encoding/decoding, the graphics programming API DirectX, and the shading language HLSL. The section about color theory will bring up the different color models that were of interest for this project.

2.1 Graphics Programming

This section will describe the theory behind the graphics programming that is used in this project. Since this project is working with live video streams from cameras, encoding and decoding will be described to get an understanding of the video data format that is used.

2.1.1 Video Encoding and Decoding

The raw video from the cameras is sent to computers which encode the video into H.264, which is a video compression format. The H.264 video is then sent with the Real-time Transport Protocol (RTP) to the RTM. The video is then decoded from H.264 to Y’CbCr data format and at last converted to RGB and displayed on monitors.

Y’CbCr often mentioned as YUV is a color encoding system used in digital video. Y’CbCr takes the human perception into account since the human eye is more sensitive to certain wavelengths. Y’ is the Luma component which is the brightness representation of the image (black and white). The Cb and Cr are blue and red chroma values representing the color relative to the Luma values [2].

Figure 2.1 Y’CbCr frame to the left followed by the representations of Y’, Cb and Cr respectively
2.1.2 DirectX

DirectX is the multimedia collection of APIs which are used for developing applications on Microsoft platforms. It contains many different components, but only Direct3D is of interest for this project.

Direct3D is the 3D graphics API from the DirectX collection, which is used to render real-time 3D graphics for games and applications. Direct3D is used when performance is essential and uses hardware acceleration in the graphics pipeline if the graphics card supports it. [3]

The Direct3D 11 graphics pipeline is divided into different programmable stages, see Figure 2.2. The first stage, Input-Assembler stage reads the primitive input data, such as points, lines, and triangles, and applies a primitive type which will be used by other pipeline stages [4]. The Vertex Shader stage does operations on the vertices such as transformations and vertex lighting [5].

The next three stages are a part of the tessellation pipeline of the GPU. The Hull-Shader stage allows programming a geometry patch based on each input patch. The Tessellator stage divides the geometry from the hull-shader into smaller objects that represent the input geometry, which gives more detail to the object. The Domain-Shader stage computes and returns the actual vertex positions based on the previous tessellation stages [6]. The Geometry shader uses input vertices to generate geometry and return those vertices [7]. Those stages are optional and are mainly used to get more detail without increasing the memory or bandwidth and moving processing from the CPU to the GPU [8].

The Stream-Output stage is used to stream or output the vertex data from the vertex shader or geometry shader to the memory. This enables the vertex data to be reused in the pipeline or accessed by the CPU [9]. The Rasterizer stage takes the vertex data and converts it into pixels, here the transformation from 3D to a 2D representation is made [10]. The Pixel Shader stage allows calculations on each pixel. Sample textures, using constant variables and other manipulations on a single pixel is done here [11]. The Output-Merger stage merges different pipeline states and returns the final rendering result. For example, it takes the data from the pixel shader stage and combines the contents of the render targets and depth stencil buffers to determine if the pixel is visible. The Output-merger stage also applies blend state. Blend state allows combining the color and alpha that has already been rendered with the previously rendered color and alpha. For instance, to render fog in front of terrain, alpha blending is used to adjust the transparency of the fog which determines how visible the underlying terrain is [12]. More detailed information about the Direct3D graphics pipeline can be found at the Microsoft resource [13].
Microsoft has developed a programming language which is used to program the graphics pipeline. The language is called High-Level Shading Language (HLSL). HLSL is C like, and there are some strategies to consider when developing shader algorithms. Since the pixel shader algorithm runs for each pixel, and the vertex shader runs for every frame, it is strongly recommended to execute calculations in the vertex shader, if possible. HLSL comes with many intrinsic functions which are bug-free and performs well [14]. One thing to keep in mind is that in this project, sampled textures returns a value from 0.0 to 1.0 independently of the texture’s format. For example, the overlay pixels is saved as RGBA(255, 255, 255, 255) and sampled in the pixel shader as RGBA(1.0, 1.0, 1.0, 1.0).

2.2 Color Theory

When working with graphical programming, it can be useful to know some basic color theory. This section will briefly explain two color models.

A color model is a way to describe the way color can be represented mathematically. In this project, the Red Green Blue (RGB) and Hue Saturation Lightness (HSL) models are used and will be described below.
2.2.1 RGB

The RGB color model contains three color components, red, green and blue. The values of the color components represent the light of the color which is then added to each other resulting in the final color. The RGB colors are often represented by a cube, see Figure 2.3.

![Figure 2.3 RGB Color Cube](image)

The RGB color model is used in many different technologies, such as the electronic displays, for example, LCD, plasma, LED and more. RGB is also the color standard for HTML. [16]

2.2.1 HSL

The Hue Saturation Lightness color model is a cylindrical-coordinate color model. Hue is the angular dimension of the primary colors red at 0°, green at 120° and blue at 240°. Saturation is the mixture of the hue and gray (also called tone) from 0 to 1, where 0 corresponds to 0% hue and 100% gray, 0.5 corresponds to 50% gray and 50% hue, and 1 corresponds to 100% hue and 0% gray. Lightness is the mixture of the hue and white or black (tint or shade), going from 0 (black) to 1 (white). When lightness (L) is equal to 0.5, there is no mixture, when L < 0.5 the hue is a mixed with black and when L > 0.5 the hue is mixed with white [17], [18]. See Figure 2.4 and Figure 2.5, where Figure 2.5 is a cropped picture from the source at [19] and Figure 2.4 is a cylindrical representation of HSL [20].

![Figure 2.5 HSL swatches](image)  
![Figure 2.4 HSL color cylinder](image)
HSL is mainly used in color picking tools since it’s compared to RGB faster and easier to find a wanted color by first adjusting the hue and then saturation/lightness in a 2D representation similar to figure 2.4. HSL is also good when working with monochrome colors [18].
3 Implementation

The study of the technologies and theories led to the main approach on which the solution was built. This chapter will describe the main approach, the different steps in building the dynamic overlay algorithm, the tests that were made on these steps leading to the final algorithm and the motion detection functionality.

3.1 Dynamic Overlay

The overlays are stored pictures, in which the airfield runways and taxiways are marked as white lines with a transparent background. Loading pictures into the graphics pipeline is done by creating 2D Textures of the picture, assign the 2D Texture to a Shader Resource View and then bind the shader resource view to the Pixel Shader. This means that in the pixel shader, both video and overlay are accessible in the graphics pipeline. With this knowledge, an analysis program was developed with the dynamic overlay algorithm in the pixel shader. For simplicity, the analysis program was built only rendering one saved frame from three different scenarios. Later when the dynamic overlay algorithm was satisfying for one frame and the knowledge of Direct3D was increased the algorithm was implemented into SAAB research decoder, which will be described further down in this chapter.

Originally the overlays marked the edges of the runways with a small line, but to be able to make analyzing different methods and settings easier, the overlays were configured to fill the runways and taxiways, as shown in Figure 3.1 and Figure 3.2. The filled overlays also turned out to be an interesting approach for the result and are discussed in chapter 4.

![Figure 3.1 Overlay marking the edges of the runway and taxiway](image1.jpg)

![Figure 3.2 Overlay with the filled runway and taxiway](image2.jpg)
3.1.1 Analysis Program and Dynamic Overlay Algorithm

The analysis program was built by first initialize all the necessary Direct3D components. To render the video frame, two triangles were created in the vertex shader which represents a rectangle. The Y’CbCr textures were then bound to the pixel shader. In the pixel shader, the Y’CbCr data was sampled and converted to RGB which was then rendered to the monitor. The Y’CbCr conversion into RGB was provided by SAAB. Once the program could render a video frame, the development of the dynamic overlay algorithm in the pixel shader started, using the shader language HLSL. The algorithm was built in multiple steps leading to its final state.

The first step was to add the overlay texture to the pixel shader and find a solution to identify if there is painted overlay on the current pixel. To determine if there is overlay on the current pixel, the overlay textures alpha value is sampled, and since the background is transparent, this means that where there is no overlay, the alpha is equal to zero. An IF statement could be used to check if the value of the overlay alpha is bigger then zero. Since IF statements are costly for the graphics card, linear interpolation was used instead. The first implementation for this was to manipulate the Cb and Cr values where the overlay texture had a high alpha. The linear interpolation function is:

\[
(1 - \alpha) \times value1 + (\alpha \times value2)
\]

Equation 3.1 Linear Interpolation with factor alpha

Value1 represents the original video pixel color, and value2 represents the chosen color where there is overlay. The alpha value will be either 0.0 or 1.0 except if there is any antialiasing. This implies the following three different scenarios: first, where there is no overlay, value1 will be returned, second, where there is antialiasing, a mix of value1 and value2 will be returned, and third, where the overlay is solid, value2 will be returned. To control that this worked in practice, the Cb and Cr values were set using the linear interpolation function, in which value1 was the sampled value and value2 was set to 0.5. As expected, the result was that where there is overlay the color was black and white.

The next step in building the algorithm was to manipulate each pixel where there is overlay to a monochrome color dependent of the videos light (Y’ luma) value. After some study about color models, HSL turned out to be the color model best suited for this purpose. The reason HSL was best suited for monochrome representation in this project is that for any hue, the resulting color can be between completely black to white and all other representations of the chosen hue. This gives a good contrast between underlying objects. See the HSL swatches Figure 2.5 again which illustrates the different representations of hue=30% and hue=210%.
To test and calibrate the HSL values, analysis functionality was implemented into the application. Three different scenarios were added: Night-, Snow- and daylight-scenario. Buttons to change the hue value, saturation value, scenario, and manipulation of the overlay alpha was added which made it possible to analyze different settings and scenarios in real-time.

Different functions on how the lightness should be dependent on the video light ($Y^*$ luma) value were tested and had changeable in-parameters from the application. The different light-dependency functions that were tested were: linear dependency with an offset and sinusoidal dependency with offset see Figure 3.5 and Figure 3.6. The reason why there is a need for the offset became clear when trying the different lightness-dependency functions on the night-scenario. During the night the runways become completely dark, and since the HSLs lightness value at zero represents black, the overlay wasn’t visible during the night. Therefore an offset with 0.1 was added.

The sinusoidal light-dependency function was tried to get more contrast between moving objects and the background, but since information and detail were lost by making the lighter parts lighter and the darker parts darker the linear light-dependency function was chosen. The last step in the algorithm is to translate the monochrome HSL color into RGB and return it from the pixel shader. This was done using a conversion algorithm that was taken from [21] which works well.
3.1.2 Implementation into the video decoder

When the dynamic overlay looked satisfying on the three different scenarios, the next step was to try it on video to see how it looks when there is moving objects behind the overlay. Since it would be too much work to implement this into the analysis application, the dynamic overlay and analysis functionality was implemented into SAAB research video decoder instead. SAABs video decoder takes all the 14 cameras video and builds a panorama view. This decoder also has features such as zoom and rotation, which were helpful when comparing different approaches and settings.

Implementing this into the video decoder was done by building an overlay service. The service supplies every camera with the corresponding overlay texture. This works as when a video resource (one per camera) gets a new frame; it checks if the overlay texture is already loaded. If not, the video resource sends a function call to the overlay service with the cameras unique id as in-parameter, the overlay service finds the corresponding overlay texture and returns it to the video resource, which is then rendered. The service also listens to the keyboard commands which updates the different analysis settings mentioned in the previous section. Finally, the pixel shader was swapped to the one developed in the single frame analysis program.

Rendering the dynamic overlay on live video was much more convenient than rendering it on a single frame. Analyzing the method when moving objects were interacting with the overlay, illustrated an operative scenario. Moving objects of different colors such as, white, black and yellow were analyzed and led to the final overlay algorithm, which can be seen in Figure 3.7.

![Figure 3.5 Linear HSL Overlay](image1)
![Figure 3.6 Sinusoidal HSL Overlay](image2)
3.2 Motion Detection

When the results of the dynamic monochrome overlay were satisfying, research started on how the motion detection could be achieved. Searching the web, many examples of motion detection and object tracking was using the open video processing library OpenCV. OpenCV was fast concluded to be too expensive to use on 14 full HD video cameras, with the frequency rate at 60 frames per second. Instead, an idea was developed on how to make motion detection from scratch. The paper [22] gave inspiration to the initial plan to have a background texture on which the difference of the current frame would be calculated and used to identify motion.

The background texture would have to be updated continuously to follow the weather changes and so that moving vehicles that parks would blend into the background texture after a while. A background texture was created for each camera and for every new frame a small percentage of that frame was added to the background texture.

To achieve this, render target views which points to the background textures was created. For every new frame, the render target had to be changed from the render target which points to the monitor, to the render target which points to the background texture. The next step was to build a simple pixel shader which samples the new frame. After the frame has been sampled, a blend state using factor blending was applied. The factor blending which is a part of the output-merger state was set to take a factor of the background texture plus the inverse factor of the sampled frame which was then rendered to the texture. The blending factor can be set from 0.0 to 1.0, see the equation below.

\[ bgTex = factor \times bgTex + (1 - factor) \times sampledFrame \]

Equation 3.2 Background, blend state equation

Choosing the blend factor determines how much the background texture updates for each frame, and since the background texture should not contain any moving objects the factor had to be set close to 1.0. After testing different
values on the blend factor, the value 0.997 was chosen, which means that the background texture is set to 99.7 percent of its old value plus 0.3 percent of the current frame, for each new frame.

This functionality was implemented so that each video resource contained an instance of the motion detection class, and when a new frame arrived, the video resource class calls a function on the motion detection class to update the background texture with the new frame as in-parameter. The updated background texture was then returned to the video resource which bound the new background texture to the pixel shader.

The pixel shader that before only rendered manipulated monochrome color based on the overlay texture, now had to be adjusted to also take the difference between the background texture and the current frame into account. Since it is costly to sample textures and the background texture only needs to be sampled where there is overlay, an if-statement that determines if there is overlay was implemented. The HSL calculations and conversions were moved to the body of the if-statement. The background texture was sampled, and the difference in light between the background pixel and the current pixel was calculated to determine if there is motion. Instead of using linear interpolation based on the overlay alpha, seen in Equation 3.1, linear interpolation was used, based on the difference between the background texture and the current frame. The linear interpolation still determines whether the color of either, the monochrome overlay or the original video should be rendered, see Equation 3.3.

\[
(1 \text{ – difference}) \times \text{overlayColor} + (\text{difference} \times \text{videoColor})
\]

*Equation 3.3 Linear interpolation between monochrome overlay color and video color*

The difference could be any value between 0.0 and 1.0. This resulted that the linear interpolation didn’t work as wanted since a relatively small difference could indicate that a dark object is moving above a dark background. For example, if a black helicopter is approaching asphalt the light difference could be 0.1, which would make the linear interpolation to return 0.9 * monochrome overlay color + 0.1 * video color when the wanted result is 0.0 * monochrome overlay color + 1.0 video color. To solve this problem the difference was used in a unit step function with an offset, the offset determined if the difference was significant enough to be counted as a movement.

This resulted that for all the difference values that were greater or equal to the offset were set to 1.0 and the value less than the offset was set to 0.0. After trying different offsets, an offset with 0.05 was chosen.
As seen in Figure 3.8 and Figure 3.9 the difference is increased which causes the linear interpolation to completely mask out the airplane. Figure 3.9 also displays difference above the airplane. That difference comes from trees and clouds which is not a problem since the difference is only used at the runways. The final pixel shader algorithm resulted as described in the pseudo code below.

```
> Sample overlay texture alpha
> Sample Y' Luma texture
> Sample Cb and Cr texture
> RGB = Convert Y'CbCr to RGB
> if(overlay alpha is greater than zero){
>   Define HSL with linear dependency of Luma with offset 0.1
>   Convert HSL to RGB
>   Sample background texture
>   Calculate difference between background and Luma
>   Set difference values that’s equal or greater than 0.05 to 1.0
>   RGB = Linear interpolation between the RGB values of the monochrome overlay and video frame based on the difference.
> }
> Return RGB
```

Figure 3.10 The Dynamic Overlay Algorithm with Motion Masking
4 Results and Discussion

In this chapter, the results will be presented and discussed. The results will be presented by displaying different screenshots from the different methods, which are, alpha blending, dynamic monochrome overlay without the motion masking, and the dynamic monochrome overlay with motion masking.

4.1 Dynamic Overlay

The dynamic overlay method was created to solve mainly two problems. First, take away the need for the air traffic controller to manually adjust the overlay to get a comfortable contrast between overlay and video. Second, find a good balance between both, visible overlay and visible underlying video.

Figure 4.1 shows how the dynamic overlay is presented in three different scenarios, snow, day and night.

![Dynamic overlay in three different scenarios](image)

Comparing the different scenarios shows that the overlay automatically follows the lightness of the video, this can reduce or even remove the moments where the ATC would have to manually change the appearance to get a pleasing appearance of the overlay.

Figure 4.2 shows four pictures that illustrate the difference in visibility of the alpha blending method (to the left) and the dynamic monochrome method (to
the right). The two upper screenshots are the same pictures as the lower but are zoomed in on the airplane.

![Figure 4.2 Alpha blending to the left and dynamic monochrome to the right.](image)

As seen in the figure, the alpha blending method covers the underlying video resulting that information is lost, whereas the dynamic monochrome method manipulates the video in a way that the lightness information from the video is kept and only the color information is lost. Comparing the different methods shows that the airplane is more visible using the dynamic monochrome method than using the alpha blending method.

4.2 Motion masking

The motion detection and masking were implemented to make moving objects even more noticeable by keeping the video color of the moving object.

Figure 4.3 shows the difference when motion masking is used (to the right) and not used (to the left).
As seen in the figure the airplane is masked out which results that the original video color is kept. This leads to a more significant contrast between airplane and overlay which makes the airplane much more visible.

Since the overlay was changed to cover the runway and taxiway instead of only marking the edges, a result of the original overlay textures is also presented in Figure 4.4.

In my opinion, comparing the dynamic monochrome method using motion masking in Figure 4.3 and Figure 4.4, both the overlay and the airplane is more visible in Figure 4.3. The overlay with filled runways has a higher contrast to the moving objects and their surroundings, which makes them easier to spot.
4.3 Limits and Areas of Improvement

Because of the limited time this project had, there wasn’t enough time to solve the limits that will be presented in this section.

4.3.1 Lightness Offset

One disadvantage with the monochrome HSL overlay is the need for an offset to see any overlay during the night. This results that the monochrome HSL color does not cover the whole lightness spectra. With information about the average video light for the entire frame, the offset could be adjusted only to be set when the average light is low. See Figure 3.3 and Figure 3.4 which shows the difference with and without an offset in night scenario.

4.3.2 Masking Shadows

The motion masking has some limits for improvement. Since it only detects motion, shadows from moving objects are masked out. Especially shadows from clouds are a problem because it can redirect the users focus, see Figure 4.5.

4.3.3 Trail from Slowly Moving Objects

The motion masking method has a limit when it comes to objects that move slowly. Slowly moving objects are partly blended into the background texture. This results in light-difference, even after the object have left the area, which then produces a trail behind the moving object, see Figure 4.6.
This limitation could be solved by updating the background texture at a lower frequency. Since there is a problem with shadows from the clouds, as seen in Figure 4.5, compromises were done between fast blend in the shadows from the clouds into the background and the appearance of a trail behind slowly moving objects for a short time.

4.3.4 Masking objects

The motion masking has a limit when the difference in light of moving objects and background texture is small, see Figure 4.7. Changing the unit step function to include minimal differences would make the motion masking to include small differences that do not correspond to a motion which would result in flicker.
5 Conclusion

As seen in chapter 4, the monochrome overlay adapts to the lightness of the video, and the underlying video is visible since the lightness information is kept and only the color information is manipulated. The motion masking makes the moving objects even more noticeable.

Another approach that would be interesting to try in future development of dynamic overlay rendering, would be object recognition. Successful object recognition would make it possible to mask out object instead of masking out movement, which would solve the limits presented in 4.3.2 to 4.3.4. Some frameworks that would be interesting to try which offer solutions to object recognition is OpenCV [23] and TensorFlow [24]. Using those frameworks would require a high-performance computer that can handle 14 full HD cameras streaming 60 frames per second.

The solution is determined by comparing different settings and scenarios. Since the different settings and calibrations result in visually changes, many decisions in this project are my subjective opinion of the optimal overlay appearance. The air traffic controllers probably have different opinions about the optimal appearance; which would not be a problem since the method is flexible and can be adjusted by implementing unique profiles for each ATC.

The overlay method developed, offers an interactive feeling and makes moving objects more noticeable. The overlay method can easily be adjusted to personal preferences, which increases the user experience.
References


