Enhanced Full-body Motion Detection for Web Based Games using WebGL

by

Oskar Havsvik

LIU-IDA/LITH-EX-A--15/045--SE

2015-06-22
Final Thesis

Enhanced Full-body Motion Detection for Web Based Games using WebGL

by

Oskar Havsvik

LIU-IDA/LITH-EX-A--15/045--SE

2015-06-22

Supervisor: Aseel Berglund
Examiner: Johan Åberg
Abstract

By applying the image processing algorithms used in surveillance systems on video data obtained from a web camera, a motion detection application can be created and incorporated into web based games. The use of motion detection opens up a vast field of new possibilities in game design and this thesis will therefore cover how to create a motion detection JavaScript module which can be used in web based games.

The performance and quality of the motion detection algorithms are important to consider when creating an application. What motion detection algorithms can be used to give a qualitative representation without affecting the performance of a web based game will be analyzed and implemented in this thesis. Since the performance of the Central Processing Unit will not suffice, WebGL and the parallelism of the Graphical Processing Unit will be utilized to implement some of the most recognized image processing algorithms used in motion detection systems. The work resulted in an application where Gaussian blur and Frame Subtraction were used to detect and return areas where motion has been detected.
Contents

1 Introduction 1
  1.1 Background and Motivation .................................. 1
  1.2 Aim .................................................................. 2
  1.3 Research Questions ........................................... 2
  1.4 Delimitations ...................................................... 3

2 Theory 4
  2.1 Background Subtraction ........................................ 4
    2.1.1 Pre-processing ............................................. 5
      2.1.1.1 Temporal Smoothing ................................. 5
      2.1.1.2 Spatial Smoothing ................................. 6
      2.1.1.3 Managing Illumination Variations ............ 7
    2.1.2 Background Modelling ................................... 9
      2.1.2.1 Non-recursive Techniques ....................... 10
      2.1.2.2 Recursive Techniques .............................. 12
    2.1.3 Foreground Detection ................................... 12
    2.1.4 Data Validation ............................................ 13
      2.1.4.1 Morphological Filters ............................ 14
  2.2 Texture-based Methods ....................................... 14
  2.3 Motion Detection on the GPU ............................... 15
3 Method

3.1 Feasibility Study
3.1.1 Frameworks Used
3.1.2 Performance and Quality Analysis

3.2 Implementation
3.2.1 Texture Computations in GLSL
3.2.2 Filter Test Program
3.2.3 WebGLMotionCam.js
3.2.4 Filter Implementation
3.2.4.1 Pre-processing Implementation
3.2.4.2 Background Modelling Implementation
3.2.4.3 Foreground Detection Implementation
3.2.4.4 Data Validation Implementation
3.2.4.5 Other Image Processing Filters

4 Result

4.1 Feasibility Study
4.2 Implementation

5 Discussion

5.1 Results
5.2 Method
5.3 The work in a wider concept

6 Conclusion
Chapter 1

Introduction

This chapter will explain the reasons why this thesis was carried out and what its goal was. A brief introduction on how motion detection is currently used and why it is relevant to compute motion detection algorithms on the Graphical Processing Unit is also given.

1.1 Background and Motivation

Full-body interaction is the use of the body as an interaction device by using a camera and computer vision to identify the body movement. In gaming for example, this is most recently approached by the Kinect Sensor by Microsoft for the Xbox 360 and Xbox One game consoles [1] and Windows Personal Computers. However, by using an application programming interface, streams such as video and audio can be accessed, which allows body motion to be detected by a standard web camera. This leads to a broader market for body motion based games since the majority of households contains a web camera, either it is on a laptop or a smartphone. The way a full-body interaction game is played differs from the usual way to play a game. When using body movements as the control, the player exercise while being immersed in the game, because it is often more intuitive to use the body for interaction. Having this in mind, developers can create games that is both fun and helps the user be physically active. K. M. Gerling et al. [2] explains that full-body motion-control games helps older people to remain active and engaged and also keeps their emotions and physical health favorable.

The problem with detecting moving objects with computer vision systems
has been a big field of study for a while now. It is used in camera surveillance, automatic light control and traffic monitoring for instance. When using computer vision to identify moving objects it is required to separate them from the background with high precision. Illumination changes, camera noise and shadows are some of the problems a robust motion detection application needs to solve and still have the ability to accurately detect changes of small magnitude [3]. By understanding the possible techniques to overcome these problems in real time, decisions on how to create a motion detection system that do not affect the performance of a web game can be made.

It is common that today’s web applications render its graphics using the canvas renderer, thus doing its computations primarily on the Central Processing Unit (CPU). This affects the performance of the application when numerous sprites needs to be drawn or extensive calculations are computed. By rendering the application using the Web Graphics Library (WebGL) [4] instead, some portions of the computations can be relocated from the CPU to the Graphics Processing Unit (GPU) instead, which results in more efficient applications. Due to the parallel structure of the GPU, it is very well suited for operations that can be done in parallel. When processing images, a great deal of per-pixel computations like two dimensional convolutions are needed and are therefore greatly efficient to be computed on the GPU. Motion detection is mainly computed with image processing algorithms and will therefore give room for more demanding and precise algorithms when computed on the GPU.

1.2 Aim

The purpose of this thesis is to analyze different techniques to be able to detect motion with as high accuracy as possible without affecting the performance of a web based motion game.

1.3 Research Questions

The problem statement in this thesis is the following:

- What motion detection algorithms can be used to give a qualitative representation of video data with a minimum impact on the performance of a web based game?
1.4 Delimitations

The compatibility of WebGL in modern browsers was not taken into account during the thesis. Although most browsers are now compatible with WebGL, browsers such as mobile Internet Explorer and earlier versions of mobile Safari are not. Since the project is focused on web based games, the motion detection module will not necessarily work with offline games.

The literature provides numerous different motion detections to cover, thus only some of the relevant techniques will be presented and implemented in this thesis.
Chapter 2

Theory

To identify moving objects in a video sequence is important for systems that maintain surveillance cameras and other security applications. The ability for computer vision to monitor suspicious and illegal activities in both the short and the long run is a critical task for security applications to accomplish [5]. The high demand for better motion detection algorithms have resulted in a big research area where there have been presented numerous of ways that are trying to solve the problems of detecting motion in a video sequence.

This section will briefly go through the keywords related to motion detection and some of the techniques found in the literature. The main method, Background Subtraction, will first be described. It is followed by a summary of a couple of texture based methods. Finally, an explanation of why doing motion detection algorithms on the GPU should have a valuable result is given.

2.1 Background Subtraction

One of the most common approaches to detect moving objects in a video sequence is the Background Subtraction [6]. It works by having a background and foreground model that are compared with each other. If the resulting pixels from subtracting the foreground model from the background model exceeds a certain threshold, the pixels are classified to represent a moving object [6]. The foreground model is described as the current frame in the video sequence. The background model, or “reference image”, is the representation of the current scene. It needs to adapt to not be affected by
changes in illuminance conditions and static objects to successfully subtract the current frame [7]. It is therefore important to maintain the background model, also called background modelling.

The ignorance of the information acquired from neighbouring pixels when creating the background model is one of the main limitations the Background Subtraction algorithms suffers from, which tends to create noise in the final image. This results in numerous of false classifications of motion detected. It also causes false classification of background pixels since there exists cases where there is a small value difference between the foreground pixel and the background pixel. [3]

Most of the current techniques for Background Subtraction follows a flow diagram [6]. The flow diagram consists of four steps; Pre-processing, Background Modeling, Foreground Detection and Data Validation. The different steps is explained in the following four main subsections.

![Flow Diagram](image)

Figure 2.1: An image that illustrates the flow diagram of the Background Subtraction process [6].

### 2.1.1 Pre-processing

The first part of the Background Subtraction process is used to filter irrelevant changes in the video sequence, thus preventing them from influencing the moving object decision later on. Camera noise and environmental noise such as rain and snow are some examples of said changes. These changes are often removed by using temporal or spatial smoothing. Lowering the rate of data can be done in the Pre-processing stage by reducing the frame-size or frame-rate and is often important for systems with performance restrictions. Some of the Pre-processing algorithms used in the field are described in short below. [3]

#### 2.1.1.1 Temporal Smoothing

Temporal smoothing of an image implies combining a reference image with the current image [8], hence using the input from a previous frame. The combination process is done on a pixel by pixel basis and can be described with the following equation:

\[
Combined_p(t + \Delta t) = G \ast Combined_p(t) + (1 - G) \ast Current_p(t)
\]

(2.1)
The gradient $G$ in equation (2.1) is the temporal smoothing gradient and should be equal or lesser than one. The higher its value is, the lesser it will use the contribution from the current image. The $t$ is the time and $\Delta t$ the time interval between the frames.

### 2.1.1.2 Spatial Smoothing

A way of using spatial smoothing is to average each pixel with its neighbours, which results in a blurred image. By increasing the lower frequencies, sharp edge’s effect will decrease. The median of all the neighbouring values can also be used for smoothening of a gray video sequence [5]. A. Rosenfeld [9] explains that when the image noise consists of only isolated points and lines that have another contrast level, the pixel containing noise can be interpolated by replacing it with the average gray value of its neighbours. The neighbourhood size determines the degree of noise reduction. When blurring images, unwanted blurring of edges and lines that actually represents objects in the scene needs to be taken into account for. Rosenfeld A [9] presents a solution for this problem; by only averaging at selected points or with selected neighbours the averaging will never occur on, for example, edges. Noise is less salient when it is occurring around such features, thus it will not affect the result as much when the selected points are not averaged.

D. Rákos [10] explains the use of the convolution algorithm *Gaussian blur* for blurring images. The Gaussian algorithm is used to calculate the weight which a neighbour should contribute with to the current pixel. The weights can be achieved by using the following function:

$$G(x, y) = \frac{1}{2\pi\sigma^2} e^{-\frac{x^2+y^2}{2\sigma^2}}$$

(2.2)

The $\sigma^2$ is the variance. Since fetching texture pixels can be an expensive operation, the number of texture pixel fetches needs to be as low as possible. The Gaussian function has convenient property of being able to separate in the x- and y-direction [11]. By using two 1D convolutions instead of one 2D filter, the number of texture pixel fetches are reduced and the computations will be more efficient. The two separated functions are shown below.

$$G(x) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{x^2}{2\sigma^2}}$$

(2.3a)

$$G(y) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{y^2}{2\sigma^2}}$$

(2.3b)
D. Rákos then presents a technique to optimize the computations even further by using linear sampling. The GPU’s fixed function hardware have bilinear texture filtering, thus instead of one texture pixel fetch, two texture pixels information can be fetched at the same time by using a computed offset and weight instead of the neighbour texture pixel’s center position. This reduces the number of texture fetches even more. To acquire the new offset and weight he uses the following two equations:

\[
W_L(T_1, T_2) = W_D(T_1) + W_D(T_2) \quad (2.4a)
\]

\[
O_L(T_1, T_2) = \frac{O_D(T_1) * W_D(T_1) + O_D(T_2) * O_D(T_2)}{W_L(T_1, T_2)} \quad (2.4b)
\]

The equations (2.4a) and (2.4b) uses two texture pixels \( T_1 \) and \( T_2 \) to create the linear sampled weight \( W_L(T_1, T_2) \) and offset \( O_L(T_1, T_2) \). The weights of \( W_D(T_1) \) and \( W_D(T_2) \) are acquired using equation (2.3a) and (2.3b), respectively.

### 2.1.1.3 Managing Illumination Variations

The problems with illumination variations are frequently dealt with in this stage hence it may cause inaccurate detection of movement. When a pixel’s intensity value suddenly changes, by turning on the lights for instance, the motion detection will wrongly distinguish a difference between the foreground and the background. I Vujović et al. [12] introduces the MBMD algorithm to solve this problem. The MBMD (memory-based motion detection) algorithm is using temporal averaging by loading a buffer of consecutive frames, then the frames are subtracted from a reference frame. According to if there is a difference between the current frame’s pixel and the reference frame’s pixel or not it is labeled “1” or “0”. After all frames in the buffer have been subtracted from the reference frame their values are added and stored. If the result does not exceed a certain threshold, it is known to be a background pixel.

The problem with illumination variations can also be solved by normalizing the intensity values for all pixels in the image. A normalization procedure presented by X. Dai and S. Khorram [13] normalizes a grayscale representation of a video sequence to acquire immaculate data from cameras. It is reducing the effect acquired from varying sensor system responses, whether conditions or atmospheric. The procedure works on data that have a Gaussian distribution and is described with the following equation:

\[
P_n = \frac{P_{old} - \mu_{old}}{\sigma_{old}} * \sigma_{ref} + \mu_{ref} \quad (2.5)
\]
The $P_n$ in equation (2.5) is the normalized pixel and $P_{old}$ is the original pixel. The $\mu_{old}$ are the mean of the source image and $\mu_{ref}$ the mean of the reference image (background image). The $\sigma_{old}$ are the standard deviation of the source image and $\sigma_{ref}$ is the standard deviations of the reference image.

The selection of color or grayscale space gives different properties to work with. When using the grayscale space, a luminosity value can be acquired from all pixels, but using it makes it hard to detect motion in low-contrast areas or prevent motion triggered by, for example, shadows [6]. As described by A. Prati et al. [14], shadows are a critical problem for motion detection algorithms, since it can lead to wrongly classification of foreground objects. Object merging, shape distortion and object losses are some of the results that a cast shadow can cause. Only using the RGB space to solve this problem will often not suffice [3], since the distance between two colors in RGB space will not work as a measurement. A better measuring and comparing of color differences can be achieved by transforming the RGB values to chromaticity and intensity values [15]. A variation of the chromaticity space is shown in the figure below.

![Figure 2.2: A figure displaying the CIE 1931 xy chromaticity diagram. The image is acquired from http://commons.wikimedia.org/wiki/File:CIExy1931.png.](http://commons.wikimedia.org/wiki/File:CIExy1931.png)

To acquire the chromaticity coordinates from RGB space one can use El gammal et al.’s way of doing it [16]. They use the color coordinates $R$, $G$ and $B$ to create the following chromaticity coordinates:
The sum of the chromaticity coordinates $r$, $g$ and $b$ should be equal to one. The main advantage of using these coordinates for measuring is that shadows can be suppressed and removed from the final image. The disadvantage is that the image loses its lightness information. Cases where a person with a white shirt in front of a gray wall will therefore not be detected. To solve this problem El gammal et al. use the lightness measure $s = R + G + B$ as the third coordinate instead. The real-time system Pfinder [17] uses a similar technique to determine if self-shadowing and cast shadows occur. A pixel’s intensity that becomes darker without much chromaticity change can thus be assumed to be affected by a shadow.

S J. Mckenna et al. [18] presents a rather similar algorithm that also observe when the intensity of a pixel changes. A pixel with a significant change in intensity, increased or decreased, without much chromaticity change is assumed to be affected by a shadow. They use the information from 1st order gradients to solve the problem with similar chromaticity in the foreground and background. Gradients are created with Sobel filter in both x- and y-directions, then each pixel’s gradient are created using gradient means and magnitude variances.

### 2.1.2 Background Modelling

Background modelling is the key part of the Background Subtraction. Its main purpose is to adapt the background image to the change of background. It can mathematically be formulated as a labeling problem in a series of images [19]. The paper by K. Toyama et al. [20] presents a number of different cases (some shown below) that an ideal background modelling system should manage to avoid. Although, they also mention that no perfect system exists yet, but the more of these problems the background modelling can avoid, the better.

- **Moved objects**: Background objects can never be considered to be static forever. Objects such as a removed chair should not affect the foreground permanently.
• **Time of the day**: The sun will not always stay at the same spot, therefore the illumination will change over time.

• **Light switch**: When the light turns on/off the illumination changes in an instant and may therefore cause inaccurate information.

• **Waving trees**: The background can have natural causes to movement like moving curtains or swaying vegetation.

• **Camouflage**: Motion that should be detected can be camouflaged by having similar pixel characteristics as the background.

• **Bootstrapping**: It can not be taken for granted that an environment is free from foreground objects when the system is activated. The systems that requires a training period without foreground objects cannot manage because of this.

• **Foreground aperture**: Homogeneously colored objects may not appear entirely because change in the interior pixels cannot be detected.

S. Cheung and Kamath [6] classifies the different techniques for background modelling into two categories; non-recursive and recursive. The following two subsection will describe some of the most commonly used non-recursive and recursive techniques.

### 2.1.2.1 Non-recursive Techniques

The non-recursive techniques takes advantage of a stored buffer of previous frames and uses them to compute the temporal variation of the pixels inside the buffer. Because of this, the non-recursive techniques are very adaptable to changes in the background.

**Frame Differencing**

As mentioned in [6], this method is one of the most straightforward background modeling techniques. It uses the frame previous to the current one as the background model. The method is highly adaptive to dynamic environments but suffers from the “Foreground aperture” (mentioned in Section (2.1.2)) [5].

Y. Tian and A. Hampapur [21] presents a method for detecting slow or temporarily stopped movements with frame differencing. The method uses a weighted accumulation with a fixed weight which is obtained with the following equation.
\[ I_{acc}(x, y, t + 1) = (1 - W_{acc})I_{acc}(x, y, t) + W_{acc} |I(x, y, t + 1) - I(x, y, t)| \]  

(2.7)

The variable \( W_{acc} \) in equation (2.7) is an accumulation parameter, the \( I_{acc} \) is the weighted accumulation and \( I \) the image. If the weighted accumulation \( I_{acc} \) is greater than a given threshold, the value is replaced with 1.0, otherwise 0.0.

**Median Filter**

Uses the median of a buffer of gray scale images at each pixel as the resulting background image [6]. This assumes that a background pixel is present in at least half of the buffer images. By using the medoid alternatively, filtering of color images can be achieved.

**Average Filter**

Unlike the median filter, the average filter uses the average of the buffered images as the background image [3, 22]. It cannot handle scenes where there is a lot of motion and will recover slowly from uncovered background. The process of this filter is shown in the equation below.

\[ B(x, y) = \frac{1}{N} \sum_{t=0}^{N-1} X_t(x, y) \]  

(2.8)

The \( B \) in equation (2.8) is the resulting averaged background pixel at the coordinate \((x, y)\). The \( N \) is the number of images in the buffer and \( X \) is buffer image at the time \( t \).

**Linear predictive filter**

This technique is a way of predicting the current background by estimating the pixel’s values, it is done by computing the linear combination of the previous values. K. Toyama et al. [20] uses a one-step Wiener prediction filter to make a probabilistic prediction. By inspecting if a pixel’s value diverges from the predicted value the pixel can be classified as background/foreground. The equation used for this is shown below.

\[ S_t = -\sum_{k=1}^{p} a_k S_{t-k} \]  

(2.9)

The variables \( S_t \) and \( S_{t-k} \) in equation (2.9) is the predicted value at the time \( t \) and the past predicted value, respectively. The \( p \) is the number of past values and the \( a_k \) is a prediction coefficient.
2.1.2.2 Recursive Techniques

Recursive background modelling does not store a buffer like the non-recursive ones, hence it will not require as much space. It recursively updates the background model for every frame instead. The disadvantage is that errors in the background model will take a longer time to disappear.

**Single Gaussian**

By calculating the average image of the scene and then subtract every new image from it and thresholding the result, a very simple Gaussian model is achieved. Used with an adaptive filter, this model can adapt to gradual changes in the scene.

S. Jabri et al. [23] builds their model with the help of Gaussian distribution. In their report a background model is presented that uses both the Color and Edge information of a video sequence. The mean $u_t$ and standard deviation $\sigma_t$ are calculated for each color channel and then stored in two separate images. To acquire the edge model, a Sobel edge operator is used on every color channel. The mean computed at frame $t$ for both the color and edge model is achieved by the following equation:

\[
 u_t = \alpha * x_t + (1 - \alpha) * u_{t-1} \tag{2.10}
\]

The $\alpha$ in equation (2.10) is the learning rate of the model, the higher it is, the more it forgets about the past frames. The variable $x_t$ is the current pixel value and the $u_{t-1}$ is the previous mean. This mean value can then be used to identify pixels which have changed color and can further be used in the computation of the standard deviation $\sigma^2_t$.

\[
 \sigma_t^2 = \alpha(x_t - u_t)^2 + (1 - \alpha)\sigma_{t-1}^2 \tag{2.11}
\]

2.1.3 Foreground Detection

The third stage presented in [6] is the Foreground Detection. It compares the estimated background model with the incoming frame to identify pixels belonging to the foreground. It can be achieved by subtracting the background pixels with the foreground and then comparing the result with a given threshold. If the result is greater than the threshold, the value is classified as a foreground pixel, otherwise a background pixel.

\[
 |I_t(x, y) - B_t(x, y)| > T \tag{2.12}
\]
Equation (2.12) demonstrates the pixel $I_t$ at the frame $t$ being subtracted by the background pixel $B_t$, where $T$ is the threshold.

The Color and Edge model used by S. Jabri et al. [23] makes use of a more sophisticated technique for Foreground Detection. They process the image with an edge detector in both axes and makes use of the image’s color values. Instead of subtracting the whole current frame with the estimated background, they subtract the color and edge channels independently. Then the results are combined to a final image with the detected foreground pixels. To subtract the color channels they compare each channel’s difference with two threshold, $m\sigma$ and $M\sigma$. If the difference is lower than $m\sigma$, a confidence value is set to 0%. If it is higher than $M\sigma$ the value is set to 100%. The values between $m\sigma$ and $M\sigma$ is acquired by using the following equation:

$$C = \frac{\text{Diff value} - m\sigma}{M\sigma - m\sigma} \times 100 \tag{2.13}$$

By using the maximum value achieved from the three channels, the maximum confidence value can be used to determine if a foreground pixel is present.

To compute the edge subtraction, they defines an edge gradient by using the difference from both the $x$ and $y$ mean images for each color channel.

$$\Delta G = |H - H_t| + |V - V_t| \tag{2.14}$$

The current horizontal difference $H$ and the vertical difference $V$ is subtracted by the mean horizontal image $H_t$ and the mean vertical image $V_t$, respectively. Then the confidence values are assigned by comparing the ratio of the difference in edge strength. By using the maximum value of the horizontal and vertical image added up and the horizontal mean image with the vertical mean image added up, an edge reliability $R$ can be acquired by the following equation:

$$R = \frac{\Delta G}{G_t} \tag{2.15}$$

The edge gradient difference $R\Delta G$ is then used in the same fashion as the in color subtraction phase. The maximum value from the Color and Edge subtraction is then used as the final result.

### 2.1.4 Data Validation

The Data Validation, or post-processing phase, is used to improve the data received from the Foreground Detection phase. A series of morphological operations are often used to remove false detection of foreground pixels.
2.1.4.1 Morphological Filters

W. Burger and M. J. Burge [24] describes morphological filters as a predictable way of altering the local structure of an image. Erosion and Dilation are most basic morphological operations and can be used in succession to remove noise or holes. They are described with a structuring element matrix $H$ which contains values of 0 and 1 ($H(i, j) \in [0, 1]$).

When using dilation, the regions in a binary image will grow because it compares all possible pairs of points in the structure element $H$ with the binary image $I$. In [24] it is defined as:

$$ I \oplus H = \{(p + q) \mid \text{for some } p \in I \text{ and } q \in H\} \quad (2.16) $$

Erosion, just as it sounds, shrinks the regions in a binary image by comparing the structure element $H$ with the binary image $I$. The result is stored only when the structure element $H$ is a perfect match with $I$ at the position $p$. Erosion is defined as:

$$ I \ominus H = \{p \in \mathbb{Z}^2 \mid (p + q) \in I, \text{ for every } q \in H\} \quad (2.17) $$

By using erosion followed by a dilation an opening in the image can be attained. This implies that all positions where the foreground structures are smaller than the structure element will be removed i.e. removing isolated pixels. When using dilation followed by erosion alternately, holes in foreground structures will be closed.

MMC Pawaskar et al. [25] proposes a method for removing noise and holes and still sustain a smooth structure of the moving object. The method calculates the sum of all neighbours per-pixel. If the result exceeds 4, the value of the pixel is changed to 1, otherwise 0.

2.2 Texture-based Methods

There are some methods presented for detecting moving objects by using texture features [26, 27]. These texture-based approaches uses the texture features to obtain statistics about the background. By using a local binary pattern, explained and used in [26, 27], a great tolerance against variation in the illumination is achieved while still keeping the computations simple. Most of the previously discussed approaches only uses the pixel color or intensity for background subtraction. The texture-based approaches are computed over a large area instead of just a single pixel. M. Heikkilä and M. Pietikäinen [26] states that compared with the methods that follows the flow diagram [6] (discussed in Section 2.1), this method of doing background
subtraction has many advantages and improvements. Although, they also mentions that their method cannot handle moving shadows since it was an extremely difficult problem.

2.3 Motion Detection on the GPU

The GPU is great at handling computations with textures due to its parallel structure. The computation of numerous 2D convolutions is therefore very appropriate to be mapped on the GPU’s architecture. When calculating algorithms on the GPU using the shading language GLSL, the data information pass through some certain stages, also called shaders. The data flows between these stages where each stage has a specific inputted and outputted data [28]. The shaders most used consists mainly of two different parts; a vertex shader and a fragment shader. The vertex shader transforms all the coordinates from data into 2D space, then the fragment shader generates a color for all pixels. By passing a texture containing data to the fragment shader, per pixel operations can be achieved in parallel, thus are most of the motion detection algorithms suited to be computed on the GPU.
Chapter 3

Method

This section will cover how different Background Subtraction algorithms were approached and implemented. It will include an explanation on how a program where the motion detection algorithms can be tested was created and which frameworks that were used. A JavaScript module that uses the tested algorithms was also implemented and explained.

3.1 Feasibility Study

This section will present the frameworks that were used during the process and how the program’s performance and quality were analyzed.

Before any motion detection filters were implemented a literature study was carried out to learn which filters that works best in different situations. There are numerous of different filter techniques in the literature and the testing of new filter combinations would be endless, thus a couple of the most recognized filters were chosen. When searching for algorithms to use, keywords related to shadow detection, efficient noise removal and light variations were prioritized. Since most of the motion detection algorithms consist of multiple filter stages, a program that could efficiently change the type of filters used was required. This results in a possibility to compare the quality of different filters at run time without having to change the program.

By creating a JavaScript module, the best motion detection algorithms discovered in the filter test program could be included in all sorts of JavaScript applications. Considering that the problem statement of the thesis specifies that a motion detection that does not affect the performance of a web based
game is wanted, this was a way to incorporate and test the algorithms with an external application. It is worth noting that a motion detection module, “MotionCam.js”, written at University of Linköping, already existed and its structure was used in the creation of the new motion detection module. At the time, the motion detection computations in “MotionCam.js” were done on the CPU and used a standard frame differencing method. Since the module is not well known, the parts used will also be explained.

3.1.1 Frameworks Used

The implementation of the Background Subtraction algorithms was carried out using the web-based 3D-library WebGL by the Khronos Group. It is a JavaScript API that enables hardware-accelerated 3D and 2D computer graphics on the web and is based on the OpenGL ES 2.0 [29]. Although it only works on compatible web browser, it is getting more and more attention and is today compatible with most of the modern browsers. Like OpenGL ES 2.0, WebGL uses programmable shaders to render the final image on the screen. The 3D-library “Three.js” [30] was used to build a WebGL scene.

The “ThreeRTT.js” library by S. Wittens [31] was used to create render-to-texture effects more conveniently. The library is backed by multiple buffers and lets the user access previously rendered frames more conveniently. Because of this, future contributors of the motion detection module can add their own content without having as deep knowledge about how framebuffers work. This was one of the main reasons the library was used.

The implementation used the “dat-gui” library [32] as the graphical interface. It is a lightweight library that lets the user change JavaScript variables at run time.

Although the motion detection module was written to work with all different types of JavaScript applications, it was mainly focused to suit the framework Phaser IO by Photon Storm Limited [33]. It is a free HTML5 game framework that uses Pixi.js for WebGL and Canvas rendering on web browsers. The framework is using the language JavaScript for creating applications.

![Graphical Processing Unit](image)

**Figure 3.1:** Flowchart of the graphic process.
3.1.2 Performance and Quality Analysis

The performance and quality needs to be compared between the different algorithms. It is important to note that the fastest motion detection algorithm is not necessarily the most suitable for web based games. A good trade-off between speed and quality is required, thus both fields needs to be evaluated simultaneously. The performance of the different algorithms will be evaluated on how long it takes for a frame to be rendered and will be referred as frame time. How the quality of the motion detection algorithms should be measured is a common problem [3]. This thesis will use the human eye to measure how acceptable the feedback from the motion detection is. Although this measurement is very subjective and may be a time-consuming process [34], it could result in the truthfully experience of the motion detected.

3.2 Implementation

This section will first briefly introduce some aspects of texture computation. It is then followed by two sections that explains the structure and implementation of the filter test program and the motion detection module. The module will be referred to “WebGLMotionCam.js” henceforth. Lastly, an explanation on how the different motion detection filters were implemented is given.

3.2.1 Texture Computations in GLSL

GLSL uses textures to compute and it is worth mentioning that texture coordinates always goes from 0.0 to 1.0. The textures is drawn on a full screen quad consisting of two triangles built with 6 vertices. A figure of the full screen quad is shown below.

To access neighbouring texture pixels an offset needs to be calculated. Considering the texture coordinates goes from 0.0 to 1.0, the horizontal neighbour offset is calculated by dividing 1.0 by the texture’s width. The vertical offset is calculated roughly the same but is using the height instead of the width. By rendering the screen to a framebuffer, a series of effects can be applied and it is further explained in the next sections.
3.2 IMPLEMENTATION

Figure 3.2: A figure showing the vertex coordinates for a full screen quad that will be used for rendering the textures.

3.2.2 Filter Test Program

In the initialization process of the program, a default “Three.js” scene and camera were created. A video and canvas were then created to be able to capture the video stream from the webcam and was done using the `navigator.getUserMedia`. Since all web browsers do not support the `navigator.getUserMedia` functionality, the video stream fetch was used in an if-else statement that returned an error message if it was not supported. There were also some diversity on how the source of video should be retrieved. Browsers like Opera uses the video stream as it is, but most browsers require `window.URL.createObjectURL(stream)`. A “Three.js” `DataTexture` that would hold the current video data was then initialized. The width and height of the texture was set to the current width and height of the canvas.

To be able to represent the four different Background Subtraction stages from the flow diagram described in [6] the code was divided into four parts, where each part would have a resulting variable from the currently used filter. Each stage process the texture and then pass its result to the next process stage. By dividing the code in this manner, it was convenient to change which filter that should be used at each stage since it could adjust which resulting variable the following stage should use for processing.

To incorporate the filters, “ThreeRTT.js” was used to generate textures which the filters had processed. Each time an additional filter should be used, a new `ThreeRTT.Stage` was created. By passing the current renderer, screen width and height and the wanted buffer of frames to store the “ThreeRTT” created an off-screen texture which held the current canvas data. Since the data needed a fragment shader to process the data, a variable containing a `ThreeRTT.FragmentMaterial` was initialized. It used a `ThreeRTT.Stage` and a fragment shader to render the processed result to a
resulting texture. Considering that a \textit{ThreeRTT.Stage} needs to be rendered to produce a processed texture, rendering of unused stages was not wanted. The initialized \textit{ThreeRTT.Stage} was therefore added to an object containing an additional variable which decided if the stage should be rendered or not. The object was then added to an array which was iterated in the rendering loop.

When all stages had been processed, the final texture was stored in the fourth stage’s resulting variable. By using \textit{ThreeRTT.Compose}, the texture was added to the scene.

Some filters required the past rendered frames to be able to compute. By using \textit{read(-i)} on a rendered \textit{ThreeRTT.Stage}, the previously rendered frame \(i\) was acquired. The chosen \(i\) cannot be larger than the size of stage’s frame buffer.

A graphical user interface was created using the “dat.gui” library. By adding variables and options, the user could change the internal values of the fragment shaders and which filters that should be rendered.

The texture variable containing the data from the video stream was updated in the render loop. Before updating the variable, the video needed to have a required amount of data since the render loop may update faster than the webcam can provide new data. The \textit{readystate} of the video needed to satisfy the \texttt{HAVE_ENOUGH_DATA} state to be able to proceed. By using the function \textit{drawImage} with the current video element, the current video context was updated. A texture variable was then updated with the current context of the video.

3.2.3 WebGLMotionCam.js

The module was initialized in a similar approach as the filter test program. A canvas and video element were created with a set width and height. The “Three.js” camera, scene and renderer were then initialized. Since the module will produce an image that will consist of numerous “unused” pixel values, the alpha property of the renderer was required to be true. The opacity of the overlay could not be adjusted otherwise. The pixel values of the rendered texture were needed to be read later on, hence the renderer’s property \texttt{preserveDrawingBuffer} was required to be set to true.

A function named \texttt{start(cameraAnchorX, cameraAnchorY, cameraWidth, cameraHeight)} was then created with the \texttt{cameraWidth} and \texttt{cameraHeight} as the wanted width and height of the webcam texture. The \texttt{cameraAnchorX} and \texttt{cameraAnchorY} are the position which the webcam texture should be located on the screen. By setting the \texttt{cameraAnchorX} and \texttt{cameraAnchorY}
to 0 and the $cameraWidth$ and $cameraHeight$ to the current width and height of the application, a full screen webcam texture is displayed. These values can also be changed at run time to acquire a transforming webcam texture. Because the camera resolution and the resolution of the application did not have to be equivalent, the values were used to transform the application coordinates to the camera resolution. The function also initializes all of the chosen filter computations and fetches the webcam’s video stream in a similar procedure as in the filter test program.

The module’s and the filter test program’s update function was implemented in an almost identical manner. Since the module needed to determine where in the texture movement had occurred, a `readPixels()` function was added. The function reads the pixels of the current renderer context and store the result in an `uint8` array. This array will be referred to as “TextureData” henceforth.

The function called `checkMotion()` implemented in the “MotionCam.js” were used to transform the bounds of an area and calling the function named `getMovement()`. It allows for detection of motion in a specified area of the screen by calculating the motion detected inside the boundaries. All pixel values were added up and compared with a threshold to classify if a motion was present. By averaging the position of all pixels where motion was detected, an average position of the motion was acquired. The pixel values are retrieved from the `TextureData` array by iterating with two nested for-loops. The first iterates through the width of the region and the other through the height.

It is important to note that the array was an 8-bit array, thus every four values constitutes a pixel (red, green, blue and alpha). To retrieve a pixel $P$ at the position $(x, y)$ in the array, with the camera width $w$, the following equation was used:

$$P_{x,y} = (y \times w + u) \times 4 \quad (3.1)$$

The module was mainly tested with the Phaser IO framework and a short explanation on how will therefore be given. First, a `PIXI.BaseTexture` containing the module’s canvas was created. Then a `PIXI.Texture` was using this `PIXI.BaseTexture` in its initialization because its `frame` property. The `frame` was set to a new empty `Phaser.Frame` with the height and width of the game. At last, a `Phaser.Image` was created with the aforementioned texture and frame. The sprite now contains the graphical representation of the motion detected and may be used as the user prefers.
3.2. Filter Implementation

The filters that were implemented in the filter test program and were used during this thesis are described below and are grouped into the four stages of the Background Subtraction flow diagram described in section 2.1. A short explanation on how to acquire a direction of the detected motion and how colors can be extracted from the data will also be given.

3.2.4.1 Pre-processing Implementation

The filters presented below were used to remove noise and other relevant errors in the textures.

Temporal Smoothing
The temporal smoothing was implemented by passing the current and the previous frame to a fragment shader as Sampler2D variables. Then equation (2.1) was computed. By using the gradient as an adjustable variable, a desired value of the constant could be acquired conveniently. The resulting image was then drawn and stored with the “ThreeRTT.js” library.

Average filter
The next step was to create a way to perform different spatial smoothing operations. The webcam’s current image data was sent to a shader and all eight neighbours were fetched by using the texture offset explained earlier. Then the current RGB values of the pixels were converted to grayscale values by using the YIQ conversion formula [35], which is adopted by NTSC.

\[
GRAY = 0.299 \times R + 0.587 \times G + 0.114 \times B
\]  

(3.2)

When the neighbours grayscale values were added up with the current pixel value and divided by nine, a straightforward average filter was acquired.

Gaussian Blur
Since the Gaussian function is separable, the Gaussian blur function was divided into two fragment shaders, one contained horizontal blur and the other one vertical blur. Both of the shaders works roughly the same, but uses horizontal and vertical offsets respectively. The horizontal fragment shader were computed and stored in a ThreeRTT.Stage. The stage’s texture was then passed to the vertical fragment shader to produce the final Gaussian blurred texture. Considering the computation of the Gaussian function are done on the GPU, D. Rákos [10] linear sampling can be used. A function which received a variable sigma and created two arrays containing linear sampled weights and offsets were therefore implemented. The weights and
offsets were achieved by using the equations (2.3a), (2.4a) and (2.4b).

**Chromaticity**
As said in Section 2.1.1.3, the RGB values of a texture are not enough to solve the problem of misclassification of foreground pixel because of shadows. Therefore the RGB values were converted to chromaticity coordinates using the equations in (2.6a) and considering the loss of lightness information, the third coordinate was replaced with the lightness measure $s = R + G + B$.

### 3.2.4.2 Background Modelling Implementation

The background model is used as a reference image and was created and maintained in this part of the Background Subtraction process.

**Frame Differencing**
A technique that often suffers greatly from the "Foreground aperture". It stores the previous frame as the background model. The extended method by Y. Tian and A. Hampapur [21] was implemented by passing the current and previous image into a fragment shader. Then the weighted accumulation was calculated with the equation (2.7). The weighted accumulation parameter was used as an adjustable variable, thus it could be changed at run time.

**Temporal Median and Average**
By passing in a set number of previous gray scaled frames, a median filter can be acquired by sorting the pixel values of each frame, and then picking the median of the values. The resulting texture was then stored in a $ThreeRTT.Stage$ because it was going to be used in the upcoming stage.

The Average filter was implemented similarly. But used the average of the previous gray scaled frames, as in equation (2.8), instead of the median.

**Color and Edge Model**
The method presented by S. Jabri et al. [23] was implemented. First, 2 textures were created, they stored the mean and standard deviation of each color channel, respectively. Then 2 more textures were created to store the horizontal and the vertical edge differencing images. Similar to the color channels, 2 textures were created to store the weighted means and standard deviations for each edge color.

The Sobel operators in figure 3.3 were used to detect the horizontal and vertical edges by convolving them with the color channel images. As described in section 2.1.2.2, the standard deviation and mean was calculated using the equations (2.11) and (2.11) for each color channel image and all edge images.
3.2. IMPLEMENTATION

Figure 3.3: The two Sobel operators used to obtain the vertical and horizontal edges.

3.2.4.3 Foreground Detection Implementation

Frame Subtraction
When doing the foreground detection, the current frame and the reference frame created and maintained in the previous section were passed in to a fragment shader. The current frame’s texture pixel was then subtracted from the reference frame’s texture pixel. Then the absolute value of the resulting value was compared to a threshold. If it was exceeded, the pixel value was replaced with an 1.0, otherwise 0.0.

Color and Edge Subtraction
When continuing the background model presented by S. Jabri et al. [23], the color textures and the current frame were passed to a fragment shader. Each of the color texture’s color channels were subtracted from the corresponding color value in the current frame. The three resulting values were compared with the thresholds $m\sigma$ and $M\sigma$. If the value was higher than $m\sigma$ but lower than $M\sigma$, the equation (2.13) was calculated. To acquire the color channel with the most contribution, the maximum of the three values were as used for calculations further on.

When calculating the edge subtraction, the edge gradient $\Delta G$ needs to be acquired. It was obtained by, firstly, calculating the current frame’s edge color models. The edge’s mean texture were then passed to a fragment shader to be able to subtract each color channel with the color channels of the current frame’s edge model. When the maximum value of the addition between the edge color channels and edge mean color channels was acquired, the calculated edge reliability $R$ (equation (2.15)) was used similarly as in the color subtraction. Lastly, the maximum value from the Color and Edge subtraction is used to determine if it is a foreground pixel.
Edge and Chromaticity Subtraction

By using the chromaticity textures created in the Pre-processing stage and the Sobel processed textures acquired in the aforementioned method, the subtraction method by S J. Mckenna et al. [18] could be implemented.

The chromaticity values $r$ and $g$ and the lightness $l$ are compared with a chromaticity and lightness threshold respectively. If $l$ exceeds its threshold but the chromaticity values are lower than their threshold, no motion should be detected since it is most likely triggered by a shadow. As said by S J. Mckenna et al., only using chromaticity values will not suffice since the difference in chromaticity in the foreground and background is often minor. The mean and magnitude variances of horizontal and vertical Sobel textures are thus used. If the distance between the current Sobel texture and the mean Sobel texture is greater than 3 times the maximum of the average variance and the standard deviation, the pixel is classified as a foreground pixel [18]. The standard deviation is acquired by calculating the square root of the current Sobel magnitude variance.

3.2.4.4 Data Validation Implementation

The final step of the Foreground Detection process. The resulting foreground image obtained from the foreground detection was processed with morphological filters. To achieve a dilation of the foreground image a variable $maxValue$ was set to 0.0, then all neighbours were iterated. If the current $maxValue$ was exceeded by a neighbour’s value it was replaced with the exceeded value. The erosion was processed slightly similar, a $minValue$ was set to 1.0, then the $minValue$ was compared with the minimum of the neighbours.

The filter presented by MMC Pawaskar et al. [25] were also implemented. The neighbouring values was added to a sum and if it was greater than 4, it replaced the current texture pixel with a 1.0, otherwise 0.0.

A different approach was implemented using the foreground image obtained from the color and edge subtraction. By only keeping the texture pixels with neighbours containing the value 1.0, false foreground classification was removed.

3.2.4.5 Other Image Processing Filters

This section will cover the implementation of different types of methods that uses the data acquired from the previous method.
Motion Direction
A method to detect which direction a motion has was implemented to be able to know which way an object moves. The process was divided into two steps and initiated with a filter that combined the previous texture with the current one. If a pixel’s position only contained detected motion from the previous texture the pixel’s value was changed to 0.5. Whenever a pixel’s position contained detected motion in the current texture, the pixel’s value was set to 1.0. The process is displayed in the figure below.

![Figure 3.4: A figure showing a motion moving down right. The image to the left and in the middle are the previous and current texture respectively. The image on the right is the resulting texture after the combination.](image)

The resulting texture was then used in the next step where the actual direction is calculated and all neighbors of each pixel with a value higher than 0.0 was iterated. If the neighbors value was higher than the current pixel’s the neighbors direction was added to a sum of directions. The resulting sum was divided by the number of contributing neighbors to achieve an average direction. By using the most contributing axis a color could be applied to display if the average direction is pointing right, left, up or down.

Color Extraction
By making use of the ability to measure color distance in chromaticity space a specific color can be extracted. A given RGB value was transformed to the chromaticity coordinates rg. If the distance between rg and the current pixel value is greater than a certain threshold, the pixel value will be changed to 0.0. This results in motion detected by other colors than the given one will not be displayed.
Chapter 4

Result

This section will present the results acquired from the different filters that were implemented. It will provide images that displays how a filter’s result look to be able to compare the filters to one another. The images were captured at a frame where motion occured. The performance of each filter process will be shown in a table which used the tool rStats [36] to analyze the current frame time, developed by J. Sanchez.

All filter images were captured with the specifications shown in the table below if nothing else is specified.

Table 4.1: The specifications used when collecting the resulting images and frame times.

<table>
<thead>
<tr>
<th>Type</th>
<th>Specification</th>
</tr>
</thead>
<tbody>
<tr>
<td>Computer Model</td>
<td>HP Z220 CMT WorkStation</td>
</tr>
<tr>
<td>Processor</td>
<td>Intel(R) Core(TM) i7-3770 CPU @ 3.40GHz 3.40 GHz</td>
</tr>
<tr>
<td>RAM</td>
<td>32.0 GB</td>
</tr>
<tr>
<td>System Type</td>
<td>64-bit Operating System</td>
</tr>
<tr>
<td>Display Adapter</td>
<td>Intel(R) HD Graphics 4000</td>
</tr>
<tr>
<td>Camera</td>
<td>USB Camera-B4.09.24.1</td>
</tr>
<tr>
<td>Camera Resolution</td>
<td>800x600 pixels</td>
</tr>
<tr>
<td>Canvas Resolution</td>
<td>800x600 pixels</td>
</tr>
</tbody>
</table>
4.1 Feasability Study

After an extensive research, the Gaussian, Temporal and Average blur were implemented as Pre-process filters. They all used different techniques to remove camera noise. The Temporal Median, Temporal Average and Frame Difference model where the Model Stage filters that were used. Since both the Color and Edge process and the Color and Chromaticity process could be used to suppress shadow, their associated Model and Foreground Detection where implemented. A Frame Difference filter where also created as a Background Subtraction filter. The Data Validation filter by MMC Pawaskar et al. [25] and a standard Erosion and Dilation filter where used as Data Validation filters. They were both filters that could remove isolated pixels and fill holes in the image.

4.2 Implementation

The five following figures will display the results from the different Pre-process filters. To demonstrate how much noise the filters can remove, a frame differencing method with a threshold at 0.014 that uses the previous frame as a model was used. It is worth noting that 0.014 is a considerably low threshold and produces a great amount of false classifications of motion detected.

Figure 4.1: Figures showing the result from two of the Pre-process filters. The image to the left displays motion detected without any Pre-process filter.

The figures in 4.1 shows how efficient the Gaussian and Average blur are at removing noise. The figure 4.1a is the motion detected without any Pre-process filter.
Figure 4.2: The temporal blur with the gradient 0.4 (left) and 0.7 (right).

The figures in 4.2 shows the result from the Temporal blur with the gradient 0.4 and 0.7. The higher the gradient was, the more noise was removed but the motion detected gave a longer trailing effect.

Figure 4.3: Figures showing the result from three of the Background Model filters.

The background model images shown in 4.3 were captured using Gaussian blur as Pre-processing filter and was subtracted from the current frame. The Frame Differencing model displayed in 4.3c was captured with the weighted accumulation parameter 0.59 and a threshold at 0.055. The Average 4.3a and Median model 4.3b both stored six frames in their buffer.

Figure 4.4: The result from convolving the scene with the Sobel filters.

The figures in 4.4 shows the results from convolving the scene with a horizontal (4.4a), vertical (4.4b) and both horizontal and vertical (4.4c) Sobel filters.
filter.

Figure 4.5: Figures showing the result of different Foreground Detection processes.

Figure 4.5a displays the result of the ”Color and Edge” Subtraction that used the ”Color and Edge” Model as model. It used the values shown in the table below.

Table 4.2: The values used in the Color and Edge process.

<table>
<thead>
<tr>
<th>Variable</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Edge $m$</td>
<td>0.03</td>
</tr>
<tr>
<td>Edge $M$</td>
<td>0.09</td>
</tr>
<tr>
<td>Color $m$</td>
<td>0.15</td>
</tr>
<tr>
<td>Color $M$</td>
<td>0.25</td>
</tr>
<tr>
<td>Variance $\alpha$</td>
<td>0.9</td>
</tr>
<tr>
<td>Mean $\alpha$</td>
<td>0.6</td>
</tr>
</tbody>
</table>

The Chromaticity and Edge Subtraction shown in figure 4.5a used the threshold 0.077 and the chromaticity threshold 0.06. The Frame Subtraction in figure 4.5c is captured with a threshold at 0.143.

Figure 4.6: Figures showing how efficient they suppress shadows.
The figures in 4.6 compares how well the different methods are able to suppress shadows where figure 4.6c is using the threshold 0.044. Figure 4.6a and 4.6b are using the values displayed in 4.2.

![Comparison of different methods](image)

(a) None  (b) Pawaskar  (c) Dilation and Erosion

Figure 4.7: The result from the Data Validation filters where the figure to the left is showing the motion detected without any filter.

The Data Validation filters shown in figure 4.7b and 4.7c where both using a Frame Subtraction filter with no Pre-process filter. The Frame Subtraction threshold was 0.022.

![Frame Subtraction results](image)

(a) Original  (b) Red Extracted

(c) Without extraction  (d) With extraction

Figure 4.8: The Color Extraction filter.

The figures shown in 4.8 displays the result of the color extraction. Figure 4.8a shows the original image while figure 4.8b shows the RGB value of red
being extracted. The difference between using the extracted color or not in motion detection is displayed in figure 4.8c and 4.8d.

![Image](image1.png)

(a) Right  (b) Up

Figure 4.9: The motion direction images acquired when an arm is moving right (left image) and up (right figure).

The two figures in 4.9 displays the resulting image where a hand moves right (4.9a) and up (4.9b). The color displayed indicates which direction the motion is moving.

![Image](image2.png)

(a) Moving cam  (b) Bug game

Figure 4.10: WebGLMotionCam.js used in two different Phaser IO games.

A visual representation of when the WebGLMotionCam.js is used with two different Phaser IO games is shown in figure 4.10. Figure 4.10a shows an example of when the motion detection is used in a small bounding box. A game where the player should prevent the bugs from eating the tree by moving over them is shown in figure 4.10b. The games are using WebGLMotionCam.js with Gaussian blur as Pre-process filter and Frame Subtraction as Foreground Detection. The previous frame where used as Background Model and no Data Validation filter were used.
Figure 4.11: A chart that illustrates the frame time that the filters are varying between.

Figure 4.12: A chart that illustrates the frame time the Edge Color and Edge Chromaticity process are varying between.

The charts shown in figure 4.11 and 4.12 presents the frame times which the filters och filter processes vary between when using them. All frame times where captured using the specifications presented in table 4.1. For measurement, a frame time that vary between 1.0 ms and 1.6 ms results in a FPS (Frames Per Second) between 625.0 and 1000.0.
Chapter 5

Discussion

This chapter will discuss and criticize the received results and the methods used to achieve them. There will also be a discussion about the ethical and societal aspects related to the work.

5.1 Results

The resulting images and frame times achieved from the implemented filters and programs gave an altogether good result, with some minor unexpected flaws. When comparing how efficient the Pre-process filter could remove camera noise before a frame difference was made, the Gaussian blur stands out to be the most efficient. While keeping a frame time below 0.15 ms it did not leave a trailing effect like the Temporal blur and still kept the more detailed information. When playing motion controlled games, a trailing effect is often not wanted since the player’s precision in its movement is reduced. The received result from the Gaussian blur was no surprise since it is well acknowledged and widely used in the literature [13][23].

The results acquired from the different background model filters gave varying result. The Temporal Average Model removed lots of necessary details and the Temporal Median model tended to distort the data in the image. The Frame Differencing filter maintained the details but gave surprisingly high frame time since it uses less operations and texture pixel fetches than the Temporal Average and Median Model. Although not seen in the resulting images, the Temporal Average and Median Model both managed to suppress sudden light variations better than the Frame Differencing Model. Considering the flaws of each background model filter, it seemed best to use
the previous frame as the background model when used in the WebGLMotionCam.js. Despite it having higher sensitivity to sudden light variations, a model filter that has high detail at a low cost is still favored.

The images received when convolving the Sobel filters with the scene gave expected results as they resembles the Sobel images presented in [23].

The Color and Edge process relied heavily on the values of its parameters, thus it was difficult to adjust the values to acquire a desired result. A too high or too low value of one of the parameters could result in either detail loss but suppressed shadow or high detail with a shadow still occurring. The Chromaticity and Edge process was superior when removing shadows and still provided the same detail level as the Color and Edge Process. The Chromaticity space is known to suppress shadows better than the RGB space and the results was therefore as expected, it is also shown by El gammal et al. in [16]. It is also worth noticing the 0.53 ms frame time difference between the two processes. Although the Frame Subtraction did not remove hardly any false classifications of motion detected influenced by shadows, the high detail and the relatively low frame time made it the preferred Foreground Detection filter in the WebGLMotionCam.js.

The Dilation and Erosion filter and the Pawaskar filter did not display much difference in both images and frame time. They both removed noise very efficient but since the Gaussian Pre-process filter removed almost all noise the Data Validation filters was not needed in the WebGLMotionCam.js.

Both the Color Extraction and Motion Direction are different ways the data acquired from the motion detection can be used. The Color Extraction works very well if a correct RGB value and threshold is set and the light conditions are considered. The difficulty of being able to extract only the wanted color is, for now, one of the downsides of the method, but it opens up a vast field of possibilities for new game designs when acquired.

Even though the Motion Direction filter displays a quite good result when moving in either direction moderately slow, it comes at a high frame time cost. A downside is that the resulting color only appear at the edge of the motion detected thus makes wrongly classified directions influence the result considerably. This could arise problems when actually using the motion direction filter to determine which direction a motion has moved. It could be solved by analyzing a region around each pixel to determine which color value that appears most and then replace the current pixel value with it.
5.2 Method

The idea of creating a separate program where the filters could be tested in different orders conveniently turned out to be rewarding. By using the libraries "Three.js" and "ThreeRTT.js" more time could be spent on the implementation of the filters instead of the process between. The use of external libraries has some disadvantages though, there are often a great deal of functionality that needs to be loaded but are actually never used. Since "Three.js" is 3D-library and I only use 2D-operations, it most likely contains a huge amount unused functionality that must be loaded.

At the moment, the process of creating the different filter stages is rather inefficient because I wanted to separate the stages from each other to achieve a more straightforward way to change the filters at run time. It also made the troubleshooting more convenient. This results in a new ThreeRTT.Stage was created for each filter method when a combination of filters could be used in a single ThreeRTT.Stage instead. It should be mentioned that the project were planned to use only the "Three.js" library first. But I was not able to find a more efficient solution than using "ThreeRTT.js" to acquire data from past frames.

Both the programs uses glReadPixels() to acquire data from the current context and it seemed to be a cause of loss in performance. I investigated if there was a difference in performance if the glReadPixels() only was done on multiple smaller areas than the hole image area. It appeared to get worse and it was because the glReadPixels() call forces the CPU to synchronise with the GPU and thus losing the parallelization of the GPU. A method that does not need the CPU to synchronise would therefore result in a performance boost for both of the programs.

A single image can not fully represent how well a filter works, it can only convey an approximation of it. Things such as light variations and trailing effects can not be displayed well in a single image. The optimal way to represent the filters would be to try them oneself or show a live example in a video. Since the images of the different stages were taken on varied times the comparison of the images are not on exactly on the same terms, only approximately. An image that is compared with another could have a small difference in speed of the motion detected. The importance of describing how the filters felt and looked is therefore great.

The Motion Direction filter is, for now, only working properly when moving in directions rather slowly. This is because fast movements occasionally merge edges of different sides of the moving object. It leads to false classification of found directions between the current and previous frame since the current pixel will discover lower neighbouring pixel values from other
5.3. THE WORK IN A WIDER CONCEPT

CHAPTER 5. DISCUSSION

edges. There is a great amount of improvement to be done in this area of the project.

The equipment used when gathering the result does influence final outcome. Different cameras manages noise variously and may have different frame rates they can produce. The specification used in this thesis had a relatively good processor while the display adapter was below average in performance. That said, to receive similar results the specifications needs to match the used specifications in some manner.

The use of a standard web camera as a motion detection device has its advantages and disadvantages compared with the Kinect Sensor [1]. Since the Kinect uses a depth sensor it can remove everything other than the person that is actually playing and also interpret specific gestures. It even provides full-body 3D motion capture. The standard web camera does not have any depth information and is therefore restricted to 2D motion capture. The web camera’s main advantage is its availability compared to the Kinect sensor. The motion controlled games can therefore be played by more people.

The sources used in this thesis were mostly focused on how motion is detected in a video sequence. The massive amount of work done in the motion detection field made me want to focus on literature that contains possible solutions to problems that were thought to be relevant for motion detection games. Such solutions were shadow detection, light variations or noise removal. This leads to a lot of methods that are not mentioned or implemented since there was not enough time. If there had been a broader search space where keywords similar to motion direction or color extraction were used, these areas may have been improved exceptionally. The rather small research area of Texture Based methods made it hard to find any suitable methods to implement in this thesis, since the methods found had problems with shadow detection. The validity of the sources that did not origin from a research article, such as the text about linear Gaussian Blur by D. Rákos [10], could be questioned. But their methods were well thought out and often provided their own sources and was therefore considered as valid sources.

5.3 The work in a wider concept

Motion detection is commonly used in a variety of applications such as traffic cameras or surveillance cameras and that may not be a common knowledge. When displaying the motion detection application to an older or inexperienced audience, it made me realize that people are not often comfortable with using the web camera as anything else than for video chatting. The insecurity when playing may affect the emotions of the player
since it leads to the believing that someone is watching them play when testing the motion detection games. There is no indication that there are not. An instruction where the application notice the user that recording and related actions will not be used should have been added since it only process the video data and displays it on the screen.

The main difference between motion detection games and traditional games is that the player actually needs to move their whole body to play. As said in the Introduction 1, motion detection games are a great way of exercising while having fun and being immersed in the game. The web camera is already found in the majority of the modern households and by only playing a motion detection game a few times every day, it can help to stay physically active. If the motion detection games gets more attention it may lead to an overall increase in health level in the society. It is thus important to have as good motion capture as possible, to let the players be immersed in the games without being disturbed by inaccurate motion detected.
Chapter 6

Conclusion

The main purpose of this thesis was to find a motion detection algorithm that could give a qualitative representation of video data with as low performance cost as possible. The best resulting algorithms were then used in a JavaScript module and its primary objective was to be used with web based games. Since it is important to keep a high frame rate in games, the performance of the motion detection module was thus required to be as high as possible. The quality of the motion detected was also an important factor to consider. It needed to be kept as high as possible since it influences the immersion of the player and the overall quality of the game.

In the end, the WebGLMotionCam.js used the Frame Subtraction filter as its Foreground Detection method. It was less complex than the other more sophisticated methods but it was concluded that the low frame time and high detail was a higher priority than taking shadow removal and light variations into account. If the performance and detail quality of the Edge and Chromaticity process were optimized, it would make it an essential method to use instead since the way it suppresses shadows were very efficient.

The way the Gaussian blur could be optimized on the GPU made it the best candidate for noise removal in the Pre-process stage. The good result made it possible to disregard the Data Validation stage fully since its contribution to a better quality was not worth its frame time. The WebGLMotionCam.js uses the previous frame as its Model Stage because the implemented Model Stage filters were not contributing to a better result.

The possibilities of what one can do with resulted motion detected data are almost endless. As previously mentioned, the implemented Motion Direction filter only works with slow movements and are still displaying wrongly
classified pixels. It was implemented to create new ways of designing games and a better way to detect the direction of a motion is therefore essential. It could be used to apply a force on an object in a certain direction when activated. Another usage could be to calculate an incoming angle from the direction and determine the direction which an object should bounce off to. If computing the velocity and acceleration was made possible for instance, the amount of force applied to an object could be calculated. For now, the only way to know the level of motion is to compute the total number of foreground pixels in the specified area.

The classified foreground pixels have a tendency to appear mostly around edges because the pixel difference is usually greater. It is common to analyze and label such regions to fill the areas between them to represent the whole moving object and not solely its edges. Such implementations often require a great amount of performance, but if a solution that utilizes the parallelism of the GPU were made, it may increase the feedback acquired from the WebGLMotionCam.js.

Conclusively, I was quite pleased with how well the module worked with different web based games and how the filter test program helped with finding a good method more efficiently. There will always be additional functionality to implement or more optimizations to be made, but I believe that the resulting motion detection algorithms makes good use of the GPU’s parallelism and gave an adequate representation of the video data.
Bibliography


42

[25] Mr Mahesh C Pawaskar, Mr NS Narkhede, and Mr Saurabh S Athalye. “Detection Of Moving Object Based On Background Subtraction”. In: ().


På svenska

Detta dokument hålls tillgängligt på Internet – eller dess framtida ersättare – under en längre tid från publiceringsdatum under förutsättning att inga extraordinära omständigheter uppstår.

Tillgång till dokumentet innebär tillstånd för var och en att läsa, ladda ner, skriva ut enstaka kopior för enskilt bruk och att använda det oförändrat för ickekommersiell forskning och för undervisning. Överföring av upphovsrätten vid en senare tidpunkt kan inte upphäva detta tillstånd. All annan användning av dokumentet kräver upphovsmannens medgivande. För att garantera äktheten, säkerheten och tillgängligheten finns det lösningar av teknisk och administrativ art.

Upphovsmannens ideella rätt innefattar rätt att bli nämnd som upphovsman i den omfattning som god sed kräver vid användning av dokumentet på ovan beskrivna sätt samt skydd mot att dokumentet ändras eller presenteras i sådan form eller i sådant sammanhang som är kränkande för upphovsmannens litterära eller konstnärliga anseende eller egenart.

För ytterligare information om Linköping University Electronic Press se förlagets hemsida http://www.ep.liu.se/

In English

The publishers will keep this document online on the Internet - or its possible replacement - for a considerable time from the date of publication barring exceptional circumstances.

The online availability of the document implies a permanent permission for anyone to read, to download, to print out single copies for your own use and to use it unchanged for any non-commercial research and educational purpose. Subsequent transfers of copyright cannot revoke this permission. All other uses of the document are conditional on the consent of the copyright owner. The publisher has taken technical and administrative measures to assure authenticity, security and accessibility.

According to intellectual property law the author has the right to be mentioned when his/her work is accessed as described above and to be protected against infringement.

For additional information about the Linköping University Electronic Press and its procedures for publication and for assurance of document integrity, please refer to its WWW home page: http://www.ep.liu.se/

Oskar Havsvik