MASTER'S THESIS | LUND UNIVERSITY 2018

Automated classification of graphical artefacts in 3D graphics verification

Arvid Carlman, Daniel Cheveyo

Department of Computer Science
Faculty of Engineering LTH

ISSN 1650-2884
LU-CS-EX 2018-33

Automated classification of graphical artefacts in 3D graphics verification

Arvid Carlman
arvid_carlman@hotmail.com

Daniel Cheveyo
daniel_cheveyo@hotmail.com

June 7, 2018

Master's thesis work carried out at ARM.

Supervisors: Flavius Gruian, Flavius.Gruian@cs.lth.se
Rasmus Persson, rasmus.persson@arm.com

Examiner: Michael Doggett, Michael.Doggett@cs.lth.se

Abstract

System testing is performed when developing 3D graphics hardware and drivers for GPUs. A crucial aspect when performing system testing is comparing a given output image of a specific scene to its corresponding reference image. There are many methods of quantifying the difference between two images, most of which only produce a scalar value as a measurement. To save time in the development process by speeding up the (usually manual) classification of the graphical artefacts in these images, an automated classification tool was produced, which is described in this paper. Given a relatively small set of data, a certain number of classes were determined based on the similarities of the artefacts. The classification of the test set of images gave an accuracy of 78%.

Keywords: image analysis, classification, graphics, graphical artefacts, verification

Acknowledgements

This master thesis has taken place at the company ARM in Lund from February 2018 to June 2018. The goal of this thesis was to ease the content validation team's (CV) work for identifying graphical artefacts by making the process automated. We would like to thank the CV team for their generosity and cooperation, and also special thanks to our supervisors, Flavius Gruian and Rasmus Persson, for interesting thoughts and constructive feedback. We would also like thank Staffan Gadd for his role as an informal supervisor.

Contents

Chapter 1
Introduction

System testing is performed when developing 3D graphics hardware and drivers. A crucial aspect when performing system testing is comparing a given output image of a specific scene to its corresponding reference image. The main tool used in this comparison is Peak Signal-to-Noise Ratio (PSNR). PSNR can only be used to tell whether there is a difference between the two images or not, and of what magnitude it is. As PSNR does not give any information what the difference is in terms of different possible graphical artefacts, one cannot know what faulty images are in fact due to the same bug. With this as motivation, the main aim and purpose of this thesis is to create a tool which can provide such information, as it would speed up the time to market of the product by more efficiently finding errors that correlate to each other and give an additional measurement to ensure whether or not any bugs are present.

This will be done by classifying what graphical artefacts might be contained in a given faulty image through image analysis and determining if the found errors correlate to any predetermined class. Since the artefacts that emerge are so different, as they depend on the choice of scene and frame of a test and due to the relatively small amount of images to test on, and also that the number of images in the reference set was only 112, a normal machine learning algorithm would not be sufficient to get a robust classification. That is why more general image analysis methods are used instead in this thesis. In addition to the classification, other information related to each class will be reported, e.g. the number of faulty pixels when classifying the pixel error class and other information describing the nature of the class.

The work of this thesis is divided into the following parts:

- Review. Both authors have reviewed each others code and writing throughout the thesis.

- Implementation. Both authors participated in most code and sometimes pairprogrammed, but the heaviest algorithms are implemented by A. Carlman.

- Evaluation. Aspects in the discussion have been added by both authors, but mostly evaluated by D. Cheveyo.
- Writing. Both have written parts of all sections and rewritten sections so that the text proved satisfactory, but most of the writing was performed by D. Cheveyo.
- Generation of test images were carried out by A. Carlman and other figures were made by D. Cheveyo.

The disposition of this thesis is the following:

We will start by presenting and explaining the theory of the concepts and the methods, used in this thesis, in the Theory chapter. After that we will introduce the Methods chapter, which will contain a description of how the data used were processed and collected. The chapter will also provide an overview of the data flow through the implemented algorithms, and the chosen approach to the given problem. The actual description of the implemented algorithms is written in the Implementation chapter. This will be done on a high level, to easier grasp the ideas and the aim of the algorithms. This chapter will provide motivations for and examples of the developed algorithms, connected with the Theory and Methods chapters. The outcome of the classification of the reference data is going to be presented in the Experimental evaluation chapter. It will also be evaluated by the metric of being classified correctly or incorrectly, and a model of the computation time when executing the identification of the classes. The chapter after, discussion, will be a discussion about how well the results were and possible improvements. The last chapter, Summary and conclusions, will contain a brief summary of important discoveries, that we found during the development, and conclusions about the outcome of this master thesis in general.

1.1 Related work

Since the chosen approach to solve the classification in this thesis does not include machine learning, a large portion of today's academic work in image classification is mostly not related. Instead image segmentation and other image analysis concepts are very relevant to this thesis. This section will state examples of work, that uses core concepts and methods used in this thesis.

A report, that has similar methods as this thesis, is written by S. Joseph et. al. [1]. In the report, histograms are created to adjust thresholds and edge maps are used to segment a given image.

In an article, written by J.-C. Yoo and C.W. Ahn, they use PSNR to match images [2]. In their article, they use the methods to detect occluded objects, but it is also relevant to the background of the thesis because PSNR is a central component to detect differences in images.

Many concepts and methods, that are used throughout this thesis, can also be found in a book written by Burger and Burge [3]. In their book, they describe how histograms are created and interpreted, how edges are detected from an image and what makes an edge map, and the concepts and properties of morphological and other common filters.

Chapter 2

Theory

Necessary information and description of used concepts and methods are here explained. It can be ignored at first reading and instead be used as reference for methods that the reader would like to receive further understanding of. The sections in this chapter are ordered as they are mentioned in other chapters.

2.1 Reference and output image

The reference image of a specific frame from an application is extracted and stored beforehand from a test run which could be considered to be completely correct.

The test is run once more under other circumstances and a new output image is extracted, which is to be evaluated. These new circumstances could mean a new driver revision, other hardware or some other change.

2.2 Peak signal-to-noise ratio

According to Zhou Wang and Bovik, A.C., Peak signal-to-noise ratio (PSNR) is a ratio between the maximum possible power of a signal and the power of corrupting noise that affects the fidelity of its representation [4]. PSNR is defined as follows:

$$PSNR = 10 * log_{10}\left(\frac{MAX_{intensity}^2}{MSE}\right) \tag{2.1}$$

Where $MAX_{intensity}$ is the maximum intensity in the images, in this case, 255 since the given images are coded in 8 bits per pixel. MSE stands for the mean square error, which is simply defined as:

$$MSE = \frac{1}{mn}\sum_{i=0}^{m-1}\sum_{j=0}^{n-1}(I_{ref}(i,j) - I_{out}(i,j))^2 \tag{2.2}$$

Where m specifies the number of rows, and n number of columns of the given reference image, I_{ref}, and output image, I_{out}. In the case of the data used in this thesis, an image is in RGB format (three color values for each pixel), and therefore for each $m * n$ pixels its RGB values are summed and divided by three to receive a weighted value for each pixel. With these weighted values for each pixel, the MSE is calculated as described and then the PSNR value.

In other terms, PSNR detects if there is any difference between two images, and that is enough to trigger our classification algorithms. Korhonen and You state that PSNR predicts the perceived subjective quality of images almost as well as more complex quality models [5].

2.3 Difference images

Difference images are the result of comparing images against each other. A pixel in one image against its corresponding pixel in the second image. The use of difference images can either be to know the magnitude of the difference at certain positions or using the image as a mask to segment important parts of the arbitrary image.

2.3.1 Euclidean difference image

The Euclidean difference image (EDI) is an RGB image containing the Euclidean difference between the output, I_{out}, and the reference image, I_{ref}, for every pixel value $i \in m$, $j \in n$. Where m is the number of rows and n is the number of columns of the images. This is done in every RGB channel for I_{ref} and I_{out}. The resulting image is simply defined as:

$$Image_{euclidean}(i,j) = abs(I_{ref}(i,j) - I_{out}(i,j)) \tag{2.3}$$

2.3.2 Binary difference image

The Binary difference image (BDI) has only zeros and ones, hence binary. There is a one in every position corresponding to a pixel that has a value greater than zero in the EDI. This is also done for every RGB channel, as for the EDI. The formal definition is as follows:

$$Image_{binary}(i,j) = \begin{cases} 1, EDI(i,j) > 0 \\ 0, otherwise \end{cases} \tag{2.4}$$

2.3.3 Signed difference image

The signed difference image (SDI) is very similar to the EDI, but instead of taking the absolute value of the difference, one only take the subtraction between I_{ref} and I_{out} for every pixel in the image, see equation 2.3. This is to be able to monitor if the output image has either increased or decreased its intensity at a certain pixel.

2.3.4 Edge map

An edge map is a binary image containing ones at edges, indicating a great difference between its neighboring pixels. The technique serves to simplify the analysis of images by drastically reducing the amount of data to be processed, while at the same time preserving useful structural information about object boundaries [6]. The edge map contains useful information when deciding if errors are aligned with geometry in the given images.

Sobel edge map

Matlab's default method is the Sobel edge detection. It has some advantages compared to other edge detection methods. Wenshuo G. et al claim that it has two main advantages [7]. First, it has some smoothing effect on the random noise of the image. Second, the elements of the edge on both sides have been enhanced, so that the edge seems thick and bright. The Sobel operator is also highly recommended for massive data communication found in image data transfer by Gupta et. al. [8].

In short, the operator calculates the orthogonal gradients in both x- and y-direction across the image. It uses two corresponding convolution filters, see matrix 2.1a and 2.1b. They are 3x3 templates which slide through all the pixels and weights its neighbors correspondingly as seen in table 2.1. Results from the convolutions are combined to find the absolute magnitude of the gradients, which is seen as the edges in the edge map [7], [8].

Table 2.1: Matrices used in the convolutions performed for the Sobel edge detection algorithm.

$$\begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}$$

(a) Template for estimating gradient in the x-direction.

$$\begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix}$$

(b) Template for estimating gradient in the y-direction.

Canny edge map

Sobel did not work very well for finding the somewhat blurred edges of the geometry class, see section 3.3.2, so the edge algorithm was changed to Canny instead, as it is more edge sensitive. It is simply an extension of the Sobel edge detection. Canny uses the same algorithm as Sobel when calculating the gradients, but after that Canny thins out the edges to be one pixel wide. This enables more edges to fit into the image, the thickness of the edge is irrelevant. The local maximum of an edge is found by comparing a pixel's gradient to its neighboring pixels. Next step, two thresholds are introduced. One for high magnitudes and one for low magnitudes. The pixels that have a magnitude over the high threshold is automatically included in the final binary image, and the pixels lower than the low threshold is automatically excluded. The pixels between is determined if it belongs to pixels that have a connection with the ones that are over the high threshold, if not, the pixels are excluded [6].

2.4 Mipmapping

The first suggestion of this theory was provided by Ed Catmull in his Ph.D. thesis, 1974, reported by Paul Heckbert [9]. The general idea of mipmapping is to create a pyramid of images at levels with decreasing resolution as the pyramid goes up, with the original image at the bottom, level 0. The next image, level 1, is a low-pass filtered and then down-sampled version of that image. The size of this level 1 image is half the size of the original image, both in width and height. This continues to the next level, level 2, with the level 1 image as a base, and so on. As Akenine-Möller describes, the key to mipmapping is to compute which level(s) in this pyramid should be accessed to texture an object [10]. The technique is especially used when textures are far away from the camera in a 3D scene, and therefore a scaled down version of the texture is needed to avoid flickering. When mipmapping goes wrong, artefacts that has a blocky appearance usually show up in the object's texture.

2.5 Pixel connectivity

Segmentation consists of using a binary value matrix (zeros and ones) and measuring the connectivity of each non-zero value to determine what can be said to be the individual non-zero segments of the image.

2.5.1 4-Connected Segmentation

In the case of 4-connected segmentation, each non-zero value is determined to be connected to its horizontal and vertical non-zero neighbors.

When the individual segments are determined they are labeled in the order that they where found so that they can be iterated.

An example of 4-connected segmentation. The matrix:

$$\begin{bmatrix} 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{bmatrix} \tag{2.5}$$

will, when segmented and labeled, result in:

$$\begin{bmatrix} 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 3 & 0 & 0 & 0 \\ 0 & 0 & 3 & 3 & 3 & 0 & 0 \\ 0 & 0 & 0 & 3 & 0 & 4 & 0 \\ 0 & 2 & 0 & 0 & 4 & 4 & 4 \\ 0 & 0 & 0 & 0 & 0 & 4 & 0 \end{bmatrix} \tag{2.6}$$

2.5.2 8-Connected Segmentation

In the case of 8-connected segmentation, each non-zero value is determined to be connected to its horizontal, vertical and diagonal non-zero neighbors.

When the individual segments are determined they are labeled in the order that they where found so that they can be iterated.

An example of 8-connected segmentation. The matrix:

$$\begin{bmatrix} 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{bmatrix} \tag{2.7}$$

will, when segmented and labeled, result in:

$$\begin{bmatrix} 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 2 & 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{bmatrix} \tag{2.8}$$

2.6 Morphological operations

Mathematical morphology was first introduced by Georges Matheron and Jean Serra and is today used for several reasons, such as filtering, sampling and segmentation [11]. Mathematical morphology analyses the shape and form of objects based on set theory, lattice theory, topology and random functions. Examples of two operations risen from Mathematical morphology, that are used in this thesis, are erosion and dilation. For the curious reader, the following articles review morphological operators [12] and [3]. A structure element, B, is needed to perform the operations. It is a smaller binary image, and determines how the dilation and erosion will form the objects in the given binary image. Note that the descriptions of erosion and dilation below will only change the binary image if

$\neg(\forall i,j|Binary(i,j) = 1) \lor \neg(\forall i,j|Binary(i,j) = 0)$, i.e. if the binary image does not solely consist of ones or if the binary image does not solely consist of zeros.

2.6.1 Dilation

Given a binary image A and a structure element, B. Dilation is defined as:

$$A \oplus B = \bigcup_{d \in A} B_d \tag{2.9}$$

where B_d represents a translation (shifting) for the structure element B by d, i.e. for position p, $B_d \equiv \{(p+d)|p \in B\}$. This results in larger segments, because B iterates inside A for each pixel acting as a center of B, and include pixels that contain B. A larger B results, therefore, in a larger A.

An example of dilation. Given the binary image, A:

$$\begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \tag{2.10}$$

and the structure element, B:

$$\begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix} \tag{2.11}$$

will result in the following binary image, where the bold numbers are marking the positions of the original binary image's ones:

$$\begin{bmatrix} 0 & 0 & 1 & 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & \mathbf{1} & 1 & 1 & 0 \\ 1 & 1 & \mathbf{1} & \mathbf{1} & \mathbf{1} & 1 & 1 \\ 1 & \mathbf{1} & \mathbf{1} & \mathbf{1} & \mathbf{1} & \mathbf{1} & 1 \\ 1 & 1 & \mathbf{1} & \mathbf{1} & \mathbf{1} & 1 & 1 \\ 0 & 1 & 1 & \mathbf{1} & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 & 1 & 0 & 0 \end{bmatrix} \tag{2.12}$$

2.6.2 Erosion

Given a binary image A and a structure element, B. Erosion is defined as:

$$A \ominus B \equiv \{p \in \mathbb{Z}^2 | B_p \subset A\} \tag{2.13}$$

for position p. This results in smaller segments, because B iterates inside A for each pixel acting as a center of B, and exclude pixels that can not contain B inside A. A larger B results, therefore, in a smaller A.

An example of erosion. Given the binary image, A:

$$\begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \tag{2.14}$$

and the structure element, B:

$$\begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix} \tag{2.15}$$

will result in the following binary image, where the bold numbers are marking the positions of the original binary image's ones:

$$\begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & \mathbf{0} & 0 & 0 & 0 \\ 0 & 0 & \mathbf{0} & \mathbf{0} & \mathbf{0} & 0 & 0 \\ 0 & \mathbf{0} & \mathbf{0} & \mathbf{1} & \mathbf{0} & \mathbf{0} & 0 \\ 0 & 0 & \mathbf{0} & \mathbf{0} & \mathbf{0} & 0 & 0 \\ 0 & 0 & 0 & \mathbf{0} & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \tag{2.16}$$

2.7 Skeleton image

The procedure to achieve this image is a thinning methodology. The goal is to reduce information to a minimum while it is still possible to recognize the original pattern. The term skeleton means in this case a representation of a binary pattern with a collection of thin arcs and curves. The skeleton depends on how pixel's neighboring pixels are connected, using 8-connectivity in this case, see subsection 2.5.2. By iteratively removing neighboring pixels that does not belong to the skeleton, the skeleton image is constructed. Lam et. al. investigate different thinning algorithms and compare them by subjectively comparing their resulting skeletons [13].

In this thesis, Medial Axis Transform (MAT) algorithm is used to decide if a point or a pixel belong to an object's skeleton. In short, the MAT algorithm determines, for each point, if a point's 8-connected points is in an object. A point belongs to the skeleton if it has at least two points connected to it this way [13], [14]. In other words, one could think the procedure as successive eroding away pixels from the boundary of the object, while preserving the end points, until no more thinning is possible.

For example, the given binary image:

$$\begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \tag{2.17}$$

will result in the following skeleton image, where the bold numbers are marking the positions of the original binary image's ones:

$$\begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & \mathbf{1} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{1} & 0 \\ 0 & \mathbf{0} & \mathbf{1} & \mathbf{0} & \mathbf{1} & \mathbf{0} & 0 \\ 0 & \mathbf{0} & \mathbf{1} & \mathbf{1} & \mathbf{1} & \mathbf{0} & 0 \\ 0 & \mathbf{0} & \mathbf{1} & \mathbf{0} & \mathbf{1} & \mathbf{0} & 0 \\ 0 & \mathbf{1} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{1} & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \tag{2.18}$$

2.8 Hough transform

The algorithm was founded by Paul Hough to recognize complex patterns [15]. The Hough transform is used to find aligned points in images that create lines and even curves and complex shapes [16]. The most simple case is for straight lines. Consider the following example seen in figure 2.1. The line in the figure can be described with the coordinates x and y and the parameters a and b as $y = a * x + b$. The parameters are used to define the angle of the line. a and b are in this case unknown and x and y are known. In the coordinate space, the parameters are fixed and generate different coordinates for example (x_1, y_1) and (x_2, y_2).

If one converts (x_1, y_1) and (x_2, y_2) in figure 2.1 into the parameter space, one would receive figure 2.2. In this space, the coordinates are fixed and generate possible combinations of parameters that correspond to the given coordinates. If two points in the coordinate space define a line in that space, the parameters of that line can be found by finding the intersection of the generated lines in the parameter space for those points, (a, b).

This is the basics of the Hough transform. Straight lines in the coordinate space are found by looking after the strongest local maxima, so-called peaks, in the corresponding parameter space after all points in the coordinate space are iterated.

Using the Hough transform, the parameters ρ and θ are instead used for the parameters a and b to represent lines, see figure 2.3, where ρ is the distance from the origin and θ is the angle. The principle is still the same as explained with a and b.

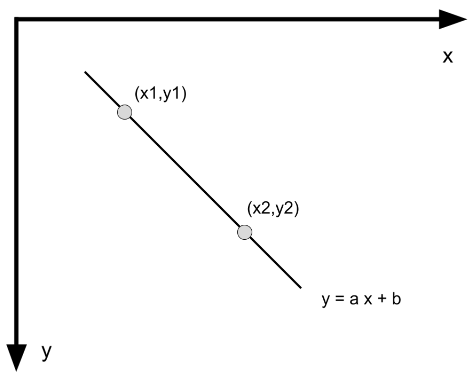


Figure 2.1: A straight line seen in the coordinate space with two points marked out, (x_1, y_1) and (x_2, y_2).

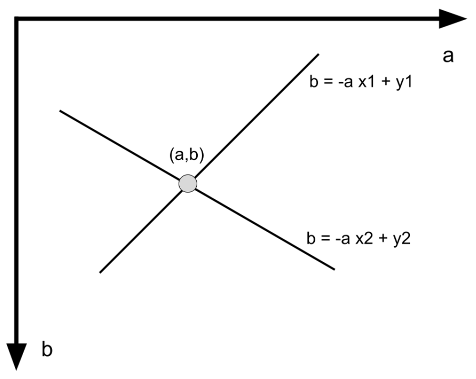


Figure 2.2: Corresponding lines for the points (x_1, y_1) and (x_2, y_2) in the parameter space, intersecting in a point (a, b), which are the parameters for the straight line in figure 2.1.

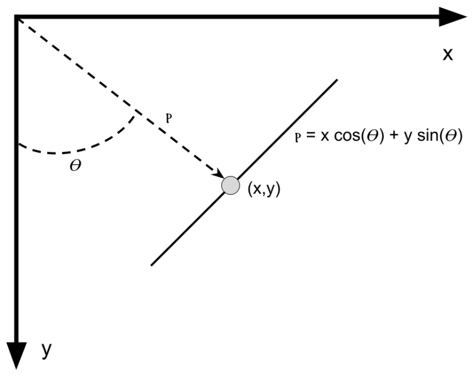


Figure 2.3: Illustration of an alternative way of defining a line, by using the parameters ρ and θ.

2.9 Performance metrics

For evaluating the performance of a classifier one has to choose a suitable metric. The metrics we have chosen to use in this paper are precision, recall, accuracy and F1-score. They are commonly used together with classification and are simple to calculate and to interpret, see [17] and [18].

Each class is interpreted independently, and the outcome of each class can either be a positive or a negative classification (binary). This is a two-class classification problem and the coincidence matrix can be seen in figure 2.4. In the figure, True positive (TP) means that the classifier, that generates the predicted outcome, has classified an image as positive, when it should, according to the ground truth. False positive (FP) means that the classifier, has classified an image as positive, when it should **not**. True negative (TN) means that the classifier, has classified an image as negative, when it should. False negative (FN) means that the classifier, has classified an image as negative, when it should **not**.

Given the coincidence matrix for a class, the precision, recall, accuracy and the F1-score can be calculated. The following definitions for them are respectively:

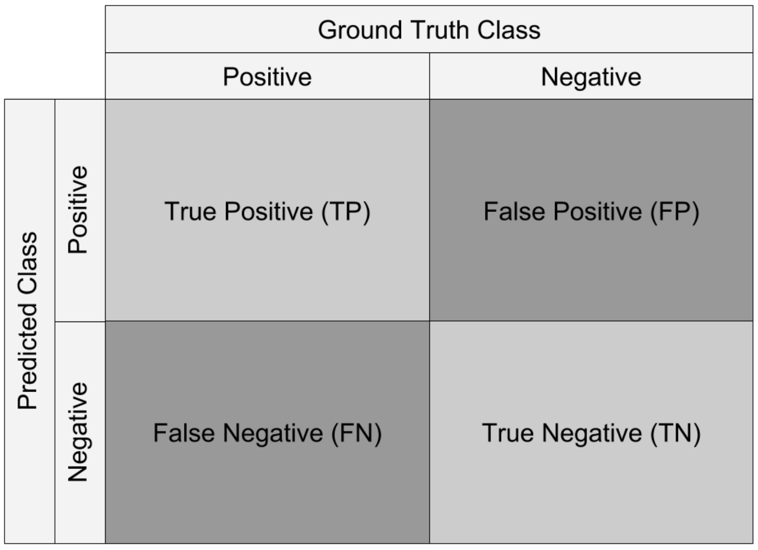
$$Precision = \frac{TP}{TP + FP} \tag{2.19}$$

$$Recall = \frac{TP}{TP + FN} \tag{2.20}$$

$$Accuracy = \frac{TP + TN}{TP + FP + FN + TN} \tag{2.21}$$

$$F1 - score = \frac{2}{\frac{1}{recall} + \frac{1}{precision}} \tag{2.22}$$

In other words, precision displays how many images that got a positive classification are right. Recall tells how many images, which are positive in the ground truth, were classified positively. Accuracy and F1-score estimates how many images received the right classification overall.



		Ground Truth Class	
		Positive	Negative
Predicted Class	Positive	True Positive (TP)	False Positive (FP)
	Negative	False Negative (FN)	True Negative (TN)

Figure 2.4: Possible outcomes for a two-class classification, resulting in this illustrated coincidence matrix.

Chapter 3
Methods

As mentioned in the introduction, this chapter will contain a description of how the used data were processed and collected. This chapter will also provide an overview of the data flow through the implemented algorithms, and the chosen approach to the given problem.

3.1 Data collection

Most of the output images and their corresponding reference images are in PNG-format. The exception to this are images given in TGA-format. These are converted to PNG via a small script since Matlab cannot handle TGA. The conversion is done in a lossless fashion, since it is of great importance to not add noise to the data in any way.

A debug feature in the GPU driver is used to dump the frame buffer to a file. The buffers contain frames of the 3D scene, that are being rendered on the screen. A reference image and a corresponding output image, see section 2.1, from the same frame and 3D scene are gathered this way.

3.2 Approach

Recently most types of image classification problems are solved by using different machine learning techniques. An article of such approach can be found in [19]. Since the given images are extracted from various arbitrary benchmarks and 3D scenes, and the types of emerging errors are similarly arbitrary in their appearance, the task of deciding suitable features for a machine learning algorithm is a very hard task. It is assumed that machine learning will most likely not give a satisfactory result, therefore, a pure image analysis approach was chosen. There are multiple reasons for this. The main reason is that the reference data is too diverse to converge into meaningful features, due to the shortage of collected data. Machine learning algorithms most often demand a large collection of data to perform well. Most of the time there only exists a couple of images per subclass, and in some cases only one. It is assumed that machine learning will most likely fail to classify the majority of the images, see chapter 6 for further discussion. A positive aspect with the chosen approach is that it does not need any further reference data, but then again it does not correct itself, when the collection of data expands, like a machine learning method would do.

With these assumptions in mind, we decided to develop algorithms that utilize image analysis methods. The problem was looked at on a class to class basis, by implementing the classification for each class of errors individually.

For the experimental evaluation, see chapter 5, the necessary data was first collected, as described in the section above.

We manually classified the given output images, containing errors, to act as a ground truth for what a specific class of error should look like. In this process, professional advice and perspectives from the CV team were given, especially on things such as how common or uncommon a specific error might be. An important observation was that one output image may contain several classes of errors at the same time. Therefore, a solution where all the classes are checked for each image pair was needed. After we spent time to read and understand enough relevant papers, we started to implement the algorithms. This was done in development cycles for each class, meaning that we spent a few days to find relevant facts and ideas about each class, implementing the algorithms and constantly testing them on all of the collected data. Testing the implemented algorithms one could minimize the number of false positives and maximize the number of images that should be classified as its corresponding class and subclass. This was done by adjusting thresholds, verifying the code, validating the outputs and thereby improving the accuracy of the algorithms.

After a development cycle has been performed for every class, we ran the scripts for the reference and test set to get the final results and execution times, see chapter 5. Some optimization of the already written and functioning code was performed to improve computation time and performance.

3.3 Data flow

This is an overview of the data flow of the algorithms. The adopted process of classifying image pairs (output and reference image) for the determined classes, with different kind of errors, is split up into three phases:

1. Preprocessing phase. The name hints that this is done before the classification and everything else. The major part of this phase is to set up the output text files for the classification and stating where the image pairs are located, to be able to process them in the following phase. This phase also includes the generation of the difference images, see section 2.3.

2. Identification phase. This contains the actual methods and algorithms for detecting the errors and update parameters used for the final classification.

3. Classification phase. This is the final phase where identified classes are compared with each other to point out a class that is the most dominant in the output image.

As the main tool, we used Matlab, with its associated image processing toolbox [20]. Matlab is a software used for algorithm development and in this case used to implement all the phases above.

3.3.1 Preprocessing phase

Before the execution of the Matlab scripts, the PSNR analysis detects any difference between the output image and its corresponding reference image, see section 2.2. If there is a difference, the main Matlab script is called and uses methods to generate necessary images for the next phase. A list of these methods is presented below.

General images generation

In this step, images, that are necessary for every class identification, are generated. This is done directly after PSNR gives a positive number for an image pair, which means that a difference between the image pair has been detected. From a given image pair the following images are generated:

- Euclidean difference image, see subsection 2.3.1.
- Binary difference image, see subsection 2.3.2.
- Signed difference image, see subsection 2.3.3.
- Edge map of the Euclidean difference, output, and reference image, see subsection 2.3.4.

3.3.2 Identification phase

At this point, the essential parts of the data needed for identifying the error classes are generated. More specific data is generated for the individual classifications from now on. In this subsection, every class is presented with its characteristics, a brief description, and observations of the class. This determined how each specific class was approached. The description of a class will include image examples, created to give a better understanding of the class. Note that these examples are not images produced from tests at ARM due to confidentiality reasons, but created by us and is a part of the test set to verify our solution in chapter 5. Figure 3.1 shows the reference image we used throughout the thesis to give examples of error classes.

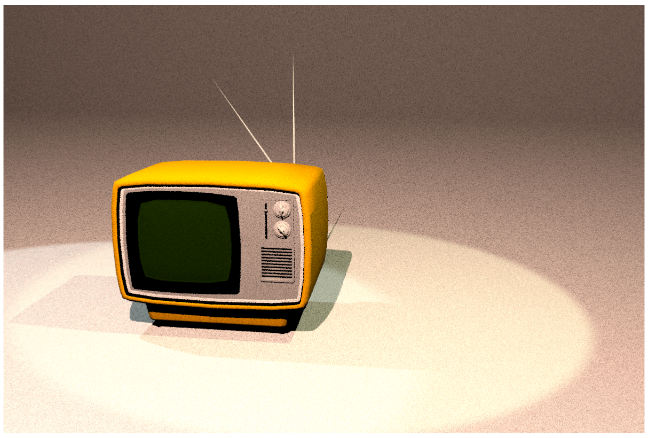


Figure 3.1: Reference image for the class examples. The image is also the reference image used for the first scene in the test data.

Characteristics of the Pixel class Output images with broken pixels are the target for this class. Broken pixels in the image are defined here as sudden (meaning non-gradual changes in this case) and significantly different color values compared to the reference image. This is often the color black/white/blue, which most of the time is a result of pixels keeping the color assigned from the image buffer clearing or a completely random color. This can either be interpreted as a small or a big scale pixel error:

- At a small scale, it means that the image has small clusters with broken pixels. The largest cluster is not larger than a 16x16 block of pixels (size of a tile).
- On a big scale, the majority of all pixels in an image are broken.

When classifying an artifact as this class, the pixels should not be following a geometry and in general be in a random pattern and placement.

Figure 3.2, 3.3, 3.4 and 3.5 are image examples of the Pixel class at a small and a big scale, with the previously given reference image as a reference, figure 3.1.

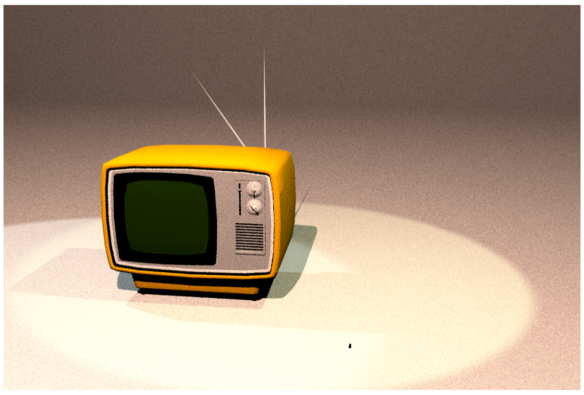


Figure 3.2: An example of the Pixel class at a small scale. Notice the error, which is a small sized black cluster to the lower right of the TV.

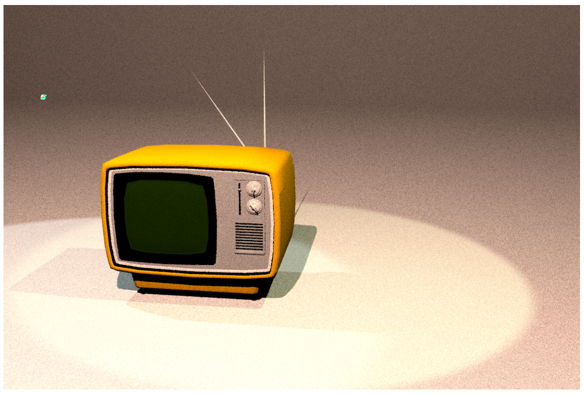


Figure 3.3: Another example of the Pixel class at a small scale. Notice the error, which is a small sized cluster with various colors to the upper left of the TV.

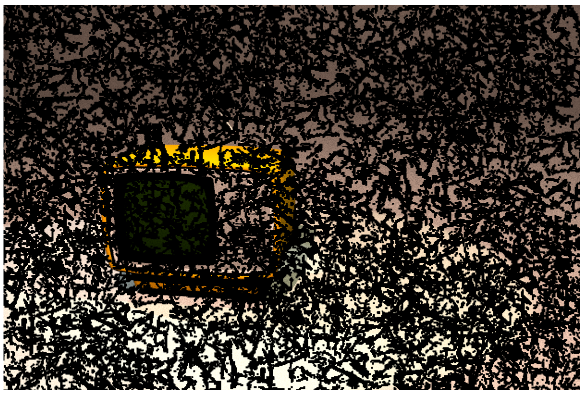


Figure 3.4: An example of the Pixel class on a big scale. The error in the figure is the black pixels over the majority of the image.

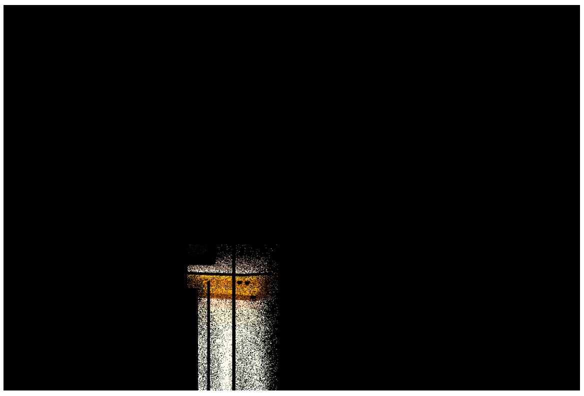


Figure 3.5: Another example of the Pixel class on a big scale. The error in the figure is the black pixels, that covers almost the whole image.

Characteristics of the Precision class

An image pair is classified as this if there exist small differences in color and/or in the position of textures, often along certain geometries. Like the pixel identification, this can be subdivided into a small and a big scale. As well as being on a small or big scale, the pixels with precision error could appear in a sporadic or non-sporadic pattern in the output image. Another subgroup to this classification is the output image having a faulty constant layer with an arbitrary color. Note that this is unrelated to the other subgroups. Below is a summary of the different subgroups:

- On a small scale, the image pair has most of its errors in a small range of intensity that is not visible to the human eye. It could also mean that only a few pixels, with precision errors, are present.

- On a big scale, most of the output image contains small differences between the output and the reference image.

- An image pair with a sporadic pattern appears to have "tiny dots" with a small difference, scattered in a random fashion around the image, see figure 3.6.

- An image pair with a non-sporadic pattern appears to have clusters with a small difference, often corresponding with a geometry at a specific position in the image pair, see figure 3.7.

- In the output image with a constant layer error, it appears to have an arbitrary color filter at a mostly constant rate over the image compared to its corresponding reference image, see figure 3.8.

Another instance of a precision error is when mipmapping goes wrong and picks out a down-scaled texture for an object, see figure 3.9 and section 2.4. Note that this is not classified as an individual subgroup. In many cases of a precision error, the difference is not visible to the human eye, but the error consists of slight changes in color. Image examples of this class can be found in figure 3.6, 3.7, 3.8 and 3.9.

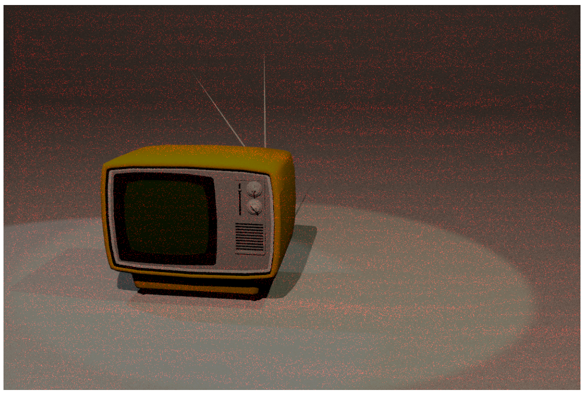


Figure 3.6: Example of Precision class with a sporadic error. For illustration purposes, a semi-transparent one channel BDI is layered over the reference image (as most precision errors of this sort are not visible to the human eye) with red pixels to represent errors. In this case, the errors are appearing randomly across the image.

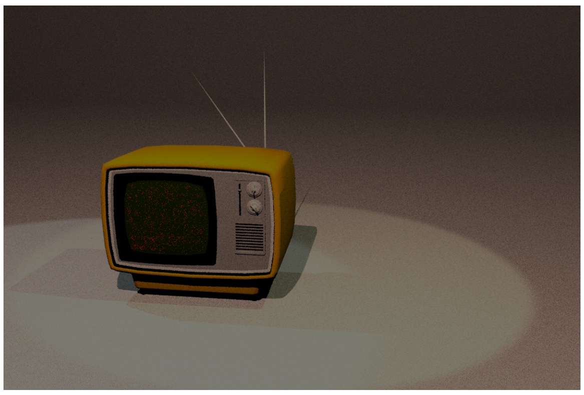


Figure 3.7: Example of Precision class with a non-sporadic error. For illustration purposes, a semi-transparent one channel BDI is layered over the reference image with red pixels to represent errors. The errors in the figure are following along the geometry of the TV screen.

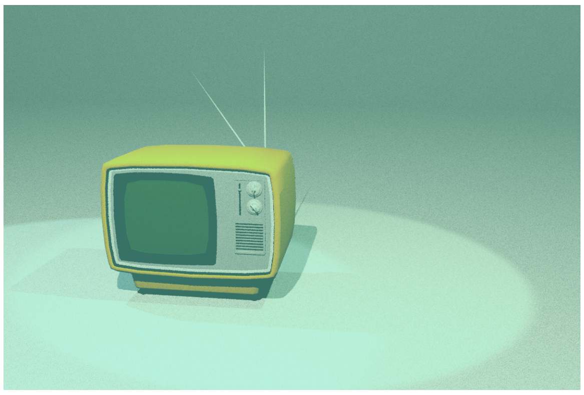


Figure 3.8: Example of Precision class with a faulty constant layer. The figure's error is a turquoise color filter over the image.

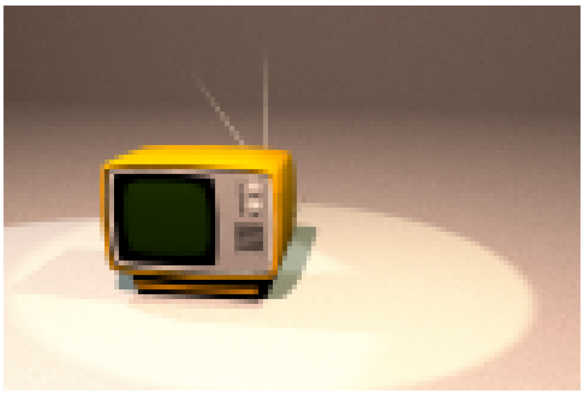


Figure 3.9: Example of Precision class with a faulty mipmapping level. Note that in a real input like this the mipmapping error is usually confined to specific textures and not to the whole image.

Characteristics of the Geometry class

This class takes whole objects into account or when there are major changes in the scene's geometry. Usually the errors to be classified to this class are:

- Missing geometries, see figure 3.10 and 3.11.
- Unexpected added geometries, see figure 3.12 and 3.11.
- Deformed geometries, see figure 3.13.
- Geometries with wrong color, or significantly faulty textures for whole geometries, see figure 3.14.

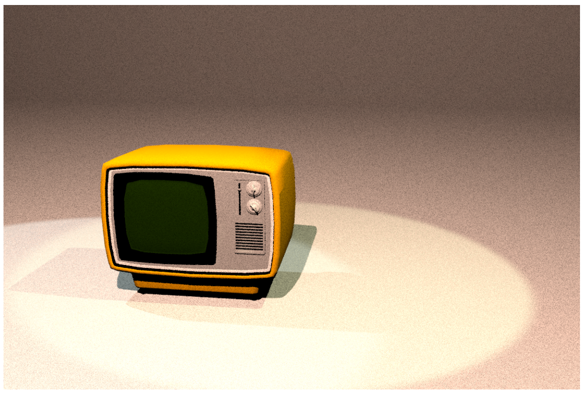


Figure 3.10: Example of Geometry class with a missing geometry. Notice that the antennas on the TV are missing.

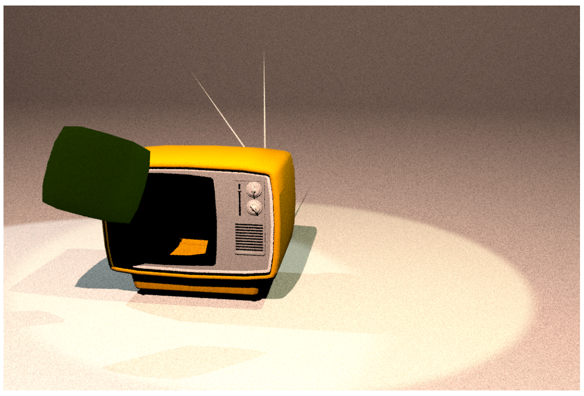


Figure 3.11: Example of Geometry class with a misplaced geometry. A combination of appearing and missing geometry. The error in the figure is that the TV screen has moved up to the left.

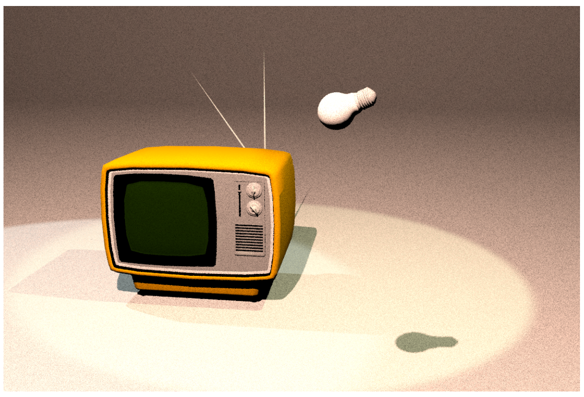


Figure 3.12: Example of Geometry class with an appearing geometry. The error in the figure is that a light bulb has appeared in the image above the TV.

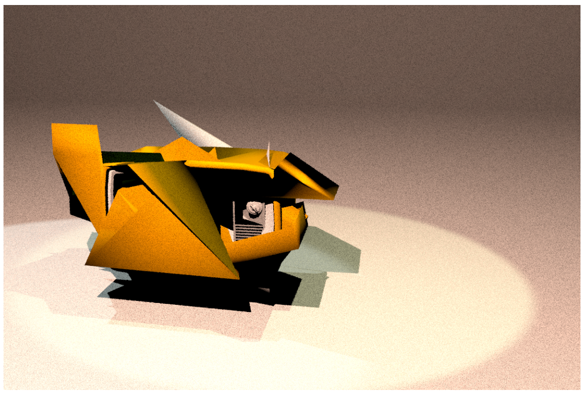


Figure 3.13: Example of Geometry class with a deformed geometry. The error in the figure is that vertices and faces of the TV geometry have the wrong position.

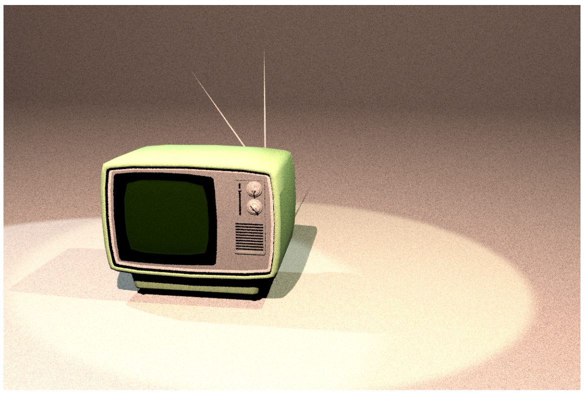


Figure 3.14: Example of Geometry class with a miscolored geometry. Notice the error, the TV body has changed color.

Characteristics of the Tile class

Clusters of errors appear in specific patterns over the image in square-shaped blocks. This class is a mixture of the pixel and the precision class because sometimes there are distinct corners around the block, and sometimes not. Tile errors may show up at fixed positions. A grid is here defined with its bars spaced equal to a tile's size, both in x- and y-direction. Note that the starting pixel position (upper left corner of a tile) of an error tile, P_{Start}, has the following property: $P_{\text{Start}_i} \bmod 8 = 0, P_{\text{Start}_j} \bmod 8 = 0$. A visualization of the grid can be found in figure 4.11.

The tile class can be divided into two subgroups:

- Fixed positioned tiles. One could look for a staircase-looking pattern because the blocks are positioned in conjunction to a fixed grid. Distinct corners around the block often show up, see figure 3.15.

- Unexpected angular appearance. If the blocks do not coincide with the grid, it is probably this subgroup.

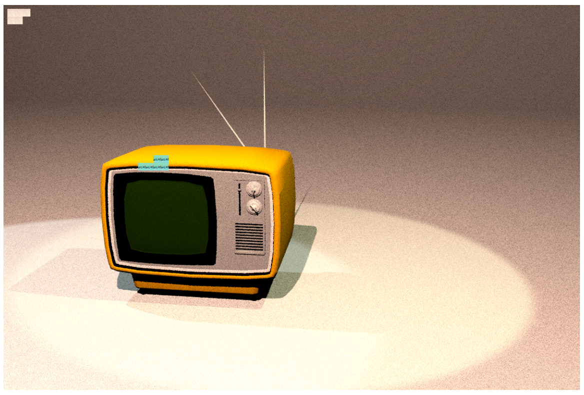


Figure 3.15: Example of Tile class (smallest error tile is of the size 8×8). The errors are the miscolored tiles in the upper corner and on top of the TV screen.

Characteristics of the Primitive class

When parts of an object receive an incorrect color or shading it is usually caused by a few faulty primitives. The error appears in a triangular shape in the case of one single primitive and in polygons with few corners relative to a geometric object in the case of multiple primitives.

The primitive class can be divided into the following subgroups:

- Faulty primitive. This subclass contains two primitive related errors. Faulty positioned vertices and faulty faces of a primitive. Faulty positioned vertices are vertices with a faulty position that generate triangle-shaped errors. See figure 3.16 for illustration. Faulty faces of a primitive is when faces of a primitive are either transparent or colored in a faulty way. See figure 3.17 for illustration.

- Visible mesh. This can either be the case when an object's mesh become visible by having an arbitrary assigned color or errors are aligned along the mesh. A mesh is defined as a polygon mesh, which is a collection of vertices and edges connecting the vertices. This defines a shape in 3D-space consisting of polygons, usually triangles. See figure 3.18 for illustration.

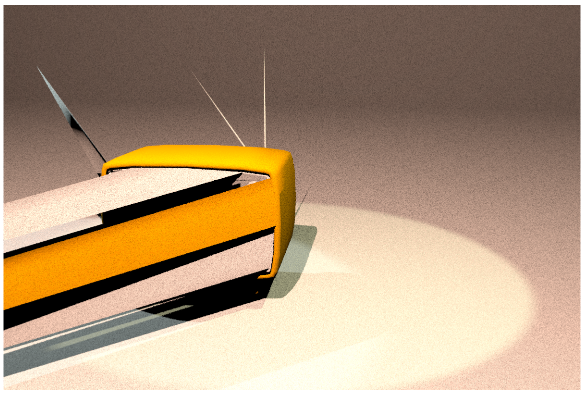


Figure 3.16: Example of Primitive class. Notice the error, some of the vertices in the primitives on the TV have faulty position values.

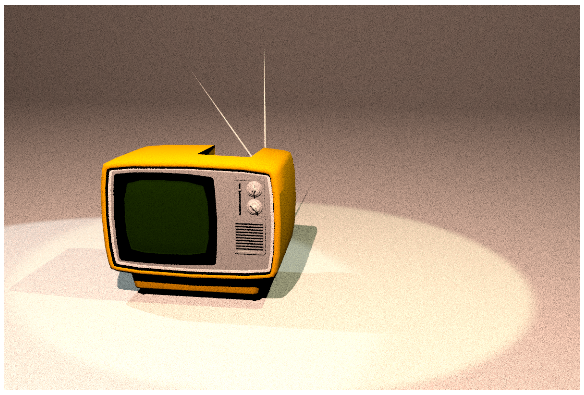


Figure 3.17: Example of Primitive class. Notice the error, some of the faces of the primitives on the TV's top side have not been rendered.

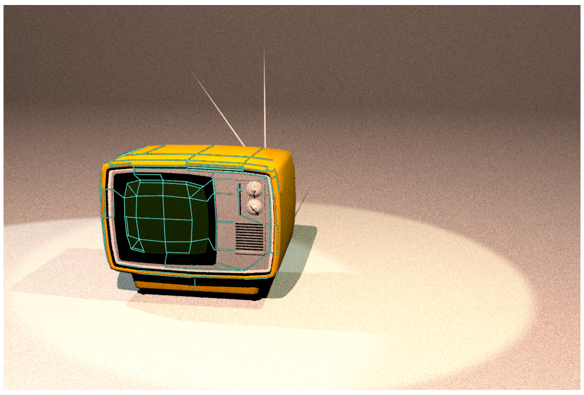


Figure 3.18: Example of Primitive class. Notice the error, parts of the mesh is outlined on the TV.

Characteristics of the Lighting class

This class represents output images with major light or light effect errors at several places in the image.

Faulty rendered reflections could be a part of this class, but since the error usually lies in that which is reflected, the error should be viewed as that specific error instead. For instance, if an object is rendered in a faulty way and this shows in the reflection, it should be classified as Geometry.

- Major lighting error. This type of error has pixels with shadow or light related errors at most of or the whole output image. See figure 3.19 and 3.20 for illustration.

- Spotted lighting error. Small spots of pixels with shadow or light related errors at several positions in the output image.

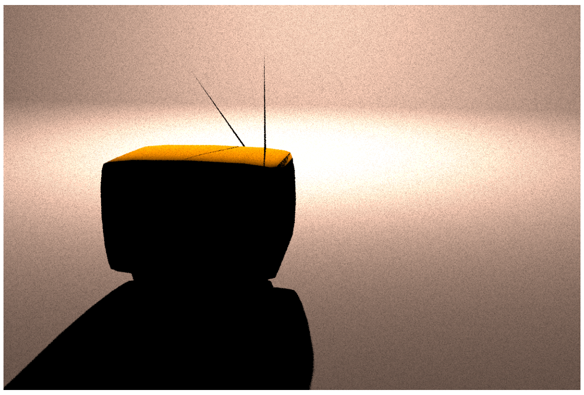


Figure 3.19: Example of Lighting class. The error covers most of the image because of a bright light behind the TV makes the TV cast a sharp shadow.

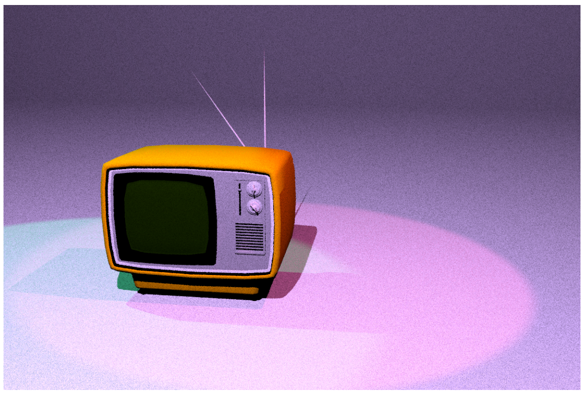


Figure 3.20: Example of Lighting class. Notice the error, the lights in the scene have different colors, and therefore makes the whole image different.

Characteristics of the Alpha class

An error is identified as an Alpha error if the transparency in the alpha channel for the output image is different from the reference image's.

User interfaces (UI) and transparent textures, with any kind of error, usually goes in this class.

3.3.3 Classification phase

The aim of this phase is to eliminate false positives from the identification phase. All identified classes contribute with their metadata, that could be for instance the number of present edge pixels or a specifically found subclass, to filter out the most obvious false positives using information from all classes at once. The classes with a big intensity error, which are pixel, geometry, tile, primitive and lighting, are prioritized over the classes with a small intensity error, which is only precision.

The alpha class cannot be compared with the other classes, because the error is in an image's alpha channel, which is separate from the RGB channels. Therefore the alpha class is added as a most likely dominant class with one other class mentioned if it is identified.

Chapter 4

Implementation

The chapter is structured by going through every class. By first giving a high-level description of its corresponding algorithm, and then a motivation behind the implementation, the implementation of our developed algorithms will be introduced. Figure 4.1 describes the most important binary images, we will call them masks, throughout the implementation. The figure is supposed to act as an overview when reading about how the different classes are implemented. It is placed here at the beginning of the chapter because it belongs to several classes, not only one, and could be ignored for the first time seeing it.

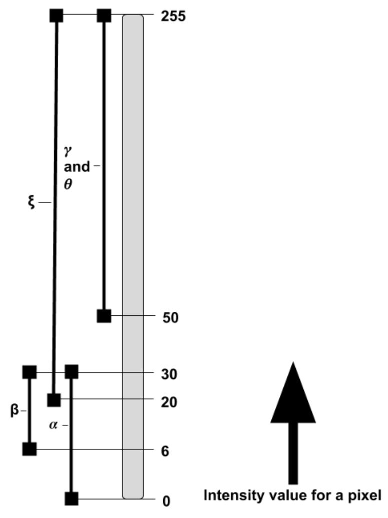


Figure 4.1: Visualization of value ranges of masks used in our solution. The figure shows which mask an arbitrary pixel belongs to depending on its intensity value in the EDI. γ can be found in section 4.1, α and β can be found in section 4.2, θ can be found in section 4.3 and ξ can be found in section 4.5.

4.1 Pixel class

4.1.1 High-level description

From the characteristics of the pixel class in subsection 3.3.2, all the subclasses were implemented.

The algorithm finds pixels with a sudden change between the reference and the output image. These pixels are contained in γ, see figure 4.1. The found pixels are grouped by 8-connected segmentation, see subsection 2.5.2. The segments that have a gradual change at the borders of the segment are removed. The remaining segments are then iterated segment-wise to determine if a segment belongs to a small or a big pixel error.

4.1.2 Motivation

Choosing a good threshold for obtaining a desired γ was first decided by analyzing the values in the EDI, relating to pixel and precision errors. The resulting γ did not become as precise as desired. Instead of just testing different thresholds and adjust after, histograms of the EDI were created, see figure A.1 in appendix A. The chosen lower threshold for γ was a value that separated the small from the big intensity values, see figure 4.1 at intensity value 50.

To group neighboring pixels in γ, 4-connected segmentation was first used. As 4-connected segmentation generated many small and unnecessary segments, the segmentation was switched to 8-connected. Since the algorithm executes segment-wise to identify a subclass, having fewer segments will shorten the compute time. This makes a significant difference for some big pixel errors with images similar to figure 3.5.

In the early stage of the development of this algorithm, only γ was used to identify the pixel class. Since geometry class errors also generate big intensity differences, this algorithm would identify those as pixel errors. Facing this problem, an important discovery was found. A pixel error is always sudden. This property varies from the geometry class since geometry related errors often are blurred out at its edges. By removing segments, that have gradual changes at its edges, only the sudden segments would remain. One can see this in figure 4.2, 4.3, 4.4 and 4.5.

After testing the solution of removing segments, that is not sudden, it was discovered that some pixels did not get removed when they should be. As illustrated by figure 4.6 and 4.7, these pixels was looked at a more detail.

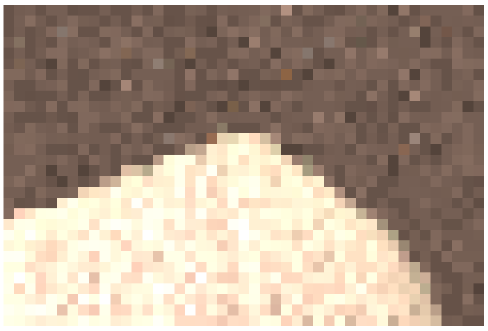


Figure 4.2: Example of an output image (zoom in of upper right corner of the appearing light bulb in figure 3.11) containing an error with a gradual change due to an appearing geometry.

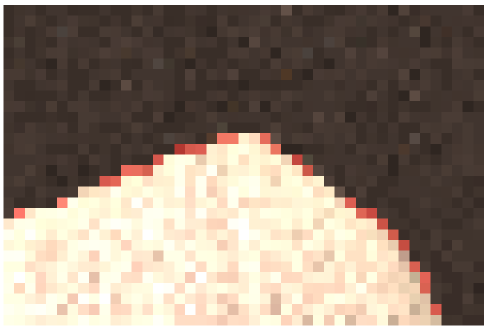


Figure 4.3: Visual representation how γ is used in the solution, by adding a semi-transparent layer to figure 4.2. The totally transparent pixels, compared to figure 4.2, represents non-zero values in γ. The darker semi-transparent pixels represent zero values in γ and zero values in the EDI. The red semi-transparent pixels represent zero values in γ and non-zero values in the EDI. This segment will be removed, as the red semi-transparent pixels are neighbors to the non-zero-segment in γ.

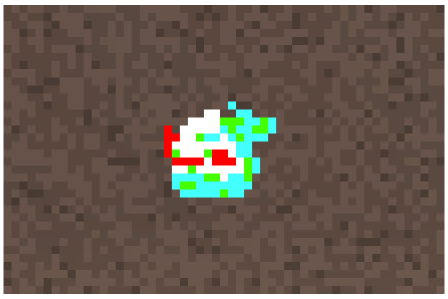


Figure 4.4: Example of an output image (zoom in of pixel error in figure 3.3) containing an error with a non-gradual change due to a pixel error.

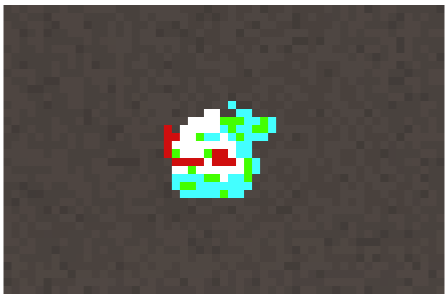


Figure 4.5: Visual representation how γ is used in the solution, by adding a semi-transparent layer to the figure above. The totally transparent pixels, compared to figure 4.4, represents non-zero values in the mask. The darker semi-transparent pixels represents zero values in the mask. Note that there are no gradual changes in this segment's edge, therefore, no semi-transparent red pixels as in figure 4.3, hence the segment will not be removed.

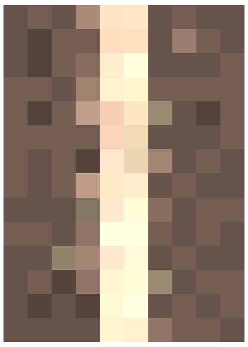


Figure 4.6: Zoom in of the upper part of the right antenna in the reference image, see figure 3.1. In figure 3.10 this antenna is missing, this results in the segmentation of the errors seen in figure 4.7.

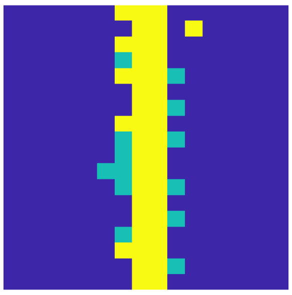


Figure 4.7: Here one can see pixels belonging to γ in yellow; pixels that do not belong to γ, but still have positive difference values of the BDI in turquoise; and pixels with zero values in blue. The larger yellow segment here will be removed as it has neighboring pixels with non-zero values in the BDI. The single yellow pixel here will also be removed since it lies in the close vicinity of a geometry error segment.

4.2 Precision class

4.2.1 High-level description

From the characteristics of the precision class in subsection 3.3.2, all the subclasses were implemented.

The algorithm finds pixels, that have a small difference in intensity between the reference and the output image. These pixels are contained in either α or β, where β is a subset of α: $\beta \subseteq \alpha$, see figure 4.1. Depending on the characteristics of α and β, the error will be identified as a big, small, sporadic or non-sporadic precision error. In other cases, an error may be identified as a constant layer precision error if the majority of the faulty pixels in the EDI are close to a constant color.

4.2.2 Motivation

At the start of the development of the algorithm, faulty geometries sometimes generated false precision errors. For example, when geometries are appearing in the output image, the added geometries could have a similar color to the background, and therefore add false precision errors to the masks. When geometries are missing, the background, that should not be visible, could also be affected by precision errors. This results in false precision errors. An approach that proved to be good, even in the implementation of the geometry class itself, was to ignore these errors using the BDI mask to detect these geometries.

As specified in the paragraph about the precision error, see subsection 3.3.2, the meaning of the term sporadic is referring to "tiny dots" of errors, that are scattered around the output image. By "tiny dots" we mean single pixels. An intuitive way to see a sporadic pattern of dots was that the dots in α should together form a mean point close to the image's center. This would mostly occur if the dots were evenly distributed around the output image.

The reader may have noticed that the α and β mask are very similar in their intensity span, see figure 4.1. Identifying big precision errors, the result became better when excluding the smallest of errors in intensity. As many of our images in the reference set have small errors in intensity, many images would be identified as a big precision error if those errors were not excluded, by using β instead of α. In other cases, such as trying to find the sporadic, non-sporadic and small precision errors, only α was needed. This is because a large portion of the errors in these subclasses has a very small intensity value in the EDI and could not get detected in the β mask.

4.3 Geometry class

4.3.1 High-level description

From the characteristics of the geometry class in subsection 3.3.2, only missing, appearing and miscolored geometry subclasses were implemented.

The algorithm finds pixels, that are present in θ, which contains pixels with a big difference in the EDI and has an error in all of the BDI's color channels. The found pixels are grouped and iterated segment-wise. A segment is identified as an appearing or missing geometry, depending on how many pixels that are present in the outer edges of the faulty geometry in the output or the reference edge map, respectively. In other cases, an error may be identified as a miscolored geometry error if the error follows the reference image's edge map, and that the majority of the faulty pixels are close to a constant color.

4.3.2 Motivation

First θ was identical to γ in section 4.1. As this version of the mask would capture too many irrelevant pixels, the usage of the BDI was included. Since a faulty geometry is most likely to give an error in all channels, only pixels, that are present in all color channels of the BDI, are included in θ.

In some cases errors coincide with the reference image by having the same color, resulting in gaps in segments of θ, or zero-segments so to speak, that is not desired. The zero-segments could generate more segments, making computing time longer, or even falsely identify a big pixel error for a geometry error. To counter this, one has to fill those zero-segments. An illustration of this can be seen in figure 4.8, 4.9 and 4.10. Since the blue areas in figure 4.9 are not desired, because for example the TV screen just happens to be black as the error in the output image, those areas have to be filled.

At the point when the zero-segments have been filled, θ is most likely to contain small segments, that did not connect with a bigger segment. These segments will most likely be detected as geometry errors. Therefore are those segments removed from θ before iterating. Experimenting with this, it was found that geometries sometimes appear as small segments. This occurs when a geometry error is for example far away or a particle. In most of these cases for geometry errors, they are "slender", meaning that their area of pixels is spread out. If this is the case, it indicates that the segment is most likely a geometry error by comparing with the other classes' characteristics. One important property these segments must have is a gradual change. This is checked in a similar manner as for the pixel identification, but the opposite result is expected, see section 4.1.

A faulty geometry has most likely a texture. The texture itself could generate edges in the output image's edge map. This will give false edges inside the geometry and may change the outcome of the identification. Only the relevant parts are interesting, which are the edges that surround the faulty geometry. The found solution for this was to perform a dilation of the segment to exclude the pixels inside the segment's outer edges.

At this stage, one knows that there is a faulty geometry at a segment. The decision to determine if a faulty segment is a missing or an appearing geometry was clear. The difference between the edge maps for the output and the reference image at the relevant parts granted the desired information. For pixels that belong to an appearing geometry, they are visible in the output image but not in the reference image, and for pixels that belong to a missing geometry, it is the opposite. Which subclass a segment belongs to becomes simple to check, by counting which image has the most visible pixels and identify the error correspondingly.

For the third subclass of the geometry class, miscolored geometry, the whole image is checked at once, instead of iterating segment-wise. Since this subclass was rare in the reference data, the algorithm to detect it became tailored according to that small set of data. It was also found when experimenting that iterating the whole image gives a shorter computation time compared to iterating segment-wise.

It is worth mentioning that the computation algorithm of the edge maps throughout the geometry class was changed from Sobel to Canny. Working with the Sobel operator gave fewer edges than desired, which made the geometry class clumsy. It was therefore needed to switch to another more sensitive operator, like Canny. The difference between them can be read in section 2.3.4.

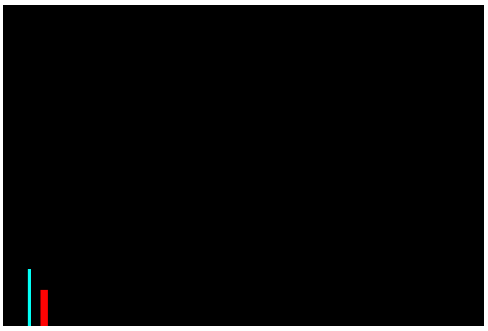


Figure 4.8: A big pixel error consisting of mainly black pixels with some seemingly random lines at the bottom left corner.

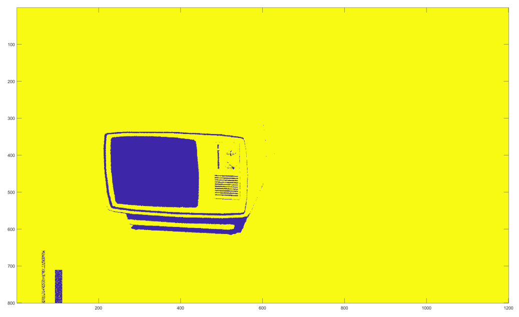


Figure 4.9: The resulting θ when evaluating figure 4.8. The yellow areas represent ones in the mask and the blue zeros.

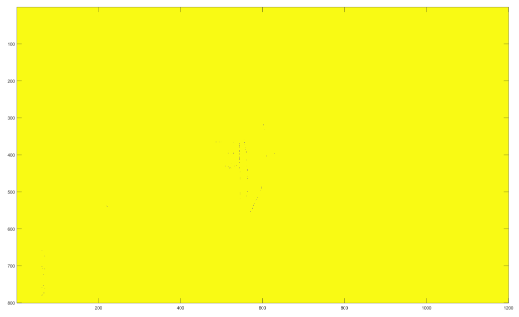


Figure 4.10: The resulting θ when the faulty zero-segments are filled. The yellow areas represent ones in the mask and the blue zeros. Note that there still are some single zero pixels present.

4.4 Tile class

4.4.1 High-level description

From the characteristics of the tile class in subsection 3.3.2 only fixed position tile subclass was implemented.

The algorithm finds pixels that form distinct corners at fixed points in the output image. These pixels are contained in ω, which is a subtraction between the output's edge map and the reference's. If the algorithm finds any pixels, the error is identified as a fixed position tile error.

More specifically, the fixed points are specified as shown in figure 4.11, where every 4-junction is iterated and checked if two straight lines connect in one of the grid's 4-junctions.

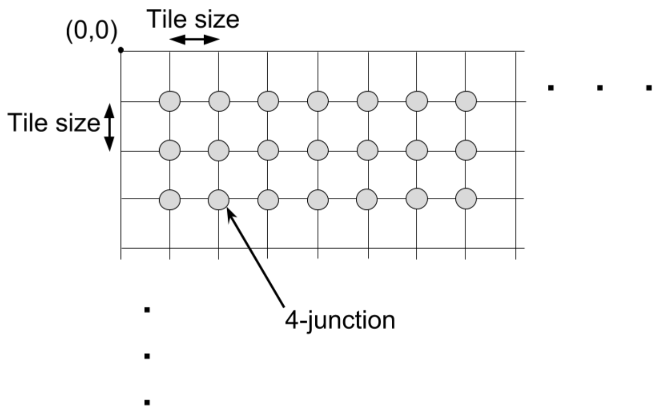


Figure 4.11: Visualization of the grid, that captures the fixed position tile errors in ω. Origin starts at an image's upper left corner.

4.4.2 Motivation

Searching for tile errors, it is assumed that the change in intensity, at the border of two tiles, is large enough to generate a clear edge. So after subtracting the edge maps, the faulty edges in the output image are visible and can be interpreted.

The first thought, that came to mind, was to find the square-shaped tiles directly. Since a faulty tile almost never appears with all its corners clearly visible, searching for squares is a more complex and time-consuming task than just search for a corner that aligns with the grid defined by the size of a tile.

One big issue with this solution was the number of false positives. Every 4-junction was iterated and it was not unlikely that a non-tile error happens to have a corner at a 4-junction. To remove false positives, the algorithm segments a small area around where there is a possible corner and evaluates the segment's edge map. Since the identification is quite accurate, the only necessary check after that is to check if any corner was registered.

4.5 Primitive class

4.5.1 High-level description

From the characteristics of the primitive class in subsection 3.3.2, all the subclasses were implemented.

The algorithm finds pixels present in ξ, which contains pixels with a moderate intensity error in EDI and have errors in all color channels in the BDI, see figure 4.1. The found pixels are grouped into segments and iterated segment-wise. If the segment closely resembles the skeleton image of the segment, see section 2.7, the image is classified as a primitive error with the visible mesh error subclass. Otherwise, the edge of each segment is searched for straight lines. Each straight line found is paired with other straight lines that form a triangle together. Triangles which are not aligning with the segment are discarded. The areas of the largest triangles, that do not intersect with each other, are found and summed. If this sum divided by the area of the segment is above a certain ratio, the image is classified as a primitive error with the faulty primitive subclass.

4.5.2 Motivation

The threshold used in ξ is lower than that used in θ, due to the fact that it is of higher importance to capture the whole faulty segment in the case of a faulty primitive subclass as it may otherwise not be seen as a primitive shape. In the case of geometry classification, this is not of the same importance, since one only cares about the correlation of edges and not so much the exact shape of the segment. This means that if the segment in the geometry classification is an approximation of the real segment, the classifier will with a high probability in at least one of its outer areas find a correlation of missing or appearing edges.

In cases of the visible mesh subclass, one can see that the similarities between a skeleton image of the faulty segments, see section 2.7, and the faulty segments themselves are almost identical. There will only be such similarity when the segments are line-shaped, see image 3.18.

Developing the implementation of the primitive class, it was obvious to take advantage of the characteristics of the primitive class. Most of the images with a faulty primitive error have more sharp edges, forming triangles, compared to an arbitrary geometry segment. To ensure that most of a segment had sharp edges, and therefore know that the error is a faulty primitive error, the areas of the triangles aligning with the edge of the segment were compared to the total area of the segment. In most cases, segments with a geometry error have more non-linear edges, which often form multiple small triangles, in relation to the total area of the segment. On the other hand, segments with a faulty primitive error have fewer corners and more straight lines for their edges, resulting in a few large triangles, also in relation to the total segment area.

To find the straight lines, a first attempt was made using the Hough transform, see section 2.8. However, this proved to not be satisfactory as it did not capture the correct lines necessary for a reasonable estimate of the triangles that we want to capture. Thus, a custom algorithm for straight line searching was developed. This algorithm uses the linear equation to determine if pixels belong to a line and if there are enough pixels to form a straight line.

It was discovered early that many found triangles were to be discarded as they did not represent an area inside the segment. Before describing the motivation further and presenting examples, there are some concepts and keywords which should be cleared out. In figure 4.12, an example is given of what triangle is actually referred to when describing the triangle created by two lines, as well as what point is meant when referring to a triangle's start point.

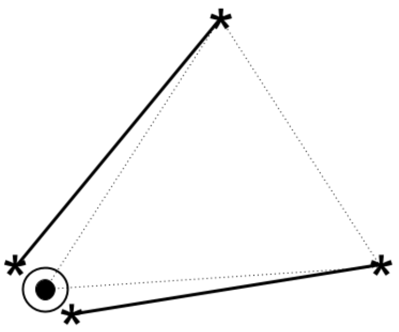


Figure 4.12: Visualization of two found lines, marked as solid lines, with their start and end points marked with stars. The centroid of the closest end or start point of the two lines, marked with a dot and a surrounding circle, will act as the triangle start point. The triangle whose area will be analyzed is the triangle marked with dotted lines.

To know if a triangle is referring to an area outside the segment, it was determined whether the centroid of the three points of the triangle was inside the given segment or not. An example of this can be seen in figure 4.14, where one can note that no triangle start point was produced in the area between the line points at $x \approx 175, y \approx 275$.

It is also worth to mention that the same area covered by triangles should not be counted twice, and thus only the largest triangles with start points not in close proximity to a larger triangle's point, should be kept. An example of this can be seen in figure 4.14, where the triangle with a start point at $x \approx 145, y \approx 270$ is not used (not marked in cyan) as the start point is in close range to the endpoint of a larger triangle (whose start point is marked at $x \approx 65, y \approx 125$).

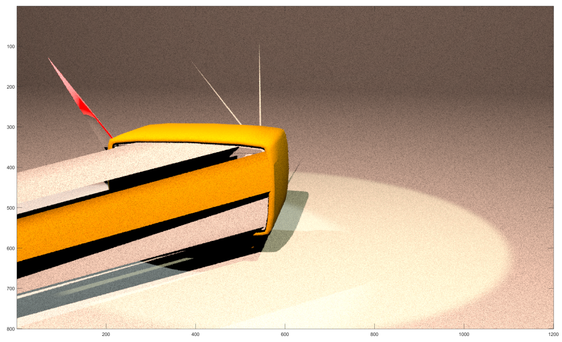


Figure 4.13: Visualization of the faulty primitive segment belonging to ξ layered in red on top of the image 3.16.

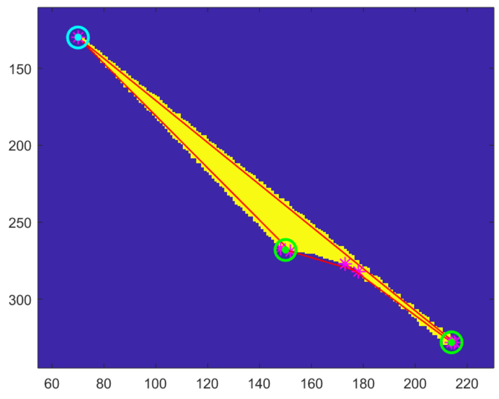


Figure 4.14: Visualization of the straight lines found at the edges of a segment (marked in red, start and end points of the lines marked as stars in magenta color) and the triangle start points (marked as points with circles). The triangle start point that is used is marked in cyan and the two unused marked in green.

4.6 Lighting class

4.6.1 High-level description

From the characteristics of the lighting class in subsection 3.3.2 only major lighting error subclass was implemented.

The algorithm finds pixels that have an error in every color channel in the output image. If the found pixels are the majority of the output image's pixels, and the output and the reference image's edge maps have both similar edges and edges that differ, then the error is identified as a major lighting error.

More specifically, edges that differ are edges that appear in the output image's edge map, but not in the reference's, or vice versa. Similar edges are the opposite of edges that differ. This means an edge in the output image that appears in the reference image or vice versa.

4.6.2 Motivation

At first sight, major lighting errors may look like a subclass to a geometry error. This is not entirely true, as it most likely does not follow geometries that cover the output image, but is present in the majority of the output image, which is the main criteria for this subclass.

It was quickly discovered that big pixel errors are almost covering the whole image, similar to major lighting error. The found difference between a major lighting error and a big pixel error is that the big pixel error has almost no similar edges when comparing the images.

An important understanding of the major lighting error is that it does not only take into account if the majority of the image's pixels are faulty but also if these have major intensity differences. The edges in the output compared to the edges in the reference image will in these cases vary more than they coincide. By determining that this is the case, it will constrain images that have a large enough intensity error to be able to change the edge map, while still depicting the same geometry as a certain number of edges still need to coincide.

4.7 Alpha class

4.7.1 High-level description

The algorithm extracts the alpha channel, also known as the transparency channel, as a matrix from both images. If these matrices differ in any way, the image is classified with an alpha error.

4.7.2 Motivation

This is the smallest classifier of all the classifiers. It could be any kind of error involved with the transparency (alpha) channel. Note that this classification can never be wrong as it should always classify positively if there is a difference in the alpha channel since only this class looks at the channel.

Chapter 5
Experimental evaluation

This chapter presents the performance of the classification. The primary metrics of measuring this will be the number of correct classifications of each respective class and the time used for the classification. All results in this section are generated from the proposed solution.

The scripts for the classifications were executed with a dataset containing 112 output images with their corresponding reference images, acting as a reference set, and a test set with 36 images. The reference set is the original data set provided by Arm, that was used to determine the classes and construct the classifiers. It was acquired as mentioned in section 3.1, while the other set consists of manually created images, created with Blender. This set consisted of the different class errors applied to three different scenes. The faulty images of the first scene are referenced for visualization purposes in this paper, see chapter 3, but also used for testing of the classifiers as mentioned. The reference image for this scene can be seen in figure 3.1. The reference images of the two other scenes are showed in figure 5.1 and figure 5.2. These were created, in terms of their appearances, with the aim to give a more varied data. The second scene, see figure 5.1, is, therefore, darker than the first. The third scene, see figure 5.2, is more filled with textures (especially the wooden and wall surfaces) and is of different proportions compared to the other two. If one would want to take a closer look at these images, they can be found at [21].

It is worth to mention that not all images in the reference set were used to train our algorithms, and therefore a portion of this set could also be viewed as a part of the test set. They are not included in the actual test set, but one should keep this in mind when analyzing the results.

The tables 5.1 and 5.2 contain models of the time to execute each respective classifier, including the average time and the standard deviation for the executed code and the average time to manually classify an artefact of respective class, estimated by the CV team, see section 6.1. For the reference set, the average time and standard deviation when executing all of the classifiers in table 5.1 is 4.44 seconds respective 19.659 seconds. For the test set, the average time and standard deviation when executing all of the classifiers in table 5.2 is

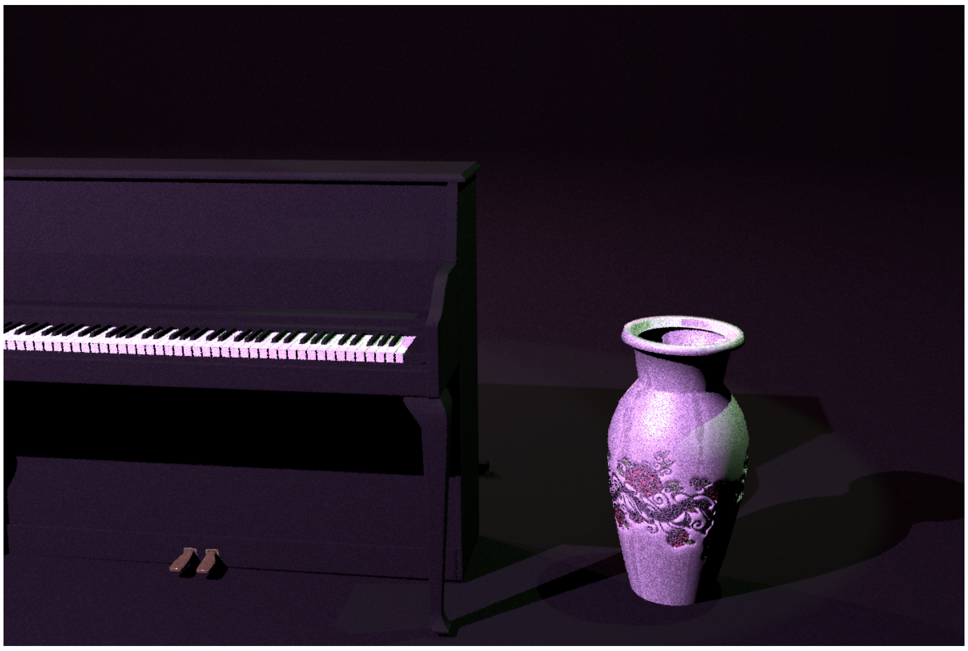


Figure 5.1: The reference image used for the second scene in the test set.

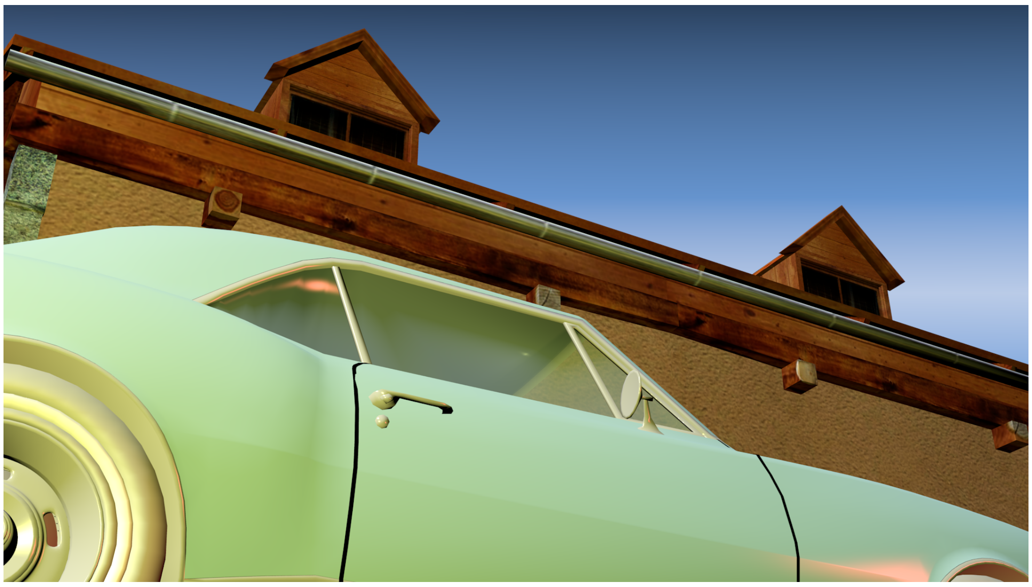


Figure 5.2: The reference image used for the third scene in the test set.

2.434 seconds respective 8.248 seconds.

The tables 5.3 and 5.4 are the classification results when executing each classifier individually. The classification results are generated by comparing the algorithms predicted classification, a binary number whether it is the specific class or not, and the ground truth classification, also a binary number whether it is the specific class or not. The metrics used for this evaluation are the commonly used recall, precision, accuracy and F1-score, see section 2.9. The total column is a sum for the integer values and a mean for the per-

Table 5.1: Average time with its corresponding standard deviation when executing every class script separately for the reference set. Measured in seconds.

	Pixel	Precision	Geometry	Tile	Primitive	Lighting	Alpha
Average time	1.653	0.561	13.607	1.452	13.498	0.293	0.031
Standard deviation	1.408	0.559	42.581	1.253	25.933	0.159	0.021
Manual classification	5	30	5	30	5	5	5

Table 5.2: Average time with its corresponding standard deviation when executing every class script separately for the test set. Measured in seconds.

	Pixel	Precision	Geometry	Tile	Primitive	Lighting	Alpha
Average time	0.874	0.168	8.201	0.826	6.775	0.202	0.002
Standard deviation	0.382	0.104	18.049	0.111	9.308	0.053	0.0002
Manual classification	5	30	5	30	5	5	5

centile values.

Table 5.3: The number of output images classified correctly or incorrectly according to its corresponding ground truth class, for the reference set.

	Pixel	Precision	Geometry	Tile	Primitive	Lighting	Alpha	Total
True positive	13	80	56	9	13	3	13	187
True negative	72	4	27	95	63	99	99	459
False positive	27	28	21	2	33	1	0	112
False negative	0	0	8	6	3	9	0	26
Precision (%)	32.5	74.1	72.7	81.8	28.3	75.0	100	62.5
Recall (%)	100	100	87.5	60.0	81.3	25.0	100	87.8
Accuracy (%)	75.9	75.0	74.1	92.9	67.9	91.1	100	82.4
F1-score (%)	49.1	85.1	79.4	69.2	42.0	37.5	100	73.0

Table 5.4: The number of output images classified correctly or incorrectly according to its corresponding ground truth class, for the test set.

	Pixel	Precision	Geometry	Tile	Primitive	Lighting	Alpha	Total
True positive	5	10	22	3	4	3	3	50
True negative	32	8	17	42	29	45	47	220
False positive	11	29	9	4	12	1	0	66
False negative	2	3	2	1	5	1	0	12
Precision (%)	31.3	25.6	71.0	42.9	25.0	75.0	100	43.1
Recall (%)	71.4	76.9	91.7	75.0	44.4	75.0	100	80.7
Accuracy (%)	74.0	36.0	78.0	90.0	66.0	96.0	100	77.6
F1-score (%)	43.5	38.4	80.0	54.6	32.0	75.0	100	56.2

Chapter 6
Discussion

In this chapter, the results in chapter 5 are discussed and analyzed. This is done by investigating if the goal and scope of the thesis were met and analyzing the overall performance of the implementation. Thoughts about subclasses that were not implemented, why some classes were harder to classify than others and other important considerations one has to think about when viewing the classification results is also discussed here. Finally a short discussion about which optimizations one could do to the algorithms and notable thoughts we had during the project development are presented.

6.1 Performance of classification

The main goal and scope of the thesis were to implement algorithms that detect and classify different graphical artefacts and quantify the result by some metric, compared with the PSNR method, which does not provide such information. With a total of 82% and 78% accuracy for the reference respective the test set for seven different classes, we see this goal as fulfilled. The results are measurable and quantified as seen in tables 5.1, 5.2, 5.3 and 5.4.

One could argue that our created test set is not realistic and also biased. It is true in a way. As mentioned, we are working with a very small set of images to develop our algorithms, even only one image per subclass in some cases. It is therefore needed to create more images in the same manner and not divide the given data into a test and a training set as one would normally do. Since we have studied the errors at a very detailed level, we create our images to have the same characteristics as the reference data. We still have a great variation between images with the same artefact, both in scene selection and the artefacts themselves. For example figure 3.1 is a relatively bright scene, figure 5.1 is a relatively dark scene and figure 5.2 is a relatively texture-heavy scene.

Analyzing the tables 5.3 and 5.4 one can see that the precision metric receives the worst results from the primitive and pixel classifiers and also from the precision classifier in table

5.4. This suggests either that these classes are not specified enough and have too loose constraints when classifying an image, or the ground truth is lacking partially incorrect, meaning that the false positive is a truly positive and that an error of the corresponding class is actually contained in the image. The former option could be said about the pixel and primitive classes. Optimizations for these classes are presented and set for future work. The latter option is mainly about the precision class. Precision errors occur in most of the given images. It is difficult to spot these types of errors with the human perception, therefore some of the positive classifications may be left out in the ground truth for this class.

As the ground truth was manually created, there were as previously mentioned often times where at first a perceived false positive classification actually proved to be a true positive. An example of this was an image giving a perceived false positive from the pixel classifier, which proved to be a true positive after an analysis, as the image actually contained an unnoticed pixel error. If one would construct a more accurate ground truth, which contains all viable classes for an output image instead of one (the most noticeable) correct class per image, one would improve the classification results, such as the precision metric.

The recall metric for table 5.3 has a relatively low value for the tile and lighting classes. This is mainly because not all subclasses are implemented and taken into account, therefore the images with those specific subclass errors are not identified nor classified. The specific subclasses are summarized in section 6.2. For table 5.4, the recall metric gives a more realistic value of how well the algorithms cover the classes.

Comparing the accuracy metric between the reference and test set one will notice that they are quite similar for almost all classes. This is what we want since then we can assure us that the classifiers are good enough as we have predicted with the reference set.

As PSNR alone does not provide any information about the artefacts, one can not use the method to classify the artefacts nor compare its results with our solution's. Though, one could compare our result with the result when performing visual perception. Our ground truth, that the solution should follow, is mostly based on our and the CV team's visual perception. As mentioned above in this section, human perception is almost certain to not notice the smallest of changes in an image. For example, if there existed a change of intensity by one for a large number of pixels or one single faulty pixel with a large intensity change, visual perception would most likely not see the difference. It would, either way, take a lot of time to perceive it, or just make an adequate guess that it is a certain type of class. These types of artefacts will be detected by our solution, and it is here our solution provides the most benefit.

Our solution will not only act like a detector when human perception is not enough. It will guide the analyst to choose which images are suitable when searching for a specific artefact or to inform the analyst with class-specific data. This is relevant as the set of data received when running system tests are often of a large magnitude. In this way, the proposed solution will save expensive engineer work hours.

At Arm, when running a benchmark, one may receive many errors in the early development of a new product. In such cases the currently used method, PSNR, could give similar values for multiple of the faulty images. As one seldom has the time to manually classify all images if they are numerous, random checks on a couple are done. With this tool however, one could in such a case receive an estimate on whether all errors produced are in fact of similar nature, which otherwise may be taken for granted if one only performed a random check. As this might reveal other anomalies in the perceived homogeneous errors, the tool may also serve to improve the quality of verification and not only save time.

Not only the classification results are of great importance. The classification results measure the validity of the developed algorithms, but not their quality in terms of execution time. Analyzing the execution time of the algorithms is a way of measuring the processing power needed. These values can be found in tables 5.1 and 5.2. The execution time informs where there are possible "bottlenecks" in the code, meaning parts in the code that computes for a significant time compared to other parts. These bottlenecks have a large impact on the performance. The obvious bottlenecks in the code are found in the geometry and the primitive class since their mean and standard deviation time are significantly longer than the rest of the classes. This mainly depends on the segment-wise iteration approach, that is implemented in both of the classes, instead of going through the whole image. An image could have for example hundreds of segments that need to be analyzed independently and therefore increase the time complexity significantly.

If one was to compare the time taken for the classifiers to execute to how long it would take to manually classify the images, one should keep the arguments above in mind; that the tool is able to improve the quality of the verification and to save engineer work hours. The procedure of the manual classification evaluates in general one to a couple of random images from a set of images, that PSNR has detected with a faulty artefact. The procedure currently used is mostly only looking at output images, since the reference is often known before hand, due to experience. The exception is when the errors are more subtle. Then so-called hint images are needed. A hint image is an output image overlaid with a layer enhancing where there are differences in the image pair. It functions like our BDI in this paper overlaid on top of the output image. In such a case, a manual classification will take more time. This was taken into account when estimating the mean times taken for each classification performed manually, visible in table 5.1 and 5.2 as the last row, where longer times indicate more cases where an analysis of the hint image was needed and/or longer duration of the respective analysis.

6.2 Considerations regarding the classes

Due to the limited time scope, not all subclasses were implemented in this paper. The left out subclasses are considered to not be as important as the implemented subclasses and are expected to be unnecessary time-consuming or complex. There is still a possibility to develop these subclasses, but they will only be discussed in this paper. These subclasses are deformed geometry, unexpected angular appearance and spotted lighting.

- Deformed geometry error, see the characteristics of the geometry class in section 3.3.2. This error is a mixture of the appearing and missing geometry subclasses. To correctly implement this subclass one has to be able to identify that an appearing or missing edge belongs to a deformed geometry. The solution is not intuitive, but it might be done by comparing color maps of the output and the reference image to find matching colors of found geometries, and therefore map a geometry in the reference image to a deformed geometry in the output image.

- Unexpected angular appearance tile error, see the characteristics of the tile class in section 3.3.2. An idea to identify this error is to find corners, like the other tile subclass, but without the grid. The problem is that these errors often appear with particle and light effects. This erases the corners of the error, because particles overlap each other, and makes the error more precision related than tile related.

- Spotted lighting error, see the characteristics of the lighting class in section 3.3.2. To separate this error from the geometry or precision errors seems very hard. A possible solution might be to convert images from RGB to a YUV representation to analyze the luminance level of the output image compared to the reference image's luminance level. This will probably highlight light spots with errors.

It is worth to mention that some subclasses are harder to classify than others from a computer's point of view, as the subclasses characteristics are very similar to other class characteristics. This results in false positives that may prove difficult to correct. It is often edge cases of classes that give these problems, but there are subclasses that are hard to separate even when manually classifying them. An example of this is the sometimes vague distinction between the primitive and the geometry classes. A primitive error can be seen as a specific edge case for the geometry class as it manipulates parts of a geometry. One edge case where it is hard to differentiate the two is when a geometry error is cut off in some manner so that is takes the shape of a primitive.

An example of subclasses, which are more difficult to differentiate than others, are the miscolored geometry and the constant layer precision subclasses. They are very similar, as they both identify errors that have a constant color change. The difference is that miscolored geometry relates to geometries and the other relates to errors that cover the most of the image. When miscolored geometries cover the majority of the image, these subclasses are very hard to tell apart.

Another situation when this tool may be used to automatically provide information between an output and a reference image is when ray tracing is used. At the time of writing ray tracing is very seldom used in real-time 3D graphics, due to the heavy computations needed for each frame. But recently a breakthrough has been made in this area, as made famous by the Unreal Engine 2018 Demo, see [22], allowing ray tracing to be more commonly used in real-time 3D graphics in the near future. Since most ray tracing methods do not provide bit exact images when repeatedly rendering the same frame, due to the random direction of bouncing rays, methods like PSNR in some cases provide a non-optimal amount of information, concerning whether it, deemed by the human eye, actually exists a difference between a given output and a reference image. In the case of the tool discussed in this paper, one will receive a precision classification for all images created with ray tracing, but one may still receive other information about the difference between an output and a reference image, especially from those classifiers who require higher differences in intensity in their thresholds.

6.3 Future work

There are multiple optimizations that could be done to shorten the time taken for the execution of the classifiers. For example, there are cases of the same calculations performed multiple times in different classifiers. This is due to the fact that the development of the classifiers was done in a sequential fashion, with the plan to optimize so that each equal calculation was only performed once a certain robustness in the accuracy was established. Due to the limited time scope, this was not carried out.

As mentioned in chapter 5, to improve the classification results and the performance metrics one has to mainly optimize the pixel and the primitive classifiers. For the primitive and geometry class, one could use the image's corresponding 3D mesh, if available, to easier spot the edges of the faulty geometry. At the moment, there is a somewhat crude and computationally heavy pattern recognition method implemented for the primitive class, used to identify straight lines and triangles in an image.

For the pixel class, one could instead search for where the RGB colors channels for faulty pixels are a combination of maximized and minimized values, For example (255, 255, 255), (255, 255, 0), (255, 0, 255) and so on. This was one of the initial development ideas for the class, but it was canceled.

The choice to not include machine learning in the solution is discussed in section 3.2. However for some classes, even though there is a limited amount of data, some machine learning techniques could be applied. For instance, if one were to train an algorithm on the segments of a faulty primitive or geometry image one could possibly improve the results of the classes' corresponding classifiers, as there are often more than one faulty segment in these classes and one would, therefore, receive a much larger reference set. Another aspect to apply machine learning could be to generate data sets by injecting errors in the GPU pipelines or programs, instead of creating a test set. This would be more realistic and non-biased, but since we do not have access to the GPU in this thesis, due to the black box fashion the solution was developed in, this could not be carried out. A machine learning solution would need to be able to classify multiple classes at once, to be comparative to our approach. This could possibly discover other classes that we have not thought about.

Another approach to incorporate machine learning with our solution could be to create "meta classes" by using the classifications produced as features. For instance when the classification for an image pair is carried out one could produce a feature vector containing ones and zeros at positions related to classes and sub-classes, indicating if they received a positive classification result or not. The mentioned extensions of machine learning to this thesis are left as future work.

Chapter 7
Summary and conclusions

The implemented algorithms for detecting graphical artefacts proved to be overall quite accurate. As mentioned in the discussion, we received 82% and 78% mean accuracy for the seven classes for the reference and test set respectively.

Given a relatively small number of images with errors and their corresponding reference images, a solution using Matlab was implemented. With image analysis methods as a basis, the characteristics of the different determined classes were identified and analyzed independently of each other.

The classification result was as expected. The number of false negatives was relatively low and the number of false positives was somewhat high for some specific classifiers. The latter was mainly due to the constraints of these classes being too loose.

The goals and questions asked at the beginning of this master thesis are considered to be fulfilled and answered. The implemented algorithms are able to detect and classify different graphical artefacts with a fairly good result, which has been quantified by our chosen performance metrics. An analysis of performance is discussed, which includes these metrics and a comparison between relevant time models of our classifiers and manual classification. This analysis is presented together with thoughts about difficult parts of the classifications, ideas that did not get implemented and future work with machine learning.

Bibliography

[1] S. Joseph, H. Ujir, and I. Hipiny, "Unsupervised classification of intrusive igneous rock thin section images using edge detection and colour analysis", Faculty of Computer Science and Information Technology, University Malaysia Sarawak, 2017. [Online]. Available: `http://arxiv.org/abs/1710.00189`.

[2] J.-C. Yoo and C. Ahn, "Image matching using peak signal-to-noise ratio-based occlusion detection", IET Image Processing, vol. 6, 5 2012.

[3] W. Burger and M. Burge, *Principles of Digital Image Processing*, ser. Undergraduate Topics in Computer Science. Springer London, 2009.

[4] J. Korhonen and J. You, "Peak signal-to-noise ratio revisited: Is simple beautiful?", *2012 Fourth International Workshop on Quality of Multimedia Experience Quality of Multimedia Experience (QoMEX)*, 2012.

[5] Z. Wang and A. Bovik, "Mean squared error: Love it or leave it? a new look at signal fidelity measures", IEEE signal processing magazine, vol. 26, 1 2009.

[6] J. Canny, "A computational approach to edge detection", *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 8, 6 1986.

[7] W. Gao, X. Zhang, L. Yang, and H. Liu, "An improved sobel edge detection", *2010 3rd International Conference on Computer Science and Information Technology Computer Science and Information Technology (ICCSIT)*, 2010.

[8] S. Gupta and S. Mazumdar, "Sobel edge detection algorithm", *International Journal of Computer Science and Management Research*, vol. 2, 2 2013.

[9] P. Heckbert, *Survey of texture mapping*, 1986.

[10] T. Akenine-Möller, *Mobile Graphics hardware*. Lund Univerity, 2007, [Online; accessed 26-March-2018]. [Online]. Available: `http://fileadmin.cs.lth.se/cs/Education/EDAN35/mGH.pdf`.

[11] P. Soille, "Morphological image analysis: Principles and applications", Sensor Review, vol. 20, 3 2000.

[12] A. Challa, S. Danda, B. D. Sagar, and L. Najman, “Some properties of interpolations using mathematical morphology”, IEEE Transactions on image processing, vol. 27, 4 2018.

[13] L. Lam, S.-W. Lee, and C. Suen, “Thinning methodologies-a comprehensive survey”, IEEE Transactions on Pattern Analysis and Machine Intelligence, vol. 14, 9 1992.

[14] M. Kerschnitzki, P. Kollmannsberger, M. Burghammer, and G. D. et. al., “Architecture of the osteocyte network correlates with bone material quality”, The Official Journal Of The American Society For Bone And Mineral Research, vol. 28, 8 2013.

[15] P. Hough, “Method and means for recognizing complex patterns”, pat., 1962.

[16] R. Duda, P. Hart, and W. Newman, “Use of the hough transformation to detect lines and curves in pictures”, Communications of the ACM, vol. 15, 1 1972.

[17] D. Powers, “Evaluation: From precision, recall and f-factor to roc, informedness, markedness and correlation”, School of Informatics and Engineering, Flinders Univeristy, 2007.

[18] D. Olson and D. Delen, *Advanced Data Mining Techniques*. Springer, 2008.

[19] S. Loussaief and A. Abdelkrim, “Machine learning framework for image classification”, 2017 International Conference on Information and Digital Technologies, 2017.

[20] I. The MathWorks, *Matlab*, [Online; accessed 20-February-2018], 2018. [Online]. Available: `https://uk.mathworks.com/`.

[21] A. Carlman and D. Cheveyo, *Dropbox link to our directory with the used test images*, [Online; accessed 24-May-2018], 2018. [Online]. Available: `https://www.dropbox.com/sh/jtw7gas4nlu9px3/AACjK3lhmnICbdxNObiTpBtya?dl=0`.

[22] *Epic games demonstrates real-time ray tracing in unreal engine 4 with ilmxlab and nvidia*, [Online; accessed 16-May-2018], 2018. [Online]. Available: `https://www.unrealengine.com/en-US/blog/epic-games-demonstrates-real-time-ray-tracing-in-unreal-engine-4-with-ilmxlab-and-nvidia`.

Appendices

Appendix A

Histograms and other plots

In this appendix, histograms and interesting graphs that we have used to determine important values and thresholds for different algorithms are displayed. A histogram is constructed by storing the frequency of a factor we want to examine closely in an image. The factor could be for example a ratio between present pixels in an image, that is segmented in different ways. The factor could also be intensity values for pixels in an image. These histograms are mainly used to find correspondences between images, that belong to the same class. They are also used to pinpoint threshold values for achieving the desired segmentation of binary images: α, β and γ, seen in figure 4.1.

The data acquired for the histograms, that have 'number of output images' as the y-axis, is stored image-wise. This is done by placing a value in a histogram for each output image.

For the figure A.1, the black bars correspond to the red color channel, the gray bars correspond to the green color channel and the white bars correspond to the blue color channel.

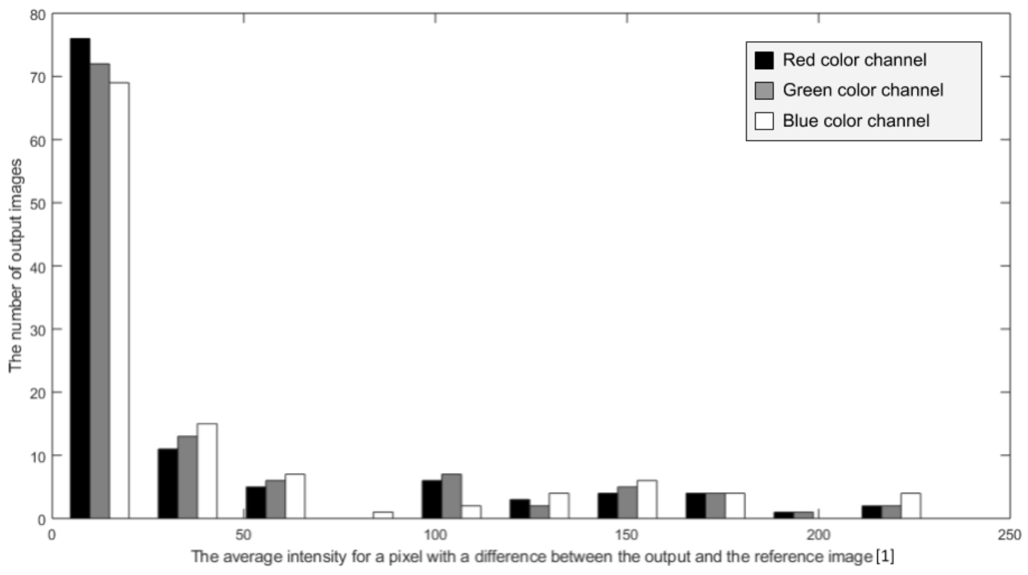


Figure A.1: Histogram for finding correspondences for classifying pixel and precision errors. The data displayed is the whole reference set. Note that 76% of the images have a value less than 30 in pixel intensity when measuring the average difference in intensity between an output and a reference image. The value is selected as the lower threshold for γ and upper threshold for β and α.

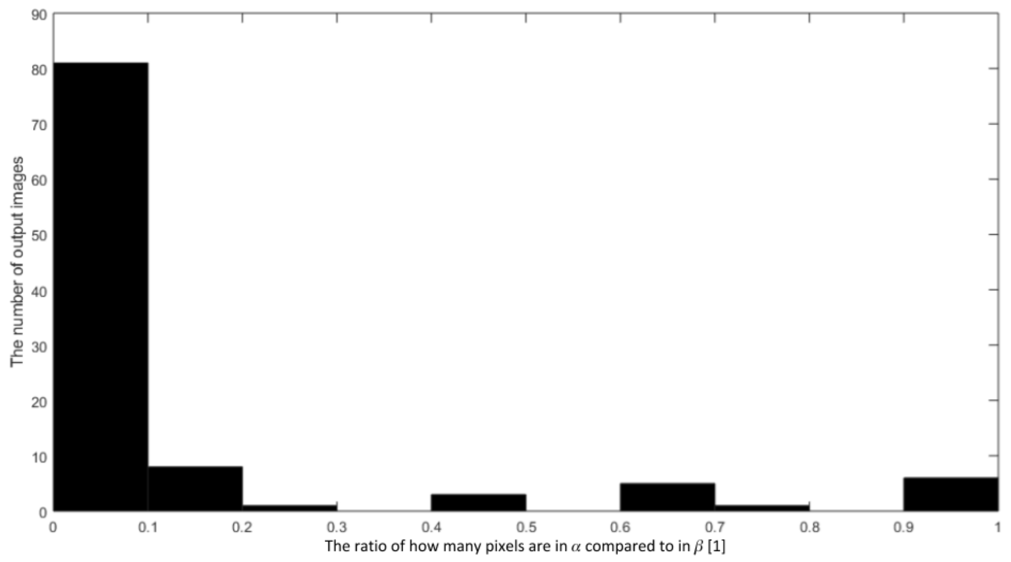


Figure A.2: Histogram for finding correspondences for classifying precision errors. The data displayed is the whole reference set. Note that 71% of the images have a value less than 0.1 in ratio. The ratio has an interval from zero to one, of how many pixels are in β compared to α. Zero means that there are only pixels present in β and one means that there are only pixels present in α.

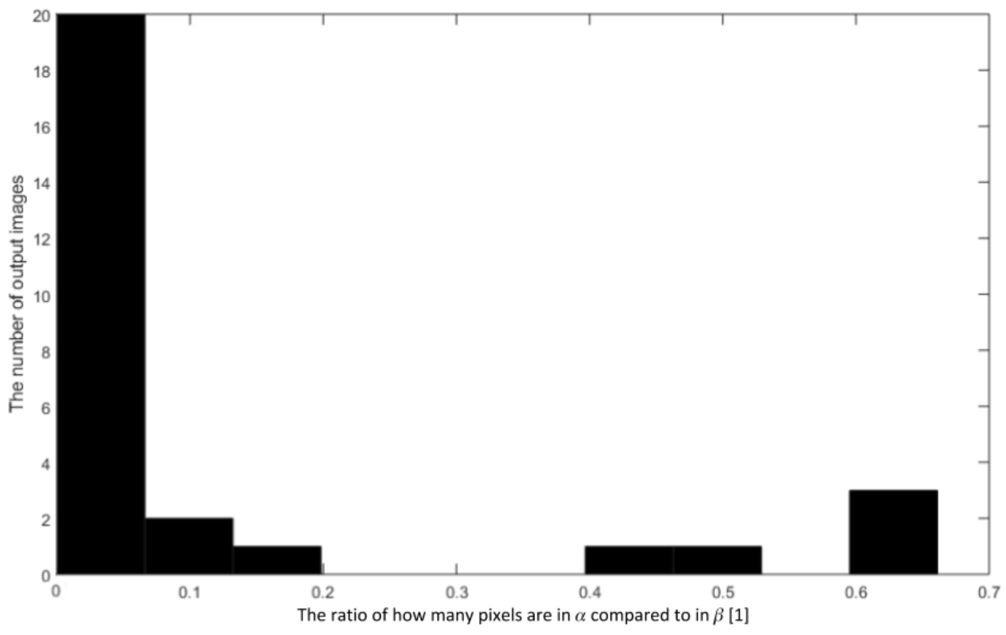


Figure A.3: Histogram for finding correspondences for classifying precision errors. The data displayed is only the images with a known precision error. By comparing with figure A.2, note that 28% of the images, that have a greater value than 0.1 in ratio, are precision errors. The ratio has an interval from zero to one, of how many pixels are in β compared to α. Zero means that there are only pixels present in β and one means that there are only pixels present in α.

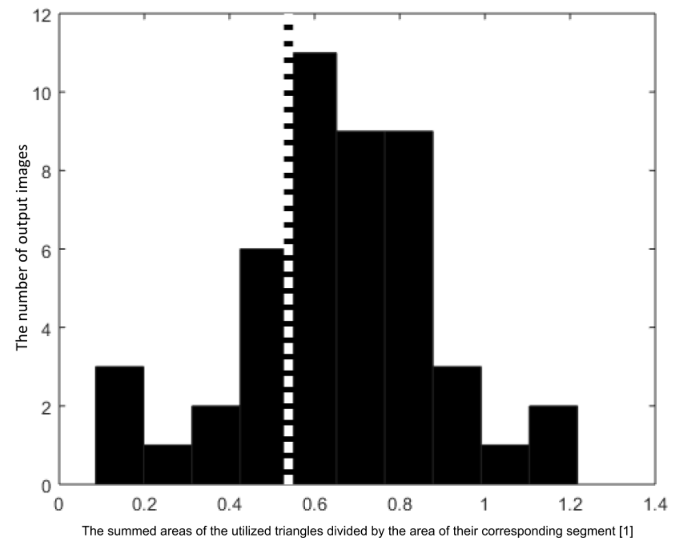


Figure A.4: This plot was used to set a threshold to restrict the primitive classifier, marked with a thick white and black striped line. Note that the segments represented in this histogram are those which first gave a positive primitive classification or the segment in the image which provided the highest ratio value, if the image was not classified as a primitive error.

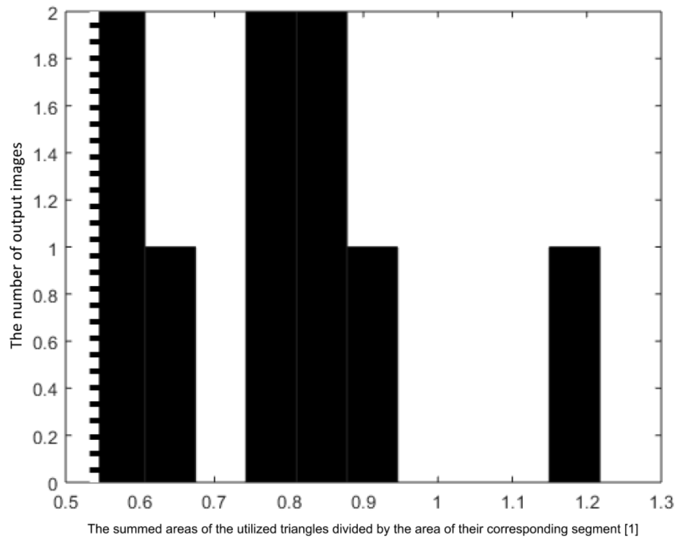


Figure A.5: This histogram was used to set a threshold to restrict the primitive classifier, marked with a thick white and black striped line. Note that the segments represented in this histogram are those, in images belonging to the primitive ground truth, which first gave a positive primitive classification or the segment in the image which provided the highest ratio value, if the image was not classified as a primitive error.

EXAMENSARBETE Automated classification of graphical artefacts in 3D graphics verification
STUDENT Daniel Cheveyo, Arvid Carlman
HANDLEDARE Flavius Gruian (LTH), Rasmus Persson (ARM)
EXAMINATOR Michael Doggett (LTH)

Automatiserad klassifikation av grafiska artefakter inom 3D-grafik

POPULÄRVETENSKAPLIG SAMMANFATTNING **Daniel Cheveyo, Arvid Carlman**

AUTOMATISERING AV TIDIGARE MANUELLT UTFÖRDA UPPGIFTER ÄR IDAG ETT POPULÄRT ÄMNE INOM UTVECKLINGSPROCESSER HOS DE FLESTA FÖRETAG. I DENNA ANDA HAR VI UTVECKLAT ETT VERKTYG FÖR KLASSIFICERING AV FEL I BILDER RENDERADE VIA 3D-GRAFIK.

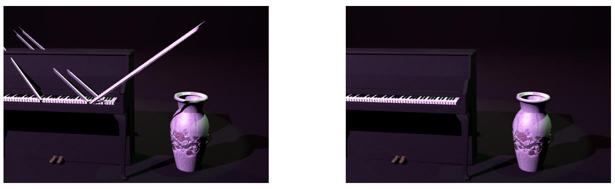
System testning används när man utvecklar hårdvara och drivrutiner för rendering av 3D-grafik. En viktig aspekt när man utför system testning är att jämföra en resulterande bild av en specifik 3D-scen mot dess korresponderande referensbild. Huvudverktyget som används för denna jämförelse är Peak Signal-to-Noise Ratio (PSNR). PSNR kan bara användas för att ge ett värde på skillnaden mellan bilderna, men inte information som exempelvis vilken typ av grafisk artefakt, vilket innebär ett fel i bilden från renderingen av dess 3D-scen, som den felaktiga bilden innehåller.

I vårt examensarbete har vi implementerat och skrivit algoritmer som kan klassificera grafiska artefakter med bildanalytiska metoder för förutbestämda klasser. Eftersom artefakterna som dyker upp är så olika och att vi var givna en liten mängd av endast 112 bilder för utveckling av algoritmerna, så valde vi att inte använda oss av de populära maskininlärningsmetoderna. Istället skapades egna skräddarsydda algoritmer för respektive artefakt, även kallad klass. Detta gav en nisch till verktyget i och med att det är användbart även om man inte har tillräckligt med relevant data till hands.

Vi implementerade totalt 7 klasser med varierande egenskaper och utseenden. Det svåra med att hitta dessa utmärkande egenskaper var när bilder med artefakter från olika klasser hade många detaljer gemensamt. Algoritmerna utvecklades genom att gränser och värden för olika parametrar i bilderna valdes ut, funna genom analys av den givna datan, som sedan testades för bilder med liknande fel som vi själva genererade, för att säkerställa att implementeringen var korrekt. En exempelbild på hur en artefakt kan se ut är illustrerad i figur 1a, brevid bildens referensbild, det vill säga hur den skulle ha sett ut i figur 1b.

Med detta verktyg kan man spara många dyra ingenjörstimmmar och finna fel som inte kan ses med blotta ögat.

Resultatet visar att 82% av alla bilder som har använts för utvecklingen av algoritmerna klassificerades rätt, och 78% av alla testbilder blev korrekt klassificerade.


(a) (b)

Figur 1