Using Image Quality Assessment to Test Rendering Algorithms

Julian Amann Bastian Weber Charles A. Wüthrich
CogVis/MMC, Faculty of Media
Bauhaus-University Weimar
Bauhausstrasse 11
99423 Weimar, Germany
{julian.amann|bastian.weber|charles.wuethrich}@uni-weimar.de

ABSTRACT

Testing rendering algorithms is time intensive. New renderings have to be compared to reference renderings whenever a change is introduced into the render system. To speed up the test process, unit testing can be applied. However, detecting differences at the pixel level does not provide a sufficient criterion for the tests. For instance, in the context of games or scientific visualization, we are often faced with random procedurally generated geometry like e.g. particle systems, waving water, plants or molecules. Therefore, a more sophisticated approach than a pixelwise comparison is needed. We propose a Smart Image Quality Assessment algorithm (SIQA) based on a self-organizing map which can handle random scene elements. We compare our method with traditional image quality assessment methods like Mean Squared Error (MSE), Peak Signal-to-Noise Ratio (PSNR) and Structural Similarity Index Maps (SSIM). The proposed method helps to prevent the detection of images being categorized wrongly as correct or having errors, and ultimately helps saving time and increases productivity in the context of a test-driven development process for rendering algorithms.

Keywords

image quality, unit testing, computer graphics, self-organizing map

1 INTRODUCTION

During the last three decades, rendering systems have grown immensely in complexity: real time rendering subsystems of 3D gaming engines such as the Unreal Engine can contain millions of lines of code, while offline rendering systems such as RenderMan or Autodesk 3ds Max have been in development for almost thirty years [AGB99]. The increase in complexity of rendering systems makes it a challenging task to ensure at the same time a high software quality.

To handle the increasing software complexity and to assure the quality of such a rendering system, the system must be tested. Testing can be done in different ways. For instance, a quality assurance team can check different outputs of the rendering system and compare them to old output. For example, a scene can be loaded and rendered with the same camera settings and same rendering settings twice: once with an early revision of the software, and a second time with the new version. If the images generated differ, a software bug or a conceptual error might have sneaked into the rendering system. This manual work is very labor intensive and a tedious task for a tester, especially when many different test scenes have to be checked with all the different variants of render settings.

To improve this situation, unit testing can be used instead of manual testing. Unit testing allows to test the system automatically whenever new code has been introduced into the codebase. Also, a unit test can be executed for instance by a nightly build server to supply the developers daily with a current quality assessment report.

This paper is concerned with the automatization of this testing process, outlines how such an automatic process can be implemented, develops a method for measuring the quality of rendered images and traces the path for managing random output tests for a rendering system.

The novel contribution of this article is a smart image quality assessment algorithm (short SIQA) which can evaluate the quality of an image based on similar reference images. This makes the algorithm unique with respect to existing reference based image quality assessment methods like Mean Square Error measures, Peak

Signal-to-Noise Ratio (PSNR) or Structural Similarity Index Maps (SSIM) [WBS03].

Our proposed algorithm (SIQA) can be used for black box testing and does not need any knowledge of the internal structure of the render system. This gives the algorithm the typical disadvantages of black box tests, such as being broad based, but also their advantages of being very general and flexible to software changes.

2 MOTIVATION

Software bugs can easily be introduced in a render system when changing its code. Often, bugs are not detected immediately. Sometimes, a bug in a subcomponent of a rendering system can affect other parts of the rendering system. A typical example of such cascading error behaviour can be seen for instance in the discontinuities in the shadows of Figure 1.

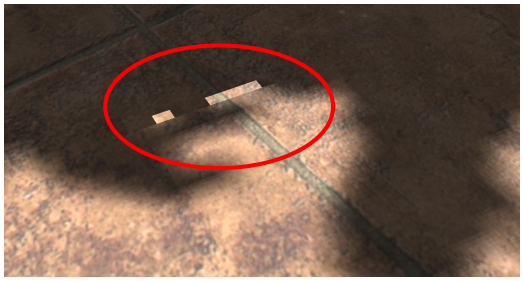


Figure 1: A discontinuity in shadow due a software bug.

The growing complexity of rendering systems called for changes in the software engineering process: to ensure a high quality of the images, bugs and errors must be detected early and in a reliable way.

One approach to detect render system bugs early is Test-Driven Devolpment (TDD) [Bec02]. TDD has become quite popular in software development. In TDD tests (unit tests) for a software component are developed before new features are implemented. After the implementation of a new software component, all existing tests are executed to see if one or more of them fail. Quite often, testing fails and early bug tracking will help to produce a better implementation, which eventually will pass all tests defined at the beginning. In the next step of TDD, refactoring is used to eliminate for example redundancies in the source code. This can again introduce new bugs which can be detected early by unit tests.

A test process for a rendering system consists of three steps: at first, the code is modified. Subsequently tests are run. Finally, the test results are analyzed (Figure 2). If a tests fails, the process is restarted. The code is modified again and eventually it will pass all tests. Even without using a test driven development process, testing makes sense.

Using TDD implies a paradigm change in software production: before the introduction of TDD, small ad hoc

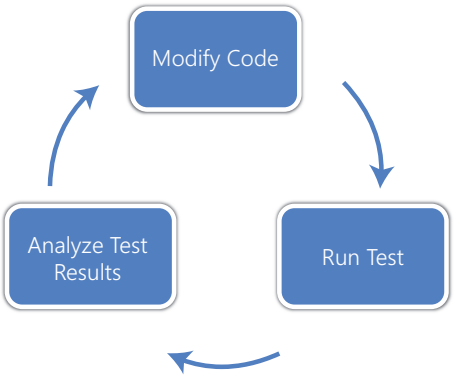


Figure 2: The test procedure

test applications were made to test the software, but they were discarded after the testing. In TDD, unit tests are as important as production code [Rob08], and code quality of the test code becomes as important as the software itself and gets the same attention and care as the actual software. Testing makes sense even when not using a test driven development process.

2.1 An automatic test process

The TDD steps illustrated in Figure 2 can be easily automated with the help of a test server (e.g. a nightly build server). Developers check in code to a repository. The code contains unit tests that can be executed automatically by a unit test system. The build server automatically checks out the repository on a dialy basis and runs all unit test. Finally, it reports the test results to the developers through a website. This test approach can be implemented for instance with CDash in combination with CTest, CMake, Visual Studio and Mercurial as a version control system [Kit13]. An overview of this test process configuration can be seen in Figure 3.

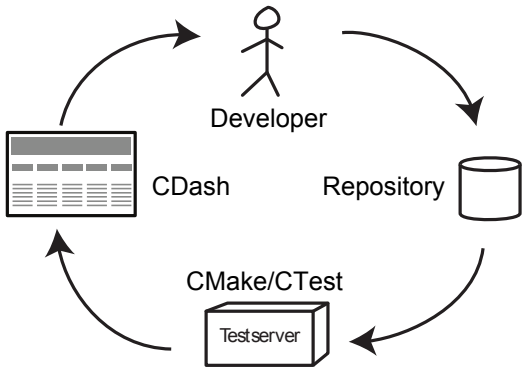


Figure 3: An automated test process with CMake, CTest and CDash

The following pseudo code demonstrates how a unit test is usually implemented:

```
TEST(Core, myAddition)
{
    int result = Bar::myAddition(3,4);
```

```
    EXPECT_TRUE(result == 7);
}
```

The method `myAddition` adds two numbers and returns the computed result. The assertion `EXPECT_TRUE(result == 7)` validates the computed result. It checks if the addition of the numbers 3 and 4 results in the sum of 7. If so, the unit test passes without failures, if not, the unit test framework reports an error.

Because images are difficult to compare, in the context of a rendering system, it is not easy to focus on how to test or what to test on. A trivial solution to this problem could be to generate a reference image, render the scene with the updated software, and compare the reference and the generated image.

For each unit test, a correct reference image (valid screenshot) has to be created. For example, we can create a unit test which checks if tessellation is working. Another test can check if pixel shaders are working or if volume rendering works. Figure 4 shows some examples of such reference images.

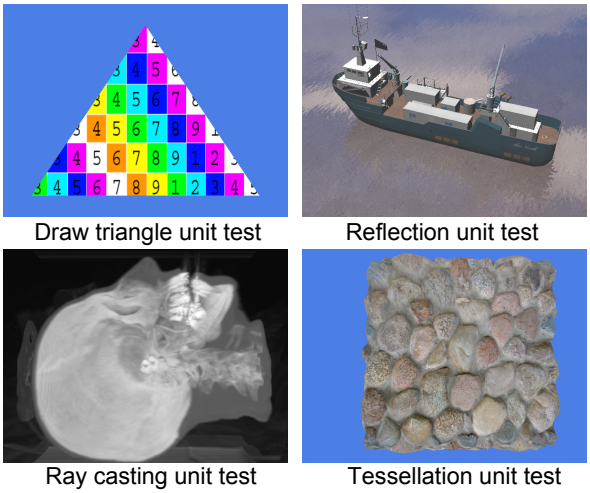
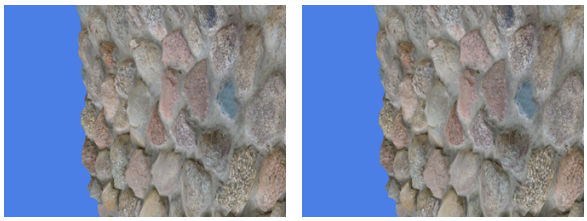


Figure 4: Each picture is a reference image for a specific unit test.

Comparison between the rendered image, the correctness of which is not known, and the reference (a valid screenshot from the past) is done pixel by pixel. If one pixel differs, the test fails. This test would work in case different renderings produce exactly the same output. Unfortunately, this is almost never the case: even the same setting, rendered through two different graphic cards, do not produce the same output. Figure 5 shows renderings of the same scene on a GeForce GTX 480 and on a GeForce GTX 560 Ti: the output is different, but it looks the same at a first glance.

Figure 6 highlights in red different pixels in the rendered output through a GeForce GTX 560 Ti and a GeForce GTX 480. Remarkable here is that both times



(a) GeForce GTX 560 Ti (b) GeForce GTX 480

Figure 5: The same unit test rendered on two different GPUs.

the same program with the same Direct3D 11 render commands and the same driver (GEFORCE R300 DRIVER, 301.42 WHQL) has been executed. This can be due to non-deterministic behavior or implementation defined behavior. The DirectX 11 specification allows some freedom in the concrete driver implementation which can result in such differences. A reason for the different render output can be for instance differences in the texture sampling pipeline across different GPUs.

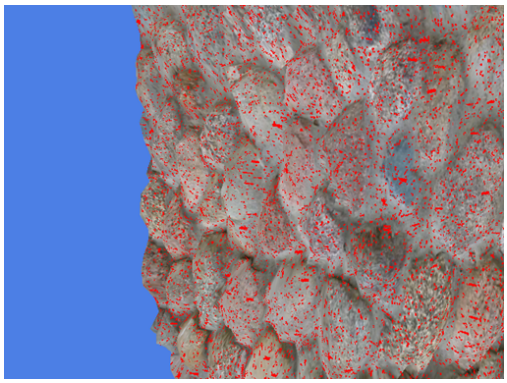
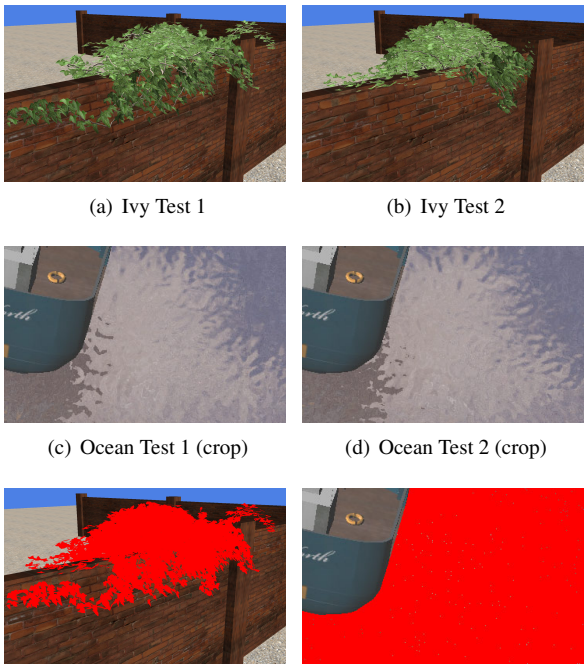


Figure 6: Red pixels mark differences between the render output of a GeForce GTX 560 Ti and a GeForce GTX 480.

A more sophisticated solution to prevent false positives in the classification has to be found. A pixelwise comparison does not work here. Furthermore, a distinction between unintentional randomness in the render output as seen above in the tessellation example and intentional randomness in a scene is necessary. Figure 7 shows two examples of intentional randomness in a scene. The first one is an ivy generator which generates a new ivy structure each time executed. The other example shows waving water. The differences are highlighted in red (Fig. 7(e) and 7(f)). Although it is possible to record the random parameters to regenerate the same scene twice, it can be sometimes quite cumbersome. For example, some data might be only GPU accessible and can be difficult to access from the CPU side. In this case, our technique can help. It is just easier, because it works without knowledge of the internal code structure as a black box test.



(a) Ivy Test 1 (b) Ivy Test 2

(c) Ocean Test 1 (crop) (d) Ocean Test 2 (crop)

(e) Highlighted Differences in Ivy Test (f) Highlighted Differences in Ocean Test (crop)

Figure 7: Two examples of intentional randomness

3 RELATED WORK

Quality assessment methods for images can be subdivided into subjective and objective approaches. In subjective assessment, human observers are asked to evaluate the quality of an image, and the results are evaluated statistically. Such methods are slow and expensive, which makes them not well-suited for their use in software development.

Objective methods instead try to determine the quality of an image automatically through some measure. They can be further subdivided into *full-reference*, *no-reference* and *reduced-reference* approaches [WB06], depending on the availability of a reference image, which is assumed to be error free and of highest quality. In reduced reference methods, features are extracted from the reference image and the method only compares this limited set of features with the image the quality of which has to be evaluated, which we will call *subject* image.

One of the most used full-reference methods is the Mean Square Error (MSE), which is the square of the L^2 distance of the images, and computes the square of the difference of corresponding pixels in the reference image A and the subject image B. Let $A = \{a_1, a_2, ..., a_N\}$ and $B = \{b_1, b_2, ..., b_N\}$ be the images, then MSE is calculated by:

$$MSE(A,B) = \frac{1}{N}\sum_{i=1}^{N}(a_i - b_i)^2. \qquad (1)$$

MSE provides an overall measure of how the intensities of two images differ from each other [WB06].

Another commonly used measure is Peak Signal-to-Noise ratio (PSNR), which is based on the L^P norm and is defined as follows:

$$PSNR(A,B) = 10log_{10}\frac{I^2}{MSE(A,B)}, \qquad (2)$$

where I is the maximum allowed pixel intensity (i.e. 255 for an 8-bit grayscale image).

PSNR is also often used to evaluate the quality of an image. A PSNR value is typically in the range of 30 dB to 40 dB. A higher PSNR value is better than a lower PSNR value, since a higher value indicates that the signal is closer to the original signal than a signal with a lower PSNR value [WB06].

MSE and PSNR work only in a full-reference setting and consider an image only on a pixel-by-pixel basis.

A third type of quality measure commonly used in full reference approaches is structural similarity. In contrast to MSE and PSNR, structural similarity does not only consider individual pixels, but also the neighborhood of a pixel. For instance, a structural similarity index can be computed by cropping two small image regions from the reference and subject images and combining the neighborhood by a weighting function. A simple structural similarity index can be computed by considering the luminance and the contrast of an image.

The Structural Similarity Index Map (SSIM) of two image regions $\mathbf{x}$ and $\mathbf{y}$ can be computed by

$$SSIM(\mathbf{x},\mathbf{y}) = \frac{(2\mu_x\mu_y + C_1)(2\sigma_{xy} + C_2)}{(\mu_x^2 + \mu_y^2 + C_1)(\sigma_x^2 + \sigma_y^2 + C_2)}, \qquad (3)$$

where C_1 and C_2 are constants which prevent a division by zero, μ_x and μ_y are sample means of $\mathbf{x}$ and $\mathbf{y}$ respectively and σ_x^2, σ_y^2 are the sample variances of $\mathbf{x}$ and $\mathbf{y}$ [LB09] . Usually, the mean SSIM value is used to evaluate the quality of an image.

In a reduced-reference setting, for example keypoint matching can be used. The goal is to find a certain number of interesting points in the reference image and match these in a similar subject image. For instance, the Scale-Invariant Feature Transform (SIFT) can be used here [Low99]. But it is not yet clear how this technique can be used to assess the quality of an image.

[Rob12] introduced a no-reference algorithm for detecting global-illumination based rendering artifacts via a machine learning approach supposedly competitive with state of the art full-reference algorithms. This algorithm uses "right" and "wrong" image samples. After manual masking of the problematic areas, a classifier is trained with features extracted from these areas.

Starting with a pool of features, the most discriminative ones concerning this artifact in consideration are selected. After the detection, a correction method is used, removing the artifacts from the image.

4 SELF ORGANIZING MAPS TO EVALUATE QUALITY

4.1 Overview

Our method is based on a self-organizing map (SOM) which allows us to classify images as correct or wrong. The idea of a SOM, also known as Kohonen map has been introduced in [KT82]. From a high level point of view, a SOM maps feature vectors of an arbitrary n dimensional space to a 2D space. The SOM has the characteristic that feature vectors with similar properties form clusters on the 2D projection of the corresponding SOM.

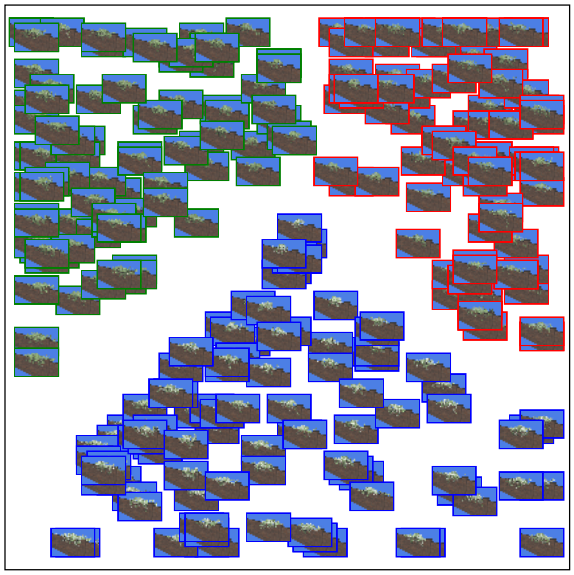


Figure 8: Projection map of classified items on a SOM

Figure 8 shows this effect: as can be seen in the picture, different types of items concentrate to different map regions. Images with a green border are correct. Images with a blue border have incorrect alpha blending issues. A red border marks images with a wrong depth stencil test.

The complicated part of the design of the method for evaluating mistakes in rendering is to find good features that help to distinguish between correct and incorrect images.

Self-organizing maps support a non-supervised learning process. However, to get interesting results one has to pre-classify at least one item. Otherwise, the SOM can only tell us that an image belongs to a certain cluster, for instance to the cluster of the images marked with a red border in image 8. However, it cannot tell us if the corresponding cluster is a cluster which contains only correct or only incorrect images. Ideally, we pre-classify a set of correct and incorrect images beforehand. A person from the quality assessment team has then to decide if the image is correct or not. The SIQA system stores this information and uses it to make an independent decision for future test images. Furthermore, the SIQA can also compute an estimate of how likely it is correct. This is done by determining the distance of the test image to the already correct classified images in the SOM.

4.1.1 Self-organizing map

Our implementation of a self-organizing map is represented by a square lattice. Each node in this lattice is associated with a vector whose dimension equals the dimension of the feature vector. These are called codebook vectors. At first, the SOM has to be initialized. We implemented random initialization and linear gradient initialization. The latter means that the codebook vectors start with 0 in the upper left corner and increase linearly until the right bottom corner where they reach 1. Afterwards, the training of the SOM may begin. The amount of training feature vectors has to be known to compute the decay of the learning rate and the neigbourhood radius. Additionally, they have to be passed as a set in order to be able to determine the minimum and maximum value of each feature which in turn are used to normalize the values so they are inbetween 0 and 1. This is done automatically during the training, so there is no need for the input to be normalized. The order in which the training vectors are used can in our case either be random or round-robin.

Another option would be to leave the ordering of the items as provided which might result in unpredictable behaviour.

Training then works as follows: the algorithm iterates through all the codebook vectors and calculates the Euclidean distance in feature space between the codebook vector and the training item vector. The item is then projected onto the node with the smallest distance, which is denoted *best matching unit*. Codebook vectors that lie inside the *neighbourhood radius* are modified in a way that they get closer to the feature vector of the projected item. Such radius decreases over time. The starting radius r_{start} is calculated as

$$r_{start} = \frac{w}{2}, \tag{4}$$

where w is the width of the square lattice. The current radius r dependent on the time t is computed as

$$r = decayFct(r_{start}, t, \tau), \tag{5}$$

while the time constant τ is obtained as

$$\tau = \frac{N}{ln(r_{start})}, \tag{6}$$

where N is the amount of training items. The decay function is defined as follows:

$$decayFct(a,b,c) = a \cdot e^{-\frac{b}{c}} \tag{7}$$

An additional variable called the learning rate (l) is used in the learning process. This variable is initialized with a user provided value l_{start} which is between 0 and 1. The learning rate controls how much impact the learning process has on the codebook vectors. It decreases over time (t) using the same decay function

$$l = decayFct(l_{start}, t, \tau). \tag{8}$$

The impact of the learning procedure on the nodes inside the neighbourhood radius also depends on the distance to the best matching unit, and decay is defined through a Gaussian bell:

$$gaussFct(\delta, \sigma) = e^{-\frac{\delta^2}{2 \cdot \sigma^2}}. \tag{9}$$

The distance can be calculated as the Euclidean distance using the x,y-coordinates of the nodes. Now with all of the aforementioned parameters, the actual learning process may take place:

$$\psi = \psi + gaussFct(\delta_{bmu}, r) \cdot l \cdot (\omega - \psi). \tag{10}$$

Here ψ is the codebook vector that is getting trained and ω the feature vector used for training. The variable δ_{bmu} denotes the distance between the best matching unit and the node containing ψ. Concerning the time value t, it has to be noted that t is initialized with 0 before starting the training procedure and is increased by 1 after each training item. The order of the training items has a great influence on the quality of the learning. We experimented with random (shuffled) and round-robin order and could achieve the best result with linear gradient initialization in combination with round-robin order.

When the training process is finished, the SOM may be saved as a file, allowing the user to load it again for classification purposes whenever he needs to, saving the training (learning) step to the user.

The classification process is very simple: similarly to the training phase, the best matching unit for an item is chosen. Then, the average distance to all the training items is calculated per item group. The item group with the lowest distance wins and the classification item is classified as that type.

It is also possible to use another type of cluster analysis. [Ran71] describes some objective criteria for the evaluation of clustering methods that can also be considered here. Similar work has also been done in [BGV92].

4.2 Feature Vector

The self-organizing map approach requires feature vectors for training. A feature vector can be seen as an n-dimensional vector, where each component represents one type of feature. Per image one feature vector is created and each feature is described by a floating point value. These values have to be greater than zero. During the training process, they will automatically be normalized between 0 and 1. The purpose of the feature vector is to describe an image as distinct as possible. In short, a number of features is extracted from each image and pooled into a vector.

The task of the self-organizing map is to distinguish different types of images. Therefore, the features must be selected in a way that they are as similar as possible for images of the same kind, and differ as much as possible when images are different. The approach we followed was to design potentially effective features at first and to select those that were actually the most useful afterwards by evaluating them directly on the image set.

4.2.1 Choice of the features

At first, the features chosen were the ones represented in table 1.

For most calculations, only the brightness information of the image is being used. Only the *average hue* and the *color histogram* make use of the color information. Hue, saturation and brightness values were calculated according to the transformations of the HSV color model [Rei08]. Alternatively, also the L*a*b* model could suit the purpose of providing luminosity values. The first block contains values that are extracted from the power spectrum of a 2-dimensional Fourier transform of the image, as shown in Figure 9.

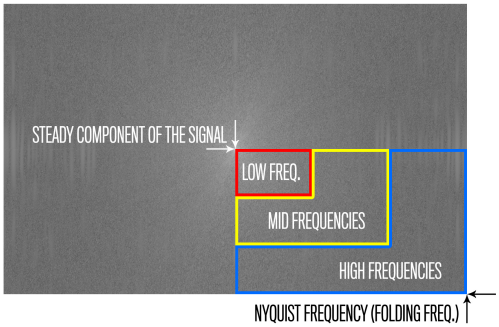


Figure 9: The extraction of the frequency measures

The values of the low, mid and high areas are getting averaged. These values give an impression on how much low, mid and high frequency detail can be found in the luminosity information, which is most important for human perception. The *Pixel-value-median* is a measure for the general brightness of the image. *Edginess* is calculated as the averaged sum of the differences from

Feature group	Feature values
Two-dimensional Fourier transform	Steady component Low frequencies Medium frequencies High frequencies
Medium luminosity	Median of the pixel values
Horizontal and vertical edginess	Summed absolute differences in horizontal direction Summed absolute differences in vertical direction
Average hue	Average of the HSV hue value
Darkest and brightest areas	Darkest pixel x-Coordinate median Darkest pixel y-Coordinate median Brightest pixel x-Coordinate median Brightest pixel y-Coordinate median Darkest pixel value Brightest pixel value Darkest pixel amount Brightest pixel amount
Average gradient	Average gradient x-direction Average gradient y-direction Average gradient length
Histogram	Red channel histogram Green channel histogram Blue channel histogram

Table 1: The features

one pixel to the next, once in the horizontal and once in the vertical direction. *Average hue* is obtained by averaging the hue information of all pixels, providing a value for the prevalent color in the image. Darkest and brightest pixels denote the pixels with the highest brightness value and the pixels with lowest brightness value in the image. There might be one or more pixels of that kind, the exact number is saved in the corresponding features *darkest pixel amount* and *brightest pixel amount*. The median of the x- and y-coordinates of these pixels is also being used as a feature, indicating the position where the brightest and darkest areas in the image are concentrated. *Darkest and brightest pixel value* denote the exact brightness values of the brightest and darkest pixel. This feature group gives a hint on how these very distinctive areas are distributed over the image. The *gradient* is a 2-dimensional vector that is based on the gradients in x- and in y-direction. These gradients are calculated by making use of the difference quotient

$$f'_{central}(x_i)' = \frac{f(x_{i+1}) - f(x_{i-1})}{2}. \tag{11}$$

The x- and y-gradients are combined into a vector. Then, all gradient vectors are averaged, while the direction is only taken into consideration, if the gradient length lies beyond a certain threshold. The direction, as well as the length of this average vector, is being used as a feature providing information of the edge distribution in the image. The *color histogram* is defined through the red, green and blue component histogram. Each of those is generated out of the red, green and blue channels of the image by calculating the relative occurence frequency of each color value, meaning the percentage of pixels of this color in the respective channel. The amount of histogram values is reduced by combining a certain amount of neighbouring values through averaging. This way, a histogram width of 32 values is reached. The color histogram can reveal changes in the color distribution.

4.2.2 *Evaluating Feature Effectiveness*

As mentioned above, an evaluation of the features was performed directly on the image set that should be worked on. The process of this evaluation was implemented in the following way: assuming we have m groups of images denoted as G_i with one feature vector v_j per image

$$G_i = \{v_1, v_2, v_3, ..., v_p\}, \tag{12}$$

and that these feature vectors all contain n features,

$$v_j = \begin{pmatrix} f_1 \\ f_2 \\ f_3 \\ \vdots \\ f_n \end{pmatrix}, \tag{13}$$

then the average α and variance σ of the feature k in group i is given by

$$\alpha(k, G_i) = \frac{1}{|G_i|} \sum_{j=1}^{p} v_j[k] \tag{14}$$

$$\sigma(k, G_i) = \frac{1}{|G_i|} \sum_{j=1}^{p} (v_j[k] - \alpha(k, G_i))^2. \tag{15}$$

Now the distance between two groups G_h and G_i concerning the feature k can be computed as

$$\delta(k, G_h, G_i) = |\alpha(k, G_h) - \alpha(k, G_i)|. \tag{16}$$

$\delta(k, G_h, G_i)$ indicates how well the feature k is suited for distinguishing items of the groups G_h and G_i.

A more complete approach for selecting the best features could be done through principal component analysis. However, this is beyond the scope of this paper.

5 RESULTS

There are tasks where conventional methods like MSE or SSI are completely sufficient for quality assessment. One is, as turns out, the evaluation of defective images as shown in Figure 5. The error is small enough to be able to discriminate correct looking images that only contain minor errors caused by differences in driver implementation (referred to as false negatives) from actually erroneous images (referred to as correct rejections) by defining a threshold value for MSE or SSIM. Table 2 shows this: for the false negatives, the SSI value is always very close to 1, while for the correct rejections the values are lower than 0.8. Hence, it would be possible to say that all images with an SSI value above 0.99 are supposed to be seen by an observer as being correct. The same conclusion can be deduced from the MSE values. They are all below 0.05 for the false negatives and above 3000 for the correct rejections.

False Negatives		Correct Rejections	
MSE	SSI	MSE	SSI
0.0490115	0.999995	18951.6	6.52E-14
0.000315755	1	3248.31	0.748823
0.031875	0.999983	10301.5	1.04E-07
0.00200065	1	10699.3	1.60E-07
0.000706706	1	10301.5	1.04E-07
0.000199653	1	19775.1	0.754946
<0.05	>>0.99	>3000	<0.8

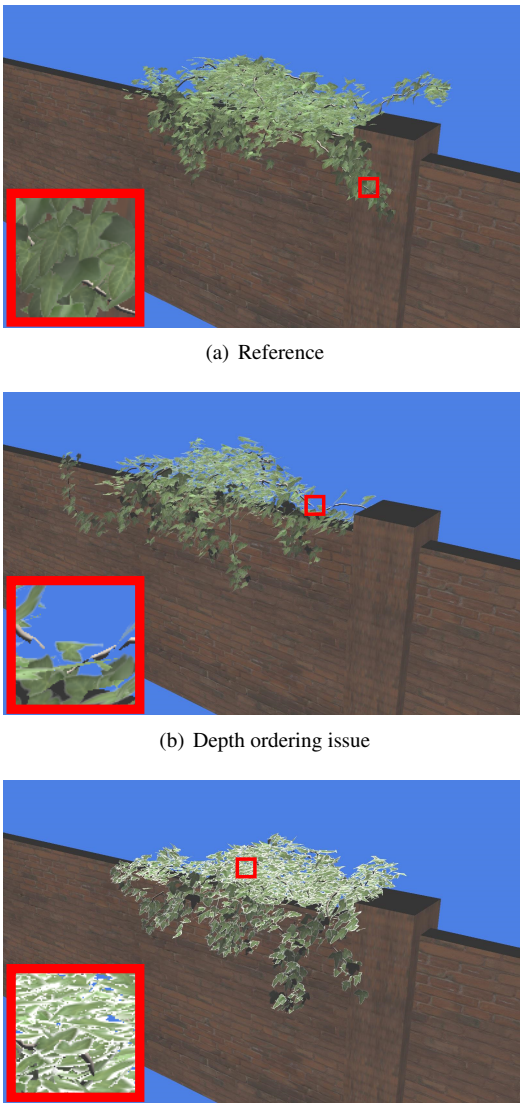
Table 2: MSE and SSI values for images with minor (false negatives) and major errors (correct rejections)

Figure 10 shows an example of randomly generated plants. While 10(a) shows a correctly rendered version, 10(b) (referred to as Error-1) and 10(c) (referred to as Error-2) show two kinds of rendering errors.

The images look different each time they are rendered, since the plants are based on a stochastic approach. Mean squared error (MSE) and Structural Similarity Index (SSI) measures are not suitable here due to the fact that there is no reference image that can be used for comparison. To prove this, we classified images through MSE and SSI. As footage, we used 300 of the ivy images, 100 for each type. We then compared each image with 30 images of each group and averaged the MSE respectively SSI value per group. The image was then classified as the type that had the lowest average MSE value or in case of SSI the highest value. The MSE results can be seen in table 3, and the SSI results are shown in table 4.

	Classified as		
	Correct	Error-1	Error-2
Correct	52	48	0
Error-1	12	88	0
Error-2	9	91	0

Table 3: Classification via MSE



(a) Reference

(b) Depth ordering issue

(c) Alpha blending issue

Figure 10: Generic plants

	Classified as		
	Correct	Error-1	Error-2
Correct	54	46	0
Error-1	49	51	0
Error-2	79	10	11

Table 4: Classification via SSI

As can be seen, results are not satisfying. In the case of MSE, no image at all was classified as Error-2 type. The rest is distributed randomly, only for Error-1 a small correlation in the case of MSE could be seen. SSI performs even worse, although it classifies 11 images as Error-2.

We then trained a self-organizing map with 100 images of each kind, using linear gradient initialization and round-robin order. Afterwards, we classified the same set with the trained SOM. Figure 8 shows the result of the classification. The red, blue and green frames rep-

resent the different image types. It turns out that they are concentrated in three different areas of the SOM and three distinct clusters can be observed. The classification success in numbers can be seen in table 5.

	Classified as		
	Correct	Error-1	Error-2
Correct	100	0	0
Error-1	0	100	0
Error-2	0	1	99

Table 5: The results of the SOM classification

The amount of false negatives, meaning images that are correct but have been classified as erroneous, in comparison between the different evaluation methods pixel-wise comparison (PWC), Mean Squared Error (MSE), Structural Similarity Index (SSI) and our Smart Image Quality Assessment Algorithm (SIQA), is shown in table 6.

	PWC	MSE	SSI	SIQA
False Negatives	100%	48%	46%	0%

Table 6: Comparison of the evaluation methods

The results of the SOM are a big improvement compared to MSE or SSI. Nearly all images were classified correctly, only *one* Error-2 image was classified as Error-1. Table 7 shows how the classification success increases, depending on the amount of items used for training the map. Here, the set was split 70:30 (classification:training). We then started with one training item per group and increased the amount gradually. Just five training images per set were enough to level down the false negatives and false positives to zero.

Training Item Count	1	2	3	4	5
False Positives	41%	9%	1%	1%	0%
False Negatives	1%	1%	0%	1%	0%

Table 7: False positive and false negative rate in correlation to the amount of training items (per set).

For practical purposes, it would only be necessary to distinguish between correct and incorrect images, which would further lower the complexity of the decision. Naturally, the quality of the results depends on the used images, the selection of the features and the quality of the SOM implementation. Hence, it would probably be unrealistic to expect equal good results in all cases. But the example shows the potential of this approach.

Concerning the performance of the algorithm, it can be stated that the runtime is linear with respect to the amount of items used and quadratically with respect to the width of the lattice (see tables 8 and 9). On a 2.4 GHz dual-core mobile CPU with 8 GB RAM training and classification both took approximately 5.9 ms per item using a lattice width of 100. Feature extraction takes most time: approximately 600 ms per image (1600x1000 pixels). It has to be noted that the feature extraction was not optimized for performance in any way and that the code was not multi-threaded.

Width (px)	32	64	128	256	512
Duration (s)	0.183	0.291	1.006	4.068	15.864

Table 8: The performance dependent on the width of the lattice

Items	8	16	32	64	128
Duration (s)	0.75	0.142	0.255	0.465	0.96

Table 9: The performance dependent on the amount of items (training)

A possible software production workflow using quality assessment could look as follows: a number of scenes is chosen, possibly using all critical parts of the rendering engine. Two pools of images are created, one with correctly rendered images, and one with images containing artifacts or errors. For each scene, the best features are selected, a SOM is trained and saved as file. Each time the code has to be assessed, an amount of images is rendered for each scene, the corresponding SOM lattice gets loaded and the images are classified. Even if one might not want to fully rely on automatic evaluation, this can give at least a good hint on whether the images are correct or not.

6 CONCLUSION AND FUTURE WORK

In this paper we proposed the use of image quality assessment methods for automatically testing rendering software. The use of automatic quality assessment can ease the rendering software development process, since errors in software development can be discovered early and do not remain undetected until the final version of the software is released. Depending on the characteristics of the scene to be rendered, traditional assessment methods such as MSE, PSNR and SSIM measures perform differently. They are quite helpful for small detail differences, but deliver wrong results if procedural or randomized algorithms are used in the rendering pipeline.

In the latter case, using Self Organizing Maps can compensate and automatically recognize and categorize errors. After a first training phase, SOMs deliver quite accurate results and allow to automatically warn if the image being generated by the software has problems. Of course, the methods proposed for quality assessment have still to be expanded and tuned for their application in real, less controlled situations. In other words, their applicability under real life circumstances has to be thoroughly tested: standard image sets with a variety of images containing artifacts and rendering errors of different types have to be developed. The self organizing map should then be trained with a selection

of erroneous images and, of course, with correct images. Then, the classification must take place with images containing errors that did *not* appear in the images used for training. Only this way a realistic estimation of whether this method is really suitable for practical use is possible. The implementation of the self organizing map still offers great potential for improvements, considering the ordering of the training items, the initialization of the map prior to training and the training process itself.

The selection of the features is a factor with a lot of potential for improvement. As mentioned before, there are alternative approaches for the discrimination of the really useful features with respect to a specific scene, like principal component analysis.

Also, instead of a SOM, other types of neural networks could be used and be tested for their suitability in image quality assessment. For instance, multilayered networks in conjunction with the back-propagation algorithm can be considered [Roj96]. There is also a wide diversity of learning methods like for instance Quickprop and network types like the Hopefield Model which can also be considered. Furthermore, variants of SSIM like Multi-scale SSIM, Modified gradient-based SSIM [LB09] or Complex Wavelet Domain Structural Similarity Map [WB06] [CW12] could be tested.

As a conclusion, this paper is a simple start of a new theme which is worth exploring, since it bears a lot of potential to ease hardware and software development for Computer Graphics.

7 REFERENCES

[AGB99] Apodaca, Anthony A. and Gritz, Larry, Advanced RenderMan: Creating CGI for Motion Picture, Morgan Kaufmann Publishers Inc., 1999

[Bec02] Beck, Test Driven Development: By Example, Addison-Wesley Longman Publishing Co., Inc., 2002

[BGV92] Bernhard E. Boser, Isabelle M. Guyon, and Vladimir N. Vapnik. 1992. A training algorithm for optimal margin classifiers. In Proceedings of the fifth annual workshop on Computational learning theory (COLT '92). ACM, New York, NY, USA, 144-152. 1992

[CW12] Jiayin Chen and Charles A. Wuethrich. "Analog film, digital images and sampling grids: a Fourier and Wavelet analysis." In D. Thalmann, K. Zhou, "CGI 12, Proceedings of Computer Graphics International 2012", Bournemouth, UK, July 2012.

[Kit13] CDash, http://www.cdash.org/, Kitware, 2013.

[KT82] Kohonen, Teuvo. Self-Organized Formation of Topologically Correct Feature Maps. Biological Cybernetics 43 (1): 59-69. 1982.

[LB09] Li, Chaofeng and Bovik, Alan C., Three-component weighted structural similarity index, Proc. SPIE 7242, Image Quality and System Performance VI, 72420Q, 2009.

[Low99] Lowe, David G., Object recognition from local scale-invariant features, Proceedings of the International Conference on Computer Vision. 2. pp. 1150-1157. 1999.

[Ran71] William M. Rand, Objective Criteria for the Evaluation of Clustering Methods, Journal of the American Statistical Association, Vol. 66, No. 336, 1971.

[Rei08] Erit Reinhard, Erum Arif Khan, Ahmet Oguz Akyuz, Garret M. Johnson, Color Imaging: Fundamentals and Applications. A K Peters, Ltd., 2008

[Rob08] Martin, Robert C., Clean Code: A Handbook of Agile Software Craftsmanship, Prentice Hall PTR, 2008

[Rob12] Robert Herzog, Martin Cadik, Tunc O. Aydin, Kwang In Kim, Karol Myszkowski and Hans-P. Seidel, NoRM: No-Reference Image Quality Metric for Realistic Image Synthesis, The Eurographics Association and Blackwell Publishing Ltd., 2012

[Roj96] Rojas, R., Neural Networks: a systematic introduction] Neural Networks: a systematic introduction, Springer, 1996.

[WB06] ZhouWang and Alan C. Bovik, Modern Image Quality Assesment. Morgan & Claypool Publishers, 2006.

[WBS03] Z. Wang, A.C. Bovik, H.R. Sheikh, and E.P. Simoncelli, Image quality assesment: From error visibility to structural similarity. IEEE Trans. Image Processing,13(4):600C612, Apr. 2004.