INSTITUT FÜR INFORMATIK
DER LUDWIG–MAXIMILIANS–UNIVERSITÄT MÜNCHEN

Master's Thesis

Non-Photorealistic Rendering in Virtual Reality

Dominik Schulz

INSTITUT FÜR INFORMATIK
DER LUDWIG–MAXIMILIANS–UNIVERSITÄT MÜNCHEN

Master's Thesis

Non-Photorealistic Rendering in Virtual Reality

Dominik Schulz

Aufgabensteller:	Prof. Dr. Dieter Kranzlmüller
Betreuer:	Dr. techn. Christoph Anthes
Abgabetermin:	22. August 2016

Hiermit versichere ich, dass ich die vorliegende Master's Thesis selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

München, den 19. August 2016

. . . . . .

(Unterschrift des Kandidaten)

Abstract

Non-photorealistic rendering (NPR) is a research field that has been studied since the 90s and consists of algorithms and methods that are able to render images or 3D models in a way that they appear artificial for example like a painterly drawing. The Goal of NPR is to computer generate these artificial effects in order to be able to apply these methods for example better visualize or draw images such as the illustration of buildings. Virtual reality (VR) is a technology that consists of methods to generate a virtual environment that is computed by a computer but still lets the user think he is part of this generated world. The user should also interact with this virtual environment. One of the fields where VR is used is to illustrate buildings or in the automotive industry for prototyping of new car models.
This thesis analyses NPR algorithms and methods in the context of VR. Since NPR in context of VR is not researched very well this thesis analysis algorithms and methods of NPR and applies them to VR in order to evaluate there ability to contribute to VR by analyzing and benchmarking those. The thesis describes the implementation of NPR algorithms inside a prototype application and there benchmark results by using standard models in computer graphics. Besides these topics the results are presented and compared to each other.

Abstract

Nicht fotorealistisches rendering (NPR) ist ein wissenschaftliches Gebiet das seit den 90er Jahren erforscht wird. NPR besteht aus Algorithmen und Methoden die es ermöglichen Bilder und 3D Modelle so zu rendern, dass diese abstrakt und künstlich wirken wie zum Beispiel ein Gemälde. Das Ziel von NPR ist es diese Methoden mit Hilfe eines Rechners zu generieren damit diese angewendet werden können, z.B. in der Visualisierung oder für digitale Zeichnungen wie etwa das darstellen von Gebäuden. Virtuelle Realität (VR) ist eine Technology mit welcher es möglich ist eine Virtuelle Umgebung zu erzeugen, welche von einem Computer erzeugt wird. Der Benutzer dieser Umgebung soll denken, dass dieser ein Teil davon ist und ebenso sollte er mit der Umgebung interagieren. Ein Anwendungsgebiet für VR ist ebenfalls das Darstellen von Gebäuden oder auch in der Automobil Industrie wo es für Prototypen neuer Fahrzeugmodelle genutzt wird.
Da NPR in Verbindung mit VR bisher nicht sehr viel erforscht wurde wird diese Thesis Algorithmen und Methoden aus dem Gebiet NPR nutzen und diese in VR anwenden. Ziel ist es zu evaluieren wie diese VR unterstützen indem sie analysiert und verglichen werden. In dieser Thesis wird neben Analyse und Vergleich auch die Implementierung der NPR Algorithmen und Methoden innerhalb eines Prototypen beschrieben. Neben der Implementierung werden die Messergebnisse, welche beim Darstellen von standard Modellen im Bereich der Computer Grafik gemessen wurden analysiert.

Contents

1. Introduction

Non-photorealistic rendering (NPR) techniques have been researched beginning in the early 90s, which resulted in a wide range of different methods, algorithms and scientific research. One of the first persons researching NPR was Paul Haeberli, with his publication about image stylization "Paint by numbers: abstract image representations" [Hae90]. However, (VR) itself was researched since the 60s, where Ivan E. Sutherland published his first work about a conceptual VR display "The Ultimate Display" [Sut65]. Only a few publications combine both worlds the one of NPR and the world of VR. This thesis describes several NPR methods, where the focus is on NPR effects that can be applied to VR, since VR requires fluent image rendering. NPR targets the opposite of photorealistic rendering which tries to computer generate (render) pictures of 3D data in a way that it looks similar or equal to the real world for example a car that drives over a bridge. Where NPR targets to render images of 3D data in a way that they look artificial for example an image that looks like drawn with the pencil.
One of the main parts of this thesis is to evaluate which methods can be applied to VR from the technical point of view and which of these methods contribute to VR and visualization in a way that improves recognition of details in an rendering, for example highlighting a specific part of the image. Furthermore, VR has some requirements that have to be respected. First of all, when rendering images in VR they need to be rendered for each eye in real-time to generate a stereo 3D effect for the observing person. Second is that depending on the display or environment used, synchronization of multiple displays can cause problems or increase computation time. Besides these issues, it is necessary to check, if displays overlap, that the images are compatible to each other in order to not distract the user with gaps in between the images.
The thesis describes for each of the NPR effects that can be applied how these can be used and how they work in VR. Besides that, the main focus of the thesis is to integrate these effects into a multi-display virtual environment for example the Audio-Visual Automatic Virtual Environment short CAVE[1], which is used for visualization and based on the work "The CAVE: audio visual experience automatic virtual environment" from Carolina Cruz-Neira [CNSD+92]. The CAVE itself consists, besides the multiple displays, of multiple computer systems where each of these systems also called render nodes display its renderbuffer onto one display. This makes it necessary to synchronize data between each of the nodes.

1.1. Motivation

The topic of NPR is widely spread and also heavily applied in the game industry to create state of the art games for the consumers. Also visualization is often using NPR effects to enhance immersion of the visualized data or models in order to assist the human eye and

[1] CAVE™ is a registered trademark of the University of Illinois Board of Trustee. The term is used in the context of this thesis to generically refer to CAVEs™ and CAVE-like displays.

observer. An example for non-photorealistic visualization can be the artificial rendering of a new shopping mall to illustrate roughly how it should look like in the end. Since VR is getting more attention in the past years the combination of both worlds is very interesting, but not researched very well. This fact makes it very interesting to research as well as finding and solving problems related to non-photorealism and VR. Besides that, NPR effects can generate very appealing effects. The main parts of these thesis are to evaluate current NPR effects and algorithms that can be applied to VR environments, in detail which can be applied to multi-side virtual reality environments such as the CAVE, with the focus on real-time and applying those on different data sets. Besides that the topic of this thesis is to compare these algorithms that can be applied. One of the main goals is to focus on effects and algorithms that work in general on any kind of 3D data that can be visualized with OpenGL and standard computer graphics hardware. Algorithms that need pre-processing or pre-computation will be left out since they would add additional effort when trying to use them for general visualization, since the goal of the thesis is to evaluate a set of plug and play non-photorealistic effects for VR and visualization. More precisely, this thesis discusses:

- Analyzing NPR algorithms and methods
- Implementing a prototype VR application using NPR algorithms
- Benchmarking the NPR algorithms
- Implement plug and play NPR algorithms
- Discussing the usage of those NPR algorithms in VR

1.2. Overview

This Section gives a short overview about the structure and contents of this thesis which is as follows:

- Chapter 2 discusses related work regarding VR and NPR as well as research that is combining both. It consists of fundamentals, rendering and image based methods, NPR perception and viszualization.
- Chapter 3 describes the concept of the thesis by pointing out the structure of the application, the rendering algorithms and effects. Besides these points the chapter also points out how measurement is done.
- Chapter 4 gives an overview and details about the implementation of the prototyping software that was used to implement and evaluate the algorithms
- Chapter 5 lists and analyses the measurement results. The algorithms are compared to the others.
- The final Chapter 6 summarizes the results of the thesis and gives examples for future or upcoming work to continue the topic.

2. Fundamentals and Related Work

Related work in the scope of non-photorealistic rendering (NPR) is diverse and full of topics and research about cell shading, line drawing and various other 3D as well as 2D rendering methods. While in the combination with virtual reality (VR) there are only a few studies which for example describe stereoscopic 3D visualization. The following section will summarize rendering based methods and image based methods found in the context of NPR as well as giving a short introduction about fundamentals in rendering and NPR. Besides that it gives some introduction to NPR perception as well as NPR in context of visualization.

2.1. Fundamentals

This section should give a short overview about the most common techniques in VR and NPR. It describes their purpose as well as shortly explains what these methods are doing.

2.1.1. Virtual Reality

VR designates a technology where a computer generates a virtual world, in which the user feels to be part of. The virtual world is computer generated in real-time and should offer functionality to interact. The interaction is needed in order to let the user feel he is part of the world and not only watching it, like a cinema movie. Besides real-time and interaction also stereoscopic representation is needed that the user can see a 3D computer generated scene. Virtual reality (VR) can be used with various hardware, for example the CAVE or head-mounted displays. Head-mounted displays are becoming more and more attention and available also for private users, where a CAVE is usually expensive and only available for research and the industry.

2.1.2. Cartoon Shading or Tone Shading

Cartoon or Tone shading means to generate a cartoon style image by applying specific rendering techniques. Most common is that at a distinct point while applying lightning the shading of the scene is strongly changed. This results in hard gradients which are usually used when drawing cartoon scenes. An example can be seen in figure 2.5a. One of the works describing Cartoon or Tone shading is “Stylized Rendering Techniques for Scalable Real-Time 3D Animation“ from Adam Lake et al. [LMHB00]

2.1.3. Edge Detection

Edge detection is used to find edges, either in 3D space or image space depending on the algorithm used. The basic idea behind these algorithms is to find important edges extract them and make them usable for further computation. The result of these algorithms is often used to enhance already rendered images to achieve a cartoon style image or a painterly

image. A publication about edge detection can be found in "A 3x3 Isotropic Gradient Operator for Image Processing" [SF68] which describes the Sobel edge detector.

2.1.4. Edge Flow Image and Flow Field

Edge flow image and flow fields are used to detect in which direction edges and or strokes are heading. Both of them describe in which direction a former edge or stroke was drawn or could be drawn when applying brush strokes or similar techniques on them. These techniques are used to retrieve a more realistic effect which takes the direction into account.

2.1.5. Stereoscopic Rendering

The term stereoscopic rendering means to render an image of a 3D scene twice one time for each eye. This is the common method to produce a 3D effect for our human eyes. For each eye the scene is rendered with there particular view to emulate how we see the world. Rendering stereo 3D can be achieved with three different methods first with parallel projection where the image is rendered with an offset between two camera perspectives. The second method is the toed-in method which also uses an offset between the cameras but additionally the cameras are pointing to towards an single focal point. The last method is the off-axis method where the camera frustum is non-symmetric but both cameras are pointing to the projection plane in parallel. For Stereoscopic Rendering the off-axis method should be used since researched showed that this generates the best 3D effect for the human eyes.

2.1.6. Multipass Rendering

Multipass rendering consists of multiple rendering passes sequentially or combining the results of different rendering passes. The common use cases are to render the scene into a texture and then apply a second rendering pass onto that texture for example edge enhancement. In the final pass the result of the first and second pass are combined to have a rendered scene with enhanced edges. Figure 2.1 shows a more complex multipass rendering.

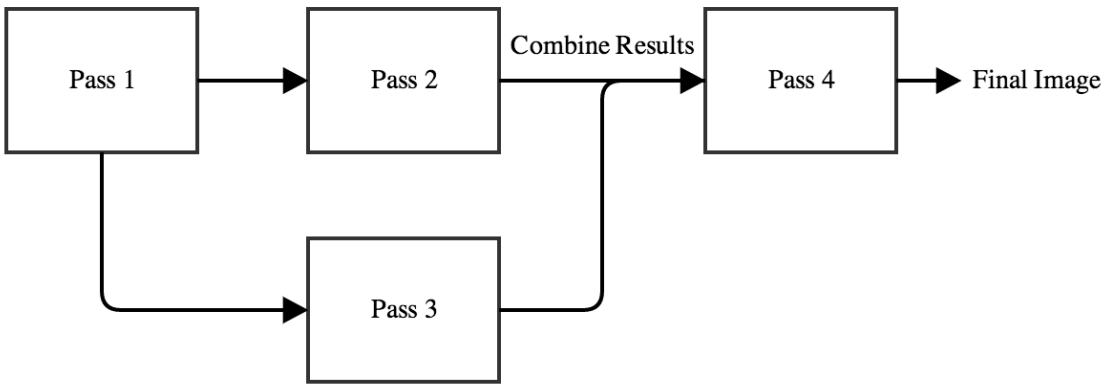


Figure 2.1.: Multipass rendering showcase

2.1.7. 3D Textures and Brush Textures

When rendering scenes often textures are used to draw complex structure onto 3D meshes, 3D textures are a special type of textures for example these textures do not only have 2-axes like usual textures but one more, this can be used to store different data in each level of the 3rd axis or to optimize the rendering process without wasting resources. Brush textures are a special usage form of 3D textures, those are used when it is needed to draw brush strokes in the scene by using multiple textures that consist of brush strokes themselves. In combination with 3D textures each level in the 3rd dimension could have different brushes depending on the strength of the stroke.

2.2. Rendering Based Methods

Rendering based methods usually work on 3D data as a source for example the 3D mesh which should be rendered. The outcome of these rendering methods can be a fully rendered scene which has no need for change or adaptions anymore. In case and if required the rendering based methods can also be used as pre processing step to the image based methods (see section 2.3).

2.2.1. Real-Time Methods

The paper "Non-Photorealistic Virtual Environments " by Allison W. Klein et al. from the year 2000 [KLK+00], describes a real-time virtual environment NPR topic. The the paper introduces a method on how to use NPR filters on images of real or artificial environments, to render a virtual environment. Besides that, the algorithm also draws 3D lines into the virtual environment, which enhances the NPR effect by adding extra digital information which gives the impression of a drawn picture.
The main method which is described in the paper is the computation of non-photorealistic textures from photographs, that largely avoid seams in the final image. The result is that the whole scene looks like drawn, by using filters to emulate van Gogh style, pastel or for example watercolor. The method in this article delivers great results in the final rendering see figure 2.2, but it needs a lot of offline pre-processing to apply the filters, before the actual rendering.

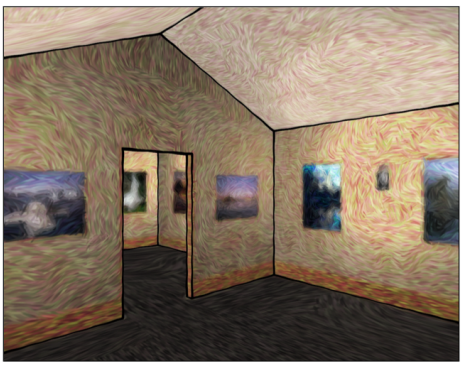


Figure 2.2.: A NPR virtual environment [KLK+00]

The next work to mention is the "Hardware-Accelerated Parallel Non-Photorealistic Vol-

ume Rendering“ by Eric B. Lum et al. [LM02]. It describes the efficient combination of different rendering techniques such as tone shading, silhouette rendering, gradients and the use of color cues. The rendering examples from figure 2.3 show that for example computed tomography scans can be rendered and visualized very well by using these rendering techniques. The example combines tone shading and adds also a silhouette around, after these steps gradients are added and color cues to receive a semi transparent effect. To compute such renderings, the capabilities of consumer graphics hardware are used, for example 3D textures. With 3D textures multiple rendering passes can be avoided.
In the article a pc cluster is used to compute the images and to run it fast enough for real-time modifications such as translation, rotation etc. Since the hardware used is from the 2000s it should be possible to compute these effects on an up-to-date pc nowadays.

Figure 2.3.: Rendered computer tomography scan [LM02]

“Real-Time Pencil Rendering“ from Hyunjun Lee et al. [LKL06] combines run-time processing and pre-processing together to achieve real-time renderings which look like pencil drawings. The basic idea behind is the combination of multiple steps together to generate the final result. Parts of these steps are contour detection where a fast image based approach is used. Next step is shaking the contour and then draw it multiple times to achieve the effect that the image is drawn with multiple strokes. Shaking the contour means to draw the contour not plain but in a way a human would draw it with his hand.
Then the result is combined with the offline generated pencil and contour textures, as well as with the paper normal texture. A paper normal texture is similar to an normal texture but it specifically consists of normals that can generate a paper effect. The paper normal texture and the pencil texture are applied to the interior to simulate pencil shading. The contour texture is applied to the extracted contour to imitate pencil drawn contours. This real-time approach could be used also in VR since it can be computed fast, as long as for each scene to render the necessary pre-processing steps are done. One example of this method can be seen in figure 2.4

The the paper “Stylized Rendering Techniques for Scalable Real-Time 3D Animation“ from Adam Lake et al. [LMHB00] describes a combination of real-time rendering techniques which support 3D animations including 3D rendering.
First of all the work starts with cartoon shading. Cartoon shading makes use of the dot

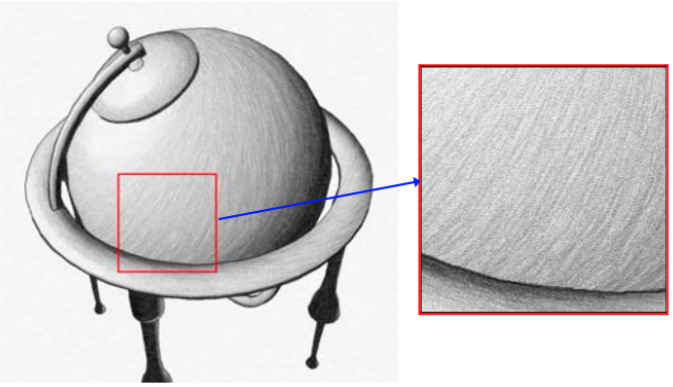


Figure 2.4.: Real-time pencil rendering [LKL06]

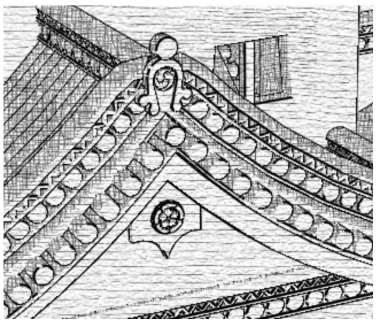
product between the face normal and the light vector to shade the model. If this equation 2.1 results in a value of 0.5 or less the pixel is shaded darker than the ones above which results in a shadow border at 0.5 see figure 2.5a

$$\vec{L} \cdot \vec{N} \tag{2.1}$$

Another method applied in the paper is "Pencil Sketch Shading" it uses the same equation 2.1 to choose a texture with the required density. The textures represent pencil strokes, they are generated by randomly selecting different kinds of strokes. All of these textures are generated in different densities. The next rendering technique used is silhouette detection which means the silhouette of a model is detected and drawn to the screen. The approach used to achieve this effect is the algorithm that tries to find faces which share their edge while one of them is facing to the front and the other is facing to the back, this is done by using the normals and checking if these faces are facing in opposite directions. In the next step the edges are rendered as stylized edges, the basic approach consists of three textures one having a rightward stroke, the second having a straight stroke and the last having a leftward stroke.
During rendering of the edge it is decided which of these three textures to take, by checking which angle the current edge to render and its successor edge together have, for an example see figure 2.5b. The last technique described is used to draw motion lines to support moving of an object. The algorithm chooses `n` vertices in a pre-processing step which will be the vertices to start the motion lines. The algorithm keeps track of the motion of an object and renders lines from the past positions of these `n` vertices. The algorithms described in publication are commonly used already and can also be applied in real-time. Since the results achieved fluent results already on a Pentium III©computer system these should be usable also in virtual reality.

A good composition of NPR effects can be found in "Post-processing NPR Effects for Video Games" from Milán Magdics et al. [MSGS13]. The paper describes different approaches of edge detection and rendering as well as texture simplification which means textures of objects are simplified by removing details. Besides these methods the shadows are recolored by extracting them and increasing the contrast. Also depth saturation with varying abstraction levels is described which is used to enhance the depth perception as well as steers the focus on important things. As last points blur effects, contour thickness and color palette modification

(a) Cartoon shading



(b) Pencil sketch shading

Figure 2.5.: Examples of stylized rendering techniques [LMHB00]

is explained. Color palette modification is in detail described as an effect where the colors are modified for example by applying the same atmosphere of one image to the output of the rendering pipeline. This results in a rendered picture which has warm colors if a photo of a sunrise was used. An example of palette modification, contour modifications plus edge detection can be seen in figure 2.6.

Figure 2.6.: Car rendering [MSGS13]

Also in augmented reality a NPR topic can be found. The publication from Michael Haller "Photorealism or/and Non-Photorealism in Augmented Reality" [Hal04] shows how to use cartoon shading and painterly rendering in augmented reality. Haller applies this rendering techniques in his example to an 3D model of a Van Gogh painting. The people are able to look deeper into the painting with the 3D model. In the algorithm for cartoon shading a two-valued step function is applied to replace the diffuse and the specular color these values are calculated with the help of a 1-dimensional texture. The basic idea behind is the same as in "Post-processing NPR Effects for Video Games" [MSGS13]. To achieve the painterly rendering effect, the algorithm uses three steps, the first is a pre-processing

step then a first rendering pass and a second rendering pass. In the pre-processing step the mesh of the 3D model is used to place particles in world space. The first rendering pass is used to render the color and the depth texture of the 3D model. A depth texture consists of depth information for each pixel on the screen which means that different color shows different depth. Before it comes to the second final rendering pass culling is applied to remove unnecessary information. The second pass generates billboards for each pixel and applies the stroke textures onto them to render the final image. To retrieve the orientation of the strokes the normals of the surface are used. The last pass is the most expensive part of the rendering process, however it shows good painterly results see figure 2.7. In this work about augmented reality are no results documented about performance, but since augmented reality heavily depends on real-time processing and also the paper states that the focus was real-time it could be also useful in VR if it can handle processing of a complete virtual world.

Figure 2.7.: Painterly rendering in augmented reality [Hal04]

A good way to draw silhouettes is described in "Coherent Stylized Silhouettes" from Robert D. Kalnins [KDMF03] the basic work shows how to draw stylized silhouettes, but their main focus is to draw coherent silhouettes which also coherent from frame to frame to look more realistic. Behind the algorithm is the approach that they draw in the first frame the silhouette by finding it, and drawing it with brushes. These lines are saved for the next frame, where the same process starts but in this step the new lines are checked against the old ones to see if something moved, this makes it possible to remove flickering of the silhouette, by drawing the new silhouette nearby the old silhouette. An example of the output of the algorithm can be seen in figure 2.8.

2.2.2. Non-Real-time Methods

One non-real-time method is the research topic "Stereoscopic 3D Line Drawing" from Kim Yongjin et al. [KLKL13], the paper describes two basic rendering approaches the center-eye-based, which uses the lines in a view from a centered perspective for both eyes to draw stereo lines, and the each-eye-based stereo line drawing, which uses lines from the perspective of

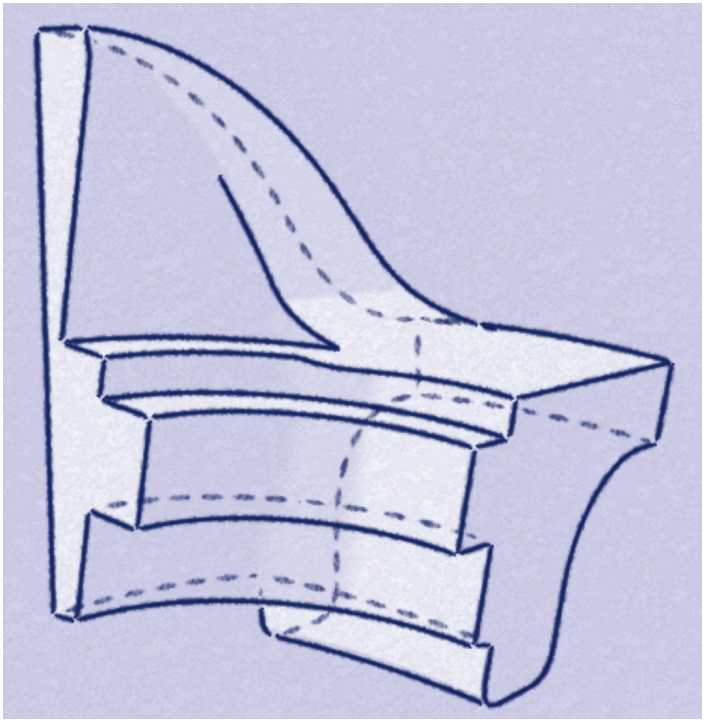


Figure 2.8.: Stylized coherent silhouette [KDMF03]

each eye to draw. The research focuses on using the each-eye-based approach since it is the only one giving valid results for each eye, even if there are issues like binocular rivalry, that can occur when one of the eyes has a different picture for example one eye sees a line the other eye can't see. The basic approach of drawing the stereoscopic 3D lines is to find points connected via lines that exists for each eye view, if there are such lines, the lines are drawn for both eyes. This simple method needs a lot of computation in object space which is not applicable for real-time rendering. To solve performance issues and reduce the computation amount a faster approach is introduced which runs in image space. However, the results of both methods look good these can't be applied to a VR environment such as a CAVE since the method is not scaling well. There results showed that an PC with a 2.66 GHz Intel Core i7 CPU 920, 6 GB memory, and an NVidia Quadro FX3800 graphics card can only render 30000 vertices at about 3fps. See figure 2.9 for an example of 3D stereo lines.
The paper "Blueprints - Illustrating Architecture and Technical Parts using Hardware- Accelerated Non-Photorealistic Rendering" from Marc Nienhaus et al. [ND04] shows how to render blueprints of 3D scenes with the help of depth images and depth peeling. The algorithm uses multiple passes to cut out the silhouette edges and the crease edges. Besides these edges which are common in line drawing, the research shows how to draw lines which are hidden such that the rendered objects look transparent, which is common for blue prints. The biggest problem about the algorithm is the heavy computation amount, in year 2004 the algorithm achieved with an NVidia GeForce FX 5600 5fps with a resolution of 512x512. In modern CAVE installations it is common that the view for 5 Sides needs to be rendered. Each side can have a high resolution which needs to be rendered for each eye. That is the reason why it is unusable in such installations, the rendering has to be fluent. Besides that, the method shows great results especially for architectural illustrations and blue prints which

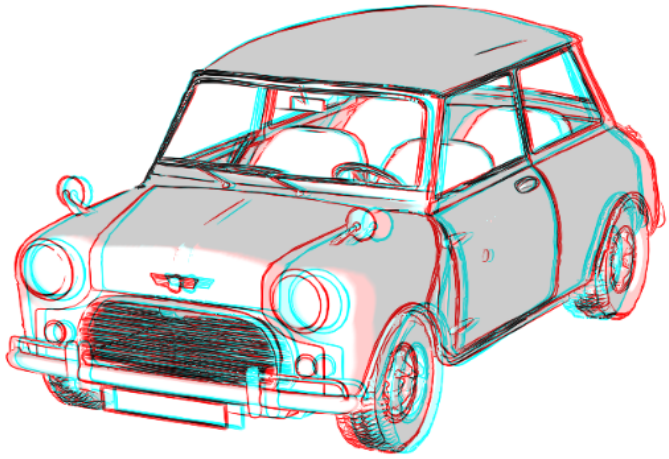


Figure 2.9.: 3D stereo lines [KLKL13]

can be seen in figure 2.10

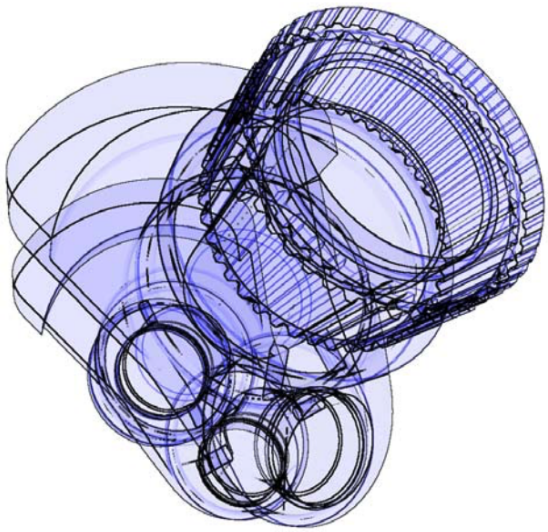


Figure 2.10.: NPR Blueprint [ND04]

The work “Programmable Rendering of Line Drawing from 3D Scenes“ from Stéphane Grabli et al. [GTDS10] describes a non-real-time rendering process to render line drawings from 3D models. The idea behind the process is following: The 3D scene is used to extract feature lines and the projection the result of this process is called view map. The user defines in the so called style description different style modules to be applied, the language used is python for the style definition. To get the final result each style module is applied to the view map and in the end put together to the final image. One of the reasons why this approach can’t be applied in real-time is that the computation of the view map according to the paper takes seconds to several minutes for a model with 50000 polygons. However, this method

can deliver great results as seen in figure 2.11.

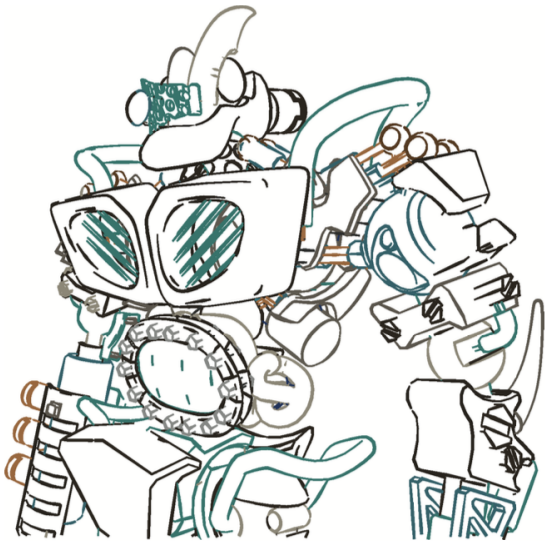


Figure 2.11.: Example of programmable rendering [GTDS10]

2.3. Image Based Methods

Image based methods are operating on image space only, that means these methods take a 2D image either real or rendered as input and compute these. The output can vary from for example edge detection to stylized images. The image based methods can also be chained like rendering based methods to apply multiple effects. First a stylized method could be applied to an image and then it could be grey-scaled to get a painterly image in black and white.

2.3.1. Stylized Effects

One image based method is the one described in "Non-Photorealistic Rendering with Spot Colour" by Paul L. Rosin et al. [RL13]. The basic idea behind is to take photos and render them using the spot color technique, which means the image will be rendered as greyscale image expect some color called spot color. The paper also shows some examples where other rendering techniques are combined with spot color like tone shading.
The first step in the algorithm is to transform RGB colors to hue and saturation which makes it possible to better match colors. Also the color channels are smoothed with a Gaussian kernel. The article describes multiple criteria to get the spot color of an image, for example based on the shape of an image, based on salience or based on the background. All of them deliver different results. The algorithm delivers great results see figure 2.12, however it is not really fast since it takes 2 seconds to add spot color to an image with 0.6M pixel on a 3.40 GHz Intel Core i7. The good thing of the method in the paper is fully computed on CPU not GPU hardware. This means it should be possible to implement a mix out of spot color and picking in an CAVE environment and for example support selection of objects.

Figure 2.12.: Spot color filter applied to an image [RL13]

The “Image Stylization by Oil Paint Filtering using Color Palettes“ from Amir Semmo et al. [SLKD15] introduces an Algorithm to transfer a digital image into an oil like painting by applying multiple steps of image processing. First of all, a user specified number of colors is extracted to a color palette, which is later used to draw again. Then for the colors of the palette the placement is found and the according color is used to paint the region. After this step the luminance is quantized and in the end the image is smoothed. From the original images the edges are detected proceeded to a flow field where brush textures are applied finally after the textures are vanished. As last step the results from the smoothed image, the edge detection and the vanished textures are put together to the final image which results in a painterly like image see figure 2.13. First of all, each of these steps needs a lot of computation which results in long rendering times.
The paper states that the algorithm needs for an Image of 800 x 600 pixels and a color palette of 25 around 50 seconds. The hardware used was an Intel©XeonTM 4x 3.06 GHz and an NVidia©GTX 760. This makes it impossible to use this algorithm in VR since it is not running in real-time. A work with a lot of different rendering techniques can be found

Figure 2.13.: Oil painting [SLKD15]

in “Non-Photorealistic Rendering of Portraits“ from Paul L. Rosin et al. [RL15]. The work is focused on rendering portraits as non-photorealistic portraits. First of all, they start with fitting the face model which results in a smoothed and filled shape of the face. Since this shape lost a lot of information they refine the face model to readd for example the shape of

the mouth.
As next step they render the skin again, but they apply also Gaussian filters to smoothen the skin over there facial model. To add more detail, the algorithm finds smooth lines and renders them into the picture. As nearly last step a shading effect is added so that the picture gains again more three dimensional details. The last step consists of adding more detail for the eyes, the hair and other important details, which need to preserve the rendering to a NPR from the original image.
The algorithm delivers accurate results see figure 2.14. However, the computation is expensive, the paper states that it takes around 30 seconds to render a 0.5 megapixel portrait on a 3.40 GHz Intel Core i7 with the unoptimised code. Since this work was done in 2015 it does not make sense to apply it in a VR environment.

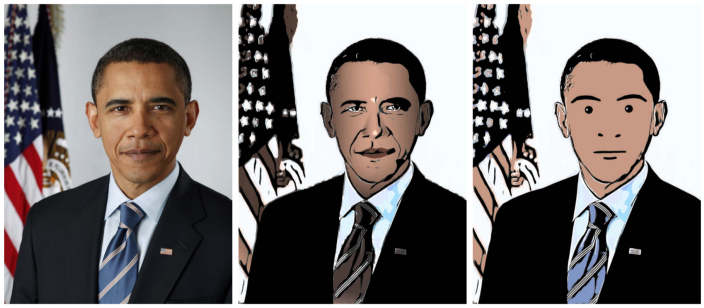


Figure 2.14.: NPR Portrait of president Obama[RL15]

2.3.2. Edge Detection

An edge detection method developed by Sobel and Feldman documented in the paper "A 3x3 Isotropic Gradient Operator for Image Processing" [SF68] is also called Sobel filter. The algorithm uses two 3x3 matrices for each axis one of them, that means in image space one for the x-axis and one for the y-axis. These matrices are also called convolution kernel and applied to each pixel, which means that the matrices are applied in a field of 3x3 around the pixel. Both results from the x and the y axis are then combined to give the resulting pixel. The final result consists of an image which is black and has white lines which represent the detected edges, see figure 2.15.
The use of matrices and vectors makes it easy to apply the Sobel filter to a rendering pipeline

Figure 2.15.: Applied Sobel filter before and after

with the help of an shader, since the graphics hardware heavily uses matrices vectors and the math to compute these.
A second approach about image based edge detection is the work “Fast Boundary Detection: A Generalization and a New Algorithm“ from Werner Frei and Chung-Ching Chen [FC77]. The work uses the same approach as the Sobel filter by using 3x3 matrices and so on. The filter developed by Frei and Chen adds seven more matrices to the processing, the result is that the improved filter finds more edges also in images and detects normal edges as thinner edges. This means the results will be more accurate and more valuable for image processing, where the results of both the Sobel filter and the Frei Chen filter heavily depend on the processed images, see figure 2.16 for a comparison of the Sobel filter and the Frei Chen filter.

Figure 2.16.: Left original image, middle Sobel filter, right Frei Chen filter

A more advanced and accurate edge detection and line drawing filter can be found in the work “Coherent Line Drawing“ from Henry Kang et al. [KLC07]. The paper shows how to implement a fast and accurate line drawing filter. The basic idea is to first construct the edge flow image from the original image and then process it with difference of Gaussian (DoG) filtering from the flow image. To compute the flow image and to apply difference of Gaussian it is needed to apply multiple kernels and calculations.
The results presented in this paper are that the flow image takes around two to four seconds and to apply the difference of Gaussian filter it takes four to six additional seconds on a dual core computer system with 3 GHz. In this case it is not possible to apply these filter to VR since each frame would take to long to render. Figure 2.17 shows the two step process of computing the final image.

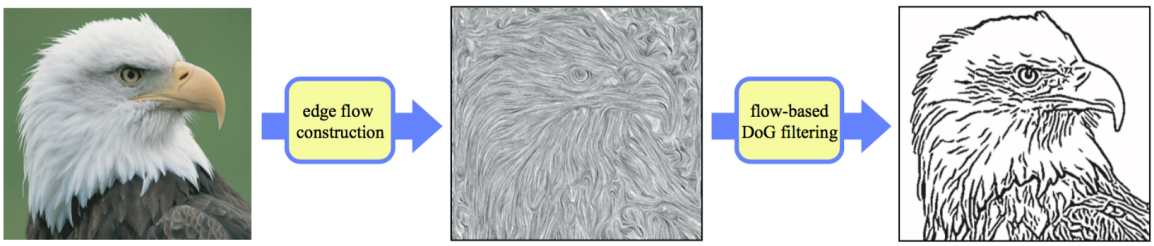


Figure 2.17.: Coherent line drawing [KLC07]

A second paper which describes oil painting like image processing is “Oil Painting Rendering through Virtual Light Effect and Regional Analysis“ from Sungkuk Chun et al. [CJK11].

The visual results are not that good than the ones of [SLKD15], but still good. The algorithm in the paper extracts edges with the help of the sobel filter [SF68]. The second step is to generate the painterly rendering by using the edges and apply them with brush strokes onto the image. As final step the intermediate image is transformed with virtual light to simulate the effect of real oil paintings which look different according to the angle of the light falling onto it.
Because there are no measurements, evaluations or benchmarks of the algorithm inside the paper it is not possible to tell if this method can be applied to VR. The steps of the algorithm are illustrated in figure 2.18.

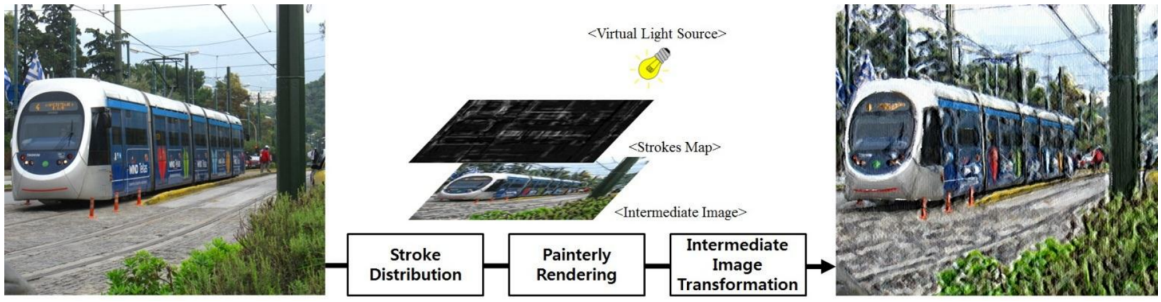


Figure 2.18.: Oil painting with Sobel filter[CJK11]

2.4. NPR Perception

Non-photorealistic perception means how do we as observer perceive NPR effects when using stereoscopic rendering. Which is necessary and good to know in order to receive good rendering results in stereoscopic 3D.

2.4.1. Binocular Depth Perception

When trying to add NPR effects to virtual environments it is good to know how people perceive the depth. The work which was researching this topic is "Binocular Depth Perception of Stereoscopic 3D Line Drawings" from Yunjin Lee et al. [LKKL13]. The paper's focus is on 3D line drawings and it consists of four different test. First of all, the first test was to test monoscopic vs. stereo lines, where all persons tested said that the stereo lines generated a stereoscopic 3D effect. This test was necessary in order to see if stereo drawn lines can produce such an effect.
As second test the work shows the results of a comparison between stylized lines and plain lines. The results of this second case where that 64.8% voted that the stylized version gives them a better distance perception, where 35.2% said the plain version produces a better distance perception.
The third of four tests, tested if a line rendered scene weakens the distance perception against a usually shaded version of the same scene. To achieve the result, the test displayed a line rendered version of the scene and below different version of the scene as shaded rendering by varying the inter-pupillary distances. The question was which of the shaded scenes does best fit to the depth perception of the line drawn scene. Most of the people choose one of the average versions of the shaded rendering.

As last test scenario the paper describes a test to check how lines added to a shaded image can influence distance perception. In order to test this a shaded image was displayed as reference and the person should choose a second shaded image with does fit best in distance perception to the reference. In each version of the test images different amounts of lines where added to the image. As result of this the work mentions that adding lines to stereo-shaded images strengthens distance perception. In figure 2.19 different version of the same object can be seen with techniques from the four test scenarios.

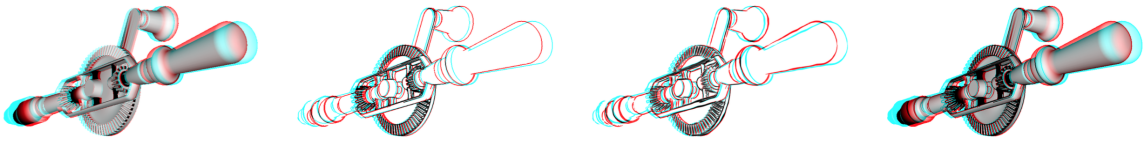


Figure 2.19.: Depth perception test renderings[LKKL13]

2.5. Summary

In order to have a better overview about the described methods and algorithms the following sections should give a better overview about them and shortly summarize what is applicable and usable in VR.

2.5.1. Rendering Based Methods

Name	Real-time	Pre-processing	Summary
NPR Virtual Environments	yes	yes	Pre-processing of input textures, draws 3D lines, painterly rendering style
Hardware-Accelerated Parallel Non-Photorealistic Volume Rendering	yes	no	tone-shading, silhouette rendering, gradients, color-cues, 3D textures for speedup
Real-time Pencil Rendering	yes	yes	contour detection, shaking contour, offline generated pencil, contour and paper normal textures
Stylized Rendering Techniques for Scalable Real-Time 3D Animation	yes	yes	Cartoon shading, pencil sketch shading, silhouette detection, stylized edges, motion lines
Post-processing NPR Effects for Video Games	yes	no	edge detection, texture simplification, shadow recoloring, depth saturation, blur effects, contour thickness, color palette modification
Photorealism or/and Non-Photorealism in Augmented Reality	yes	yes	cartoon shading with step function
Coherent Stylized Silhouettes	yes	no	stylized silhouettes, coherent silhouettes, apply brushes
Stereoscopic 3D line Drawing	no	yes	compare center-eye and each eye based, find connected points, binocular rivalry
Blueprints - Illustrating Architecture and Technical Parts using Hardware-Accelerated NPR	no	no	depth images, depth peeling, multipass rendering, silhouette and crease edges, draw hidden lines, transparent effect
Programmable Rendering of Line Drawing from 3D Scenes	no	no	feature lines, style description, heavy view-map computation

The algorithms in “NPR Virtual Environments“ can be applied to VR since it supports real-time rendering and VR, only downside is that it needs pre-processing to generate the textures

which limits the on the fly usage of loading models etc. The research from “Hardware-Accelerated Parallel Non-Photorealistic Volume Rendering“ is full of rendering methods which run in real-time and do not need pre-processing which makes it possible to use it in VR. “Real-time Pencil Rendering“ uses algorithms that can render in real-time as well but which need pre-processing for the pencil textures and so on, besides the fact of little pre-processing it is useful for VR.
The work “Stylized Rendering Techniques for Scalable Real-Time 3D Animation“ shows how real-time stylized renderings can be done and also requires pre-processing for some parts which makes it partly usable in VR. The research in “Post-processing NPR Effects for Video Games“ has lots of different techniques and also does not require pre-processing which makes it possible to use in a virtual environment. “Photorealism or/and Non-Photorealism in Augmented Reality“ shows non-photorealistic real-time rendering in augmented reality which need pre-processing, besides the pre-processing the algorithm could be applied to VR. A real-time method which needs no pre-processing and is usable in VR is described in “Coherent Stylized Silhouettes“. The method from “Stereoscopic 3D line Drawing“ is not running in real-time which makes it impossible to use it in VR. Also the technique described in “Blueprints - Illustrating Architecture and Technical Parts using Hardware-Accelerated NPR“ is not running in real-time as well, besides that also the algorithm from “Programmable Rendering of Line Drawing from 3D Scenes“ can not be applied to VR as well because it is not running in real-time.

2.5.2. Image Based Methods

Name	Real-time	Pre-processing	Summary
Non-Photorealistic Rendering with Spot Colour	no	no	greyscale, color transformation, spot color, tone shading, color channel smoothed
Image Stylization by Oil Paint Filtering using Color Palettes	no	yes	color replacement and simplification, luminance quantization, edge detection, flow field and brush textures
Non-Photorealistic Rendering of Portraits	no	no	smooth faces, smooth lines, readd details, shading effect
A 3x3 Isotropic Gradient Operator for Image Processing	yes	no	edge detection, 3x3 matrices, fast
Fast Boundary Detection: A Generalization and a new Algorithm	yes	no	edge detection, 3x3 matrices, fast more accurate then Sobel filter
Coherent line Drawing	no	no	edge flow image, difference of Gaussian
Oil Painting Rendering through Virtual Light Effect and Regional Analysis	unclear	no	uses Sobel edge detection, painterly rendering, brush strokes, virtual lighting

The algorithm described in “Non-Photorealistic Rendering with Spot Colour“ has techniques which do not need pre-processing but also do not run in real-time which makes it

unusable in VR. However, a minimized version of the techniques which extracts only specific colors could be applied. "Image Stylization by Oil Paint Filtering using Color Palettes" shows a way to render stylized oil paintings in non real-time with pre-processing, this makes it impossible to apply it in VR. The paper "Non-Photorealistic Rendering of Portraits" shows also non-real-time methods without pre-processing, the algorithms used need a lot of computation which means that it can not be used in VR.
One real-time algorithm with no pre-processing which can be applied to VR is the work "A 3x3 Isotropic Gradient Operator for Image Processing". Besides this work there is also an improved algorithm which is also real-time and without pre-processing described in "Fast Boundary Detection: A Generalization and a new Algorithm". The method described in "Coherent line Drawing" does not need pre-processing but requires a lot of computation which makes it non-real-time and also unusable in VR. A technique where it is not clear if it runs in real-time is the one described in "Oil Painting Rendering through Virtual Light Effect and Regional Analysis", besides this unclear fact it does not need pre-processing. Since it is not clear if the algorithm runs in real-time or not it is also not clear if it can be applied to VR.

2.6. NPR in Visualization

NPR is commonly used in the visualization field. The paper "Illustrative Visualization: New Technology or Useless Tautology?" [RBGV08] from Peter Rautek et al. gives an overview of what happened in visualization in terms of NPR in the past and what to come in the future. First of all, the paper starts with describing so called focus and context techniques, which are used to let the observer focus on some important parts and not on the context itself, an example can be seen in figure 2.20.

Figure 2.20.: Focus and Context [RBGV08]

The image shows how to focus on specific parts by blurring everything else and also as second example how to guide the observer to special parts of the image which is called magic lens in the paper and magnifies special parts. As second component of visualization the paper describes visual abstraction techniques which are used to artificially remove information in the image to enhance the perception of the illustrated object. The described techniques are for example drawing styles like pen, pencil and brush techniques which are stated as effective but difficult to master. The paper also mentions that these techniques are nowadays

providing good results but still distinguishable from handcrafted illustrations, and have been the main focus of NPR research in the past. After describing the research of the past the paper gives an outlook of what will come in the future. For example the paper describes that in the future it should be possible for scientists and other people which need visualization to easily illustrate there images also in a stylized way by having no knowledge in the illustration itself. By now illustrative visualization is seen as tool for efficient communication of knowledge in images.

3. Concept

The thesis itself starts by describing the requirements and the structure of the application. Followed by the implementation of a prototype application that is able to use different algorithms to render 3D stereo images in the CAVE. The prototype is then used to implement non-photorealistic rendering (NPR) algorithms and display them. Finally the algorithms are evaluated by measuring and benchmarking the NPR algorithms in the CAVE. All of these steps are then summarized in the end by giving a summary of the measurement as well as the final conclusion.

3.1. Requirements

In order to be able to compute 3D stereo images in multi-display virtual reality (VR) installations such as the CAVE it is necessary to fulfill some requirements for example:

- The algorithms and effects need to be stereoscopic 3D capable.
- The effects and algorithms need to be useable in multi-display installations.
- The computation and rendering needs to be real-time, otherwise the user would be distracted while viewing the rendered images in 3D since the point of view could not be updated fast enough.
- The algorithms should be applicable without pre-processing in order to apply them on the fly on any data set which needs to be rendered.
- The application is implemented in C++ and uses OpenGL.

3.2. Structure of the Application

The full application consists of multiple components for example the master and the render node which take care about communication between the nodes and the rendering process on the nodes. The CAVE framework, which is used by the nodes to communicate and synchronize, the window management which handles opening a window, closing it and keyboard input by the user. Furthermore, there are the core graphics components like, the mesh loading and drawing, the matrix stack, the shaders, render textures which are used as render targets and the measurement component. Illustration 3.1 shows the components of the application as well as shows which are used by the master or render node.

3.2.1. Master and Render Nodes

The master and render nodes are used to start and run the application, the main function of the master node is to start the render nodes and communicate with them, besides the

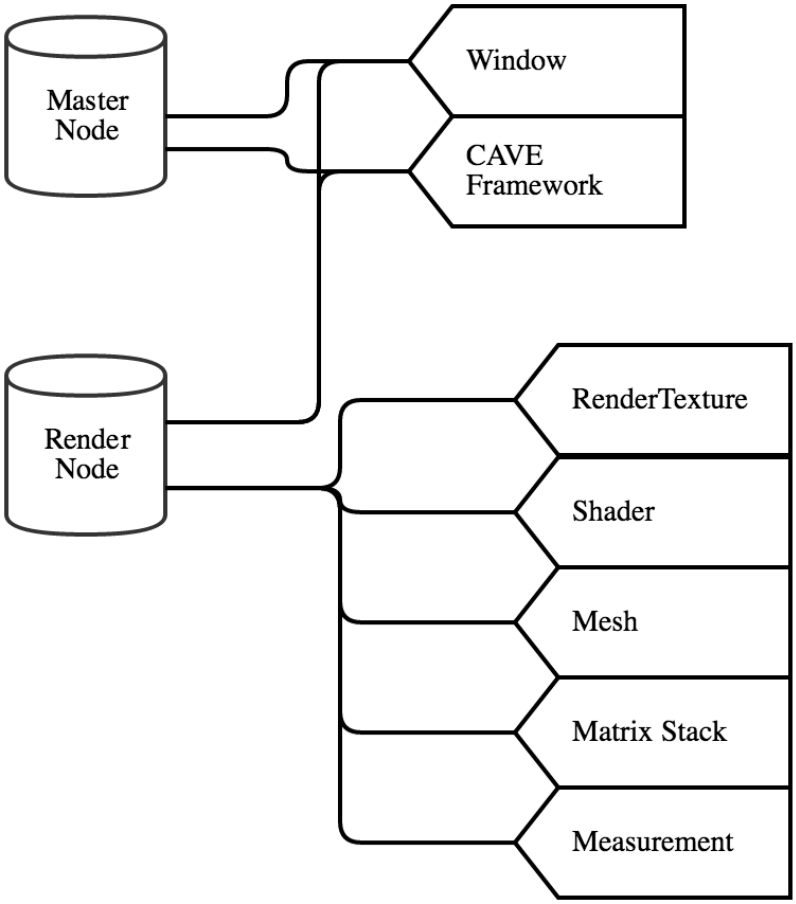


Figure 3.1.: Overall structure of the application

communication the master node is using the user input and forwards these to the render nodes in order to react on keyboard input as well as changes from the tracking system which affect the users view. The render node itself has the purpose of displaying and rendering the 3D scene to the screen it is attached. The render nodes react on changes coming from the master node and updates the 3D scene according to the new information. Both the master and the render nodes implement the CAVE framework which supports the described functionality, see section 3.2.2 for more information about the concept as well as section 4 for more information about the implementation and function in detail.

3.2.2. CAVE Framework

The CAVE Framework is developed by Markus Wiedemann at the Leibniz Supercomputing Centre. The CAVE Framework assists in launching the application by starting the render nodes needed for displaying. Besides launching it also support the communication and synchronization between the render nodes and the master node. The framework is also able to read data from the tracking system and automatically sending this information to the render nodes, as well as user input via keyboard or the wand. The detailed function and

usage of the CAVE framework is explained in chapter **??**.

3.2.3. Window Component

The window part of the application is used to create a new window, display this window and closing it, besides that it initialized everything necessary to render 3D scenes with OpenGL, for example starting the render loop and leaving it. Furthermore, the component reacts on user input via keyboard and forwards the input to the render nodes. Details about the window implementation can be found in section 4.2.4.

3.2.4. Mesh Component

The mesh component takes care about loading meshes from the file system, as well as drawing them. The mesh component is able to load file formats in obj file format and the attached material information as mtl files. The mesh component parses the information from the obj file to OpenGL buffers in order to use them at a later point for rendering, besides that also the materials are stored and the textures are loaded and stored for the later use with OpenGL. For further details about implementation see section 4.2.5.

3.2.5. Pipeline and Matrix Stack

Since OpenGL does not support the build in matrix stack and pipeline anymore in newer versions the pipeline and matrix stack reimplement these functions for the application. In detail it supports scaling, translation and rotation in order to be able to modify the 3D scene again. Besides that, it helps to apply the necessary information of the matrices to the used shader. For example, the pipeline is able to set the view matrix. Section 4.2.6 gives more information about the implementation and usage of the pipeline and matrix stack.

3.2.6. Rendering

The rendering component is used to load and compile the shader programs from the file system since they are stored as GLSL source files. The Rendering component gives detailed feedback if a shader could not be loaded, compiled or linked. Besides I/O and compile functions the Shader component is able to enable and disable a shader, as well as fetching necessary information to set specific input values of the shader so called uniforms. Section 4.3 describes the implementation of the Rendering component. The shader and rendering algorithms are described in more detail in section 3.3.

3.2.7. Render Textures

The render textures are used in the whole process of rendering and are required in order to make it possible to add post-processing effects such as greyscaling. The render texture component consists of two color-buffer render targets and one depth-buffer. The render texture can be created in the width and height required besides that it is possible to set a texture as depth buffer if necessary. Further implementation and usage details can be found in section 4.2.8.

3.3. Rendering Algorithms

This section describes which rendering algorithms are going to be evaluated and implemented. Since there are many algorithms and techniques that could be used it is necessary to limit the amount for more clear comparison and focus on valuable results. All of the following algorithms and techniques where chosen to enhance the rendered scenes for the observer, by removing or adding information.

3.3.1. Phong Shading

The first shading algorithm is the phong shader which is often called per-pixel lighting shader. This shader is based on the phong lightning model and is used to render 3D meshes with correct lightning and in a nearly photorealistic way, such that other algorithms can use the output of these to apply different effects for example greyscale. The phong shader uses the direction of the light and 3D data from the scene or model to compute the amount of light added as well as uses material information to add specular as well as ambient parts. The phong shading model is widely used in computer graphics and a standard way to render 3D meshes. Since it is one of the common approaches the measurement of the phong shading model is not needed, but needs to be measured when it is combined with other effects that work on top. The following formula shows how to compute phong shading, when the light material and vectors are known.

$$I_{ambient} = Light_{ambient} \cdot Material_{ambient} \tag{3.1}$$

$$I_{diffuse} = Light_{intensity} \cdot Material_{diffuse} \cdot (\vec{L} \cdot \vec{N}) \tag{3.2}$$

$$I_{specular} = Light_{intensity} \cdot Material_{specular} \cdot (\vec{E} \cdot \vec{N})^n \tag{3.3}$$

$$I = I_{ambient} + I_{diffuse} + I_{specular} \tag{3.4}$$

3.3.2. Cell/cartoon Shading

Cell shading which is also often called cartoon shading is a 3D effect that generates non-photorealistic renderings from 3D data, by abstracting the scene and removing realistic lightning, and also introducing hard borders during the shading, to achieve an effect that looks like cartoon style. Cartoon shading is sometimes used for visualization of buildings to generate a drawn picture of the 3D model. An example of cell shading can be seen in figure 3.2, which illustrates the hard borders of the shading and also uses simple colors instead of textures. In the equations below the cell shading formula is noted.

$$Intensity = \vec{L} \cdot \vec{N} \tag{3.5}$$

$$I = \begin{cases} color & \text{if } Intensity > 0.95 \\ color * 0.7 & \text{else if } Intensity > 0.5 \\ color * 0.3 & \text{else if } Intensity > 0.05 \\ color * 0.1 & \text{otherwise} \end{cases} \tag{3.6}$$

Figure 3.2.: Cell/cartoon shading

3.3.3. Greyscale and Sephia

Greyscaling and Sephia are two of the effects that come from image processing and manipulation. Both of them apply filters and simplification on the color information. For example, greyscaling combines the different color channels of red, green and blue and transforms these to greyscale information. This effect can be used to greyscale a currently unimportant part of the scene and let the user focus on a specific part of the scene by not removing color information. The same applies to Sephia where the colors are transformed to a more yellow centered appearance than the real one. Also the inverse could be relevant by using sephia or greyscaling on selected objects for highlighting. The following two formulas describe the greyscale and the sephia computation.

$$c = \begin{pmatrix} r \\ g \\ b \end{pmatrix} \bullet \begin{pmatrix} 0.29 \\ 0.59 \\ 0.12 \end{pmatrix} \tag{3.7}$$

$$greyscale = \begin{pmatrix} c \\ c \\ c \end{pmatrix} \tag{3.8}$$

$$i = \begin{pmatrix} r \\ g \\ b \end{pmatrix} \bullet \begin{pmatrix} 0.3 \\ 0.59 \\ 0.11 \end{pmatrix} \tag{3.9}$$

$$sephia = \begin{pmatrix} 0.2 \\ 0.05 \\ 0.00 \end{pmatrix} \cdot (1 - i) + \begin{pmatrix} 1 \\ 0.9 \\ 0.05 \end{pmatrix} \cdot i \tag{3.10}$$

3.3.4. Simple Spot Color or Color Selection

The simple spot color effect is based on the spot color idea from “Non-Photorealistic Rendering with Spot Colour“ by Paul L. Rosin et al. [RL13] which is presented in section

2.3. Since the algorithm from the paper is not efficient enough for real-time rendering the simple spot color effect uses a pre-defined color as filter and applies these to the scene by greyscaling everything that does not match the pre-defined color. This reduces the amount of computation to a minimum since shapes and other things are not taken into acount. The only computation left is to transform the RGB colors to hue and saturation to make it possible to compare the color itself with a defined delta. An effect like this makes sense when visualizing for example heat maps and extracting only the red colors to focus on important parts. Following formula shows how to distinguish if the real color should be chosen or if it should be greyscaled.

$$color = \begin{cases} color & \text{if } color_{selected} = color \\ greyscale(color) & \text{otherwise} \end{cases} \tag{3.11}$$

3.3.5. Focus + Context with Gaussian Filter

The focus and context effect is one of the common approaches in visualization nowadays according to the paper "Illustrative Visualization: New Technology or Useless Tautology?" [RBGV08] from Peter Rautek et al. that is analyzed in section 2.6. Since it is one of the common approaches this effect uses the Gaussian filter to blur out the so called context in order to let the user focus on specific parts of the scene. This focused parts can be for example a selected object or other relevant 3D information of the scene. The focus and context technique is also proven as improvement for visualization.

3.3.6. Edge Detection and Enhancement

Edge detection is one of the NPR methods that adds information by detecting edges. The edge detection and enhancement effect is based on the Sobel and on the Frei Chen edge detection algorithms which are described in the papers "A 3x3 Isotropic Gradient Operator for Image Processing" [SF68] and "Fast Boundary Detection: A Generalization and a New Algorithm" [FC77]. Where both of them are presented in section 2.3.2. Since some 3D models can be better recognized and visualized with more detailed information of the structure, edge enhancement is a good method to be used since it detects also fine lines in an image which can be used to draw edges on top of the image to enhance the structure and the 3D mesh.

3.3.7. Silhouette Drawing

Silhouette drawing is used to enhance non-photorealistic effects of for example cartoon shading by adding extra information to the shape of 3D meshes, also besides enhancement of cartoon shading it is often used to highlight picking of 3D objects by drawing a line around of them. The silhouette is drawn by scaling the actual 3D object bigger than it is and filling the complete content with the silhouette color, for example black. After the bigger object is drawn the actual object is drawn in correct size and correct colors. The result is that a slightly bigger object is drawn that represents the border and the actual object is drawn on top of the silhouette in original size. As previously described this effect can be used for enhancement of cartoon shading as well as highlighting picking which is for example also used in computer games to select units as well as in visualization by highlighting parts in a complex 3D model. An example of silhouette drawing is illustrated in figure 3.3.

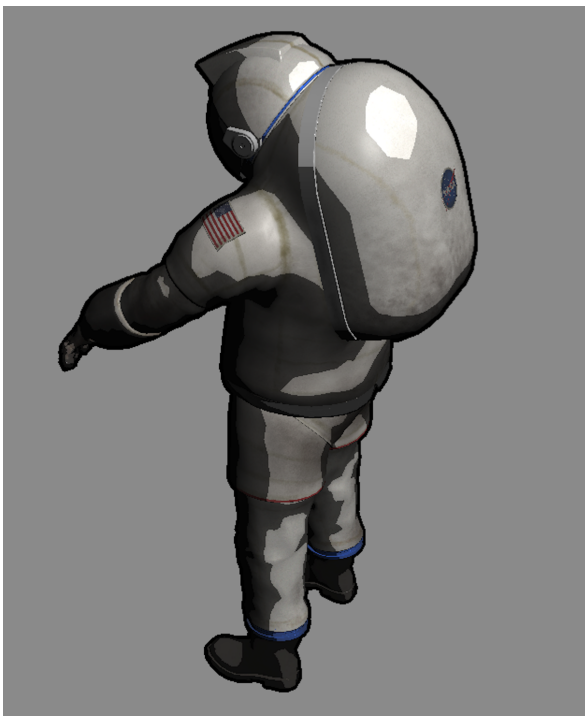


Figure 3.3.: Silhouette drawing around a space men

3.3.8. Combination of Effects

The above described effects can also be combined which can result in more rendering passes that would need more time to render for each frame and would reduce the frames per second. A combination of effects can be for example highlighting the selection by drawing a silhouette around the selected object and also applying the focus and context technique to enhance the selection even more. However, the combination of all effects will not be part of this thesis, since the amount of combinations is too high to measure and compare them all in a fixed amount of time. Only some combinations will be measured since parts of the effects need input from other effects for example all of the post-processing effects like simple spot color, focus and context need as input a rendered scene that comes from either phong lightning or cell shading.

3.4. Application Process

The application works as follows: First of all, when the Master node is starting, it initializes and starts up the render nodes. After that it shows an empty window for using the keyboard input. The master node then continues until it is closed and synchronizes the necessary data such as tracking and input by the user to the render nodes. While the render nodes are starting the necessary data is loaded from disk and initialized such as mesh data and shaders. After the data is loaded the window is created and the render process is started. The render process itself draws the 3D data in a first step into a texture and the texture is used to for rendering in a second pass the post-process effects such as greyscaling or blurring. Also if enabled the outline of the 3D mesh is drawn in the first rendering step. The rendering

process itself is illustrated in figure 3.4. The dashed lines are optional paths in the rendering process. The implementation of the rendering process is described in section 4.3. The render nodes run as long as the master node is running, as soon as the master node is closed the render nodes also quit and the application is fully shut down.

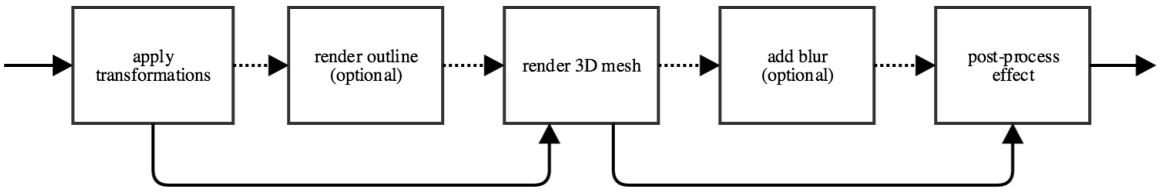


Figure 3.4.: Rendering process of the application

3.5. Measurement

In order to proof if techniques can be applied to the CAVE as virtual reality installation it is necessary to measure the algorithms and techniques. Since the limiting factor for VR in terms of performance is the frame-rate the algorithms should be compared by measurement of the frame-rate in frames per second (fps). The frame-rate needs to be measured with defined scenes which are the same for all algorithms, the pre-defined scenes are presented in section 5.1. For the frame-rate the minimum frames per second, the average and the maximum frames-per second is measured in order to see and compare how stable the frame rate is during the rendering process. The benchmark scenes show different standard models used for benchmarking 3D rendering. All of these models are constantly rotated around the y-axis while measuring constantly the performance in frames per second. Each effect and some common combinations are measured for each model. The results of the measurement are presented in chapter 5.

4. Implementation

The following sections in this chapter give an overview and details about the implementation of the application. It shows in section 4.1 which systems and software where used to implement the software and what the software is based on for example software frameworks. Besides that, section 4.2 describes the components of the software more precicse by explaining the usage of the implemented classes. In section 4.3 the rendering methods are described in a more detailed way by explaining the overall process of rendering in section 4.3 and the rendering effects in more detail. Followed by this section about rendering, the measurement implementation is described in section 4.4.

4.1. System and Software

This section describes details about the systems used to implement for example the workstation which was used to develop the software and the virtual environment where it should run in.

4.1.1. Software

The software is based on the programming language C++ and its graphics core is OpenGl in version 4. For creating a window and OpenGL context GLFW[1] is used in version 3.1.2. GLFW is written in C and has native support for Windows, OS X and Unix-like systems using the X Window System, such as Linux and FreeBSD. Since GLFW has some compile time issues on Suse Enterprise FreeGLUT[2] was implemented as well. Between GLFW and FreeGLUT can be switched through compiler directives. In order to support the loading of 3D models in OBJ Format the Tiny Obj Loader[3] project is used to load and parse the 3D information of the models. Since most of 3D models consist besides 3D information also out of textures the DevIL Library[4] is used to load for example PNG or JPEG files from the file system and bring it to an OpenGL supported format. Because the full matrix stack is no more available in OpenGL version 4 it was necessary to implement this again by using the OpenGL Mathematics (GLM) library[5] which implements vector and matrix operations. To run the application on multiple so called render nodes it was necessary to to implement the Framework from Markus Wiedemann, which is based on GLM, VRPN[6] and Boost[7]. More about the CAVE in section 4.1.3 and the Framework in section 4.2 Since it was necessary to develop and run the application on different systems it was necessary that at least MacOS X and Linux is supported by all mentioned libraries and frameworks.

[1] GLFW OpenGL Library: http://www.glfw.org/
[2] FreeGLUT Project: http://freeglut.sourceforge.net/
[3] Tiny Obj Loader: http://syoyo.github.io/tinyobjloader/
[4] DevIL Image Library: http://openil.sourceforge.net/
[5] GLM Library: http://glm.g-truc.net/
[6] VRPN Project: https://github.com/vrpn/vrpn/wiki
[7] Boost Library: http://www.boost.org/

4.1.2. Development System

In order to test and develop the application the development system was a MacBook Pro 15 inch from Mid 2014. The system was running Mac OS X El Capitan. Build in it has 16 GB RAM, an Intel Core i7 with 2,5 GHz and also two graphic cards the first is an Intel Iris Pro graphics card and the second more powerful is a NVIDIA GeForce GT 750M. The Development System was running IntelliJ AppCode[8] as IDE and CMake[9]CAVE as under laying build-system.

4.1.3. Production System

The production system was the 5-sided-projection installation at the Leibniz Supercomputing Centre in Garching Germany which is based on the Audio Visual Experience Automatic Virtual Environment (CAVE) concept [CNSD+92] from Carolina Cruz-Neira. The CAVE compute system is an SGI Altix XE500 cluster out of 12 nodes. Each of the nodes has two Intel 6core Xeon with 3.06GHz, 48GB RAM and NVIDIA Quadro 6000 graphics card. Besides that, the CAVE has 5 screens formed to a cube with each 2.7m x 2.7m size. Each of the screens is projected by two Christie DLP-Projectors with each full hd resolution. On top of that the CAVE has an ART TrackPack4 tracking system. Ten out of the 12 nodes are used to compute and display where each of them is connected to one of the projectors. The other two nodes are used control the so called render-nodes. The production system can run Suse Linux Enterprise or Microsoft Windows if necessary. To obtain the 3D effect an active stereo system with shutter glasses is used.

4.2. Components

This section should give detailed information about the different components of the implemented application. It also should give detailed information about each component itself how it is implemented what it does and what problems can occur. Figure 4.1 illustrate an overview about the components of the Application. In detail the CAVE framework is described in section 4.2.1, the Window component is described in section 4.2.4, besides these two components which are used by the render and master node the RenderTexture component is explained in 4.2.8. Furthermore, Shader is explained in section 4.2.7 as well as the Mesh component in section 4.2.5 and the Matrix stack in section 4.2.6

4.2.1. CAVE Framework

The CAVE Framework is developed by Markus Wiedemann and is used to render in a synchronized manner on multiple machines in parallel. In detail the Application implements the CAVE framework in two applications. The first application is the so called Master application. The second application is the so called render node. First of all, the Master application has the purpose of reading the configurations of the environment the application should run in this can be either the development system or the CAVE itself as production system.
The configuration defines how many render nodes are used in the environment and where the

[8]IntelliJ AppCode: https://www.jetbrains.com/objc/
[9]CMake: https://cmake.org/

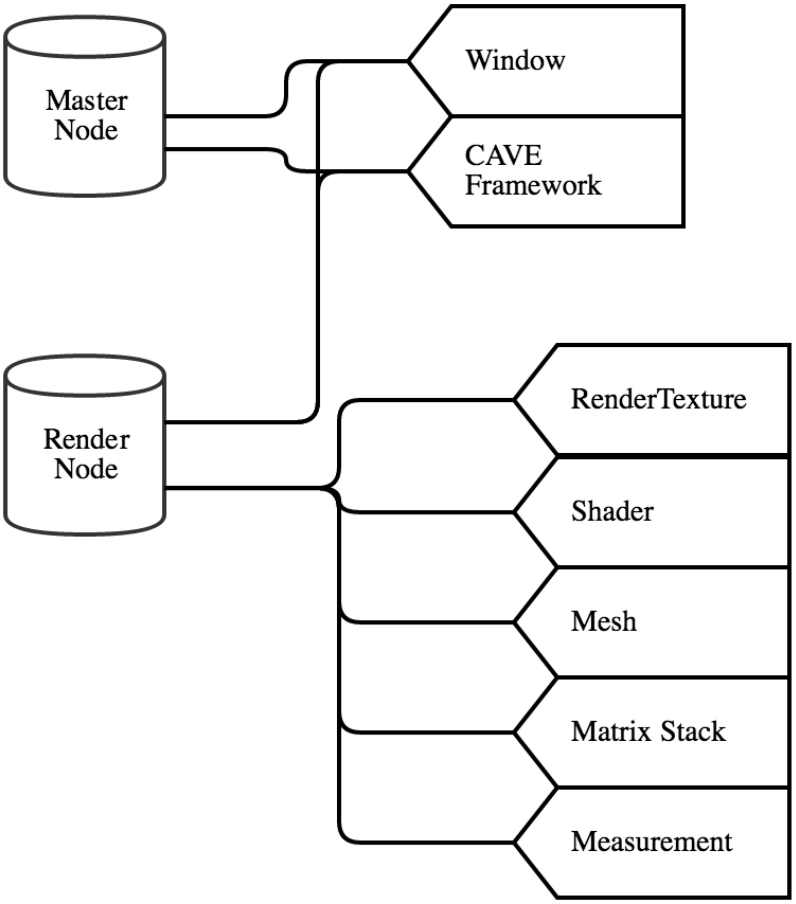


Figure 4.1.: Components of Master and Render node

output of their displays is shown in the system for example in the CAVE that could be the front facing wall. Besides display information the configuration also provides IP addresses of the target system for communication. After the configuration is read the Master application connects to the different machines which are used as render nodes and starts the render node application. If this is done the Master is used to take user input from Mouse, Keyboard or the tracking system and distributes this to the render nodes. Besides user interaction the master also synchronizes the render nodes in order to prevent flickering of the scene.
The second part of the application which runs on the render nodes is used to display the scene from the view of the render nodes specific display for example everything which is shown on the left side of the CAVE. When the render node is started the first thing it does is to load the configuration and fetch its specific view information this is necessary in order to know which viewpoint is to render.
After that the render node connects to the master and waits until all render nodes are ready. If all of them are ready they start with rendering and displaying the scene. In order to get the viewport of the user to project the scene correctly the render node offers matrices for the view and projection matrix as well as the user transformation. With this information it is possible to render the scene correctly for the human eyes. After each frame the render

nodes are synchronizing their frame with a blocking function that functions like a barrier which is released if all render nodes have reached it. Figure 4.2 illustrates how the master and render nodes work together. Requirements for running the application with the CAVE

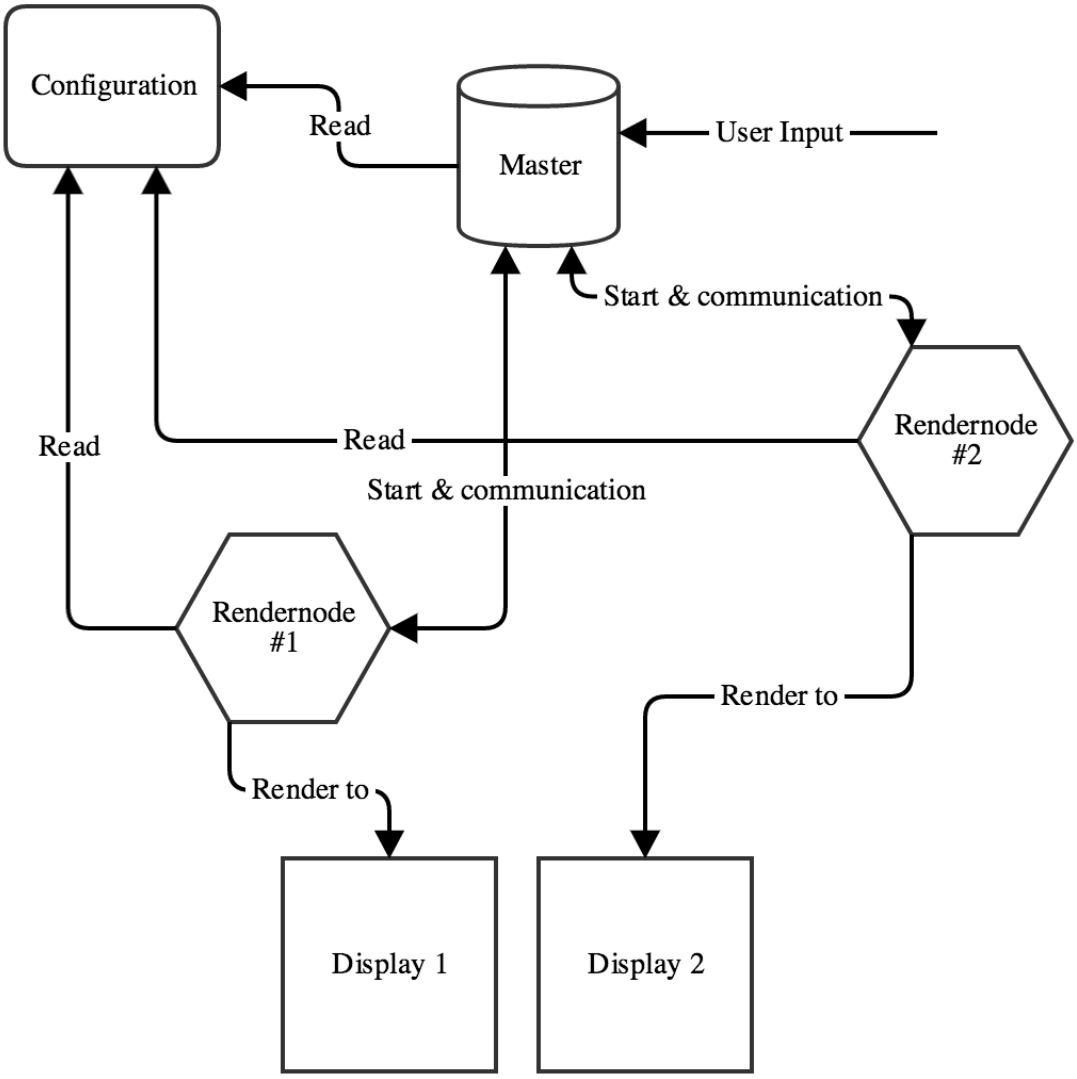


Figure 4.2.: Master application and Render nodes

framework is that the master can reach all nodes via network and that a broadcast network address exists through which the communication can be done.

User Input Via Wand

The CAVE framework provides user input inside the CAVE by tracking a so called wand. The wand is a device the user holds in his hands and consists usually of multiple buttons to press and a joystick. The tracking system receives the tracking information and also if a button was pressed. In the CAVE framework the master implements callbacks for this types

of events and uses these events to process the user input from the wand. During processing the information about the position and button states etc. is send to the render nodes. The render nodes itself can retrieve this information if necessary and also combine for example the joystick input information with the scene to make translation of the rendered scene possible.

4.2.2. Master Node

The master node is used to start the render nodes, control the render nodes and take user input which is then send to the render nodes. Besides that, the master node is used to provide a synchronized time for all render nodes. First of all, when the master node is started a new window is created with the `Window` class (see section 4.2.4 for more details. After the window is created the display function is set which always renders an empty window and updates the synced time for the render nodes.
Next step after the display function is to attach the keyboard callback to the window. The keyboard callback is used to react on key presses from the user for example change the render mode etc. For each of the assigned actions per key the action is also sent to the render nodes that all of them can switch at the same time to a new render mode if necessary. When all of the window related functions etc. have been initialized the master node is constructed with the help of the CAVE framework. The master node is created with the CAVE specific configuration file.
After creating the master node, the `init()` function of the master node is called in order to start all render nodes. If the initialization and start of the render nodes was successful several sync objects are added to the master node which can be used for synchronizing the time user input etc. and the sync process is started. If everything went well the main loop of the master node which calls the display function is started and runs as long as the window is not closed.

4.2.3. Render Node

The purpose of the render node is to render the 3D scene from its particular viewpoint. In order to syncronize with other render nodes each render node implements the render node from the CAVE framework. The node also receives data from the master node in order to react on user input or changes of the user position and viewport. The render node also implements all rendering logic starting by loading meshes (section 4.2.5) up to non-photorealistic rendering techniques. After the render node is started the configuration is read and a full screen window is created by using the sizes from the configuration. After the window is created and the display function is set, the actual render node from the CAVE framework is created and initialized.
As next step the texture rendering is initialized by creating different instances of the `RenderTexture` (section 4.2.8). The difference in the instances of the classes is that the render node can render for a single non-stereo display, a 3D TV by rendering to the display half for left and half for right eye, and also full stereo to the display. Therefore, three textures are needed one full screen texture and two texture half the width of the screen for each eye. While the texture rendering is initialized also all 2D related non-photorealistic rendering shaders are loaded with help of the `Shader` class (section 4.2.7).
In order to render the textures later on the screen using the effect shaders a vertex buffer is

initialized which consists of a square filling the screen. After these steps the DevIL library is initialized to provide image loading in the render node. Since the application also supports 3D shader effects like toon shading also the 3D shaders are now loaded. When all shaders are loaded the 3D models are being loaded with the `Mesh` class (section 4.2.5). Finally, before starting the sync process and the main loop also the necessary sync objects are added to the render node. The render node is running the main loop as long as it is having a connection to the master node, if the master node is shutting down also the render node is shut down.

4.2.4. Window Creation and Keyboard Input

The application supports window creation with GLFW and FreeGLUT through the Window class. To provide both ways the implementation of the window class can be changed with compiler definitions. The window class can be used to create the window by calling `Window::createWindow(int width, int height, std::string windowTitle, int *pargc, char **argv, bool fullscreen)`, which opens a new OpenGL window with the specified width, height and title. The method also supports opening a full screen window if necessary. First of all, the method initializes either GLFW or FreeGLUT and opens a new window, also for both the OpenGL version is set to 4.1 to provide the newest GLSL shader version which is necessary for the implemented shaders. After the window and the OpenGL context is created glew is initialized as well to provide the OpenGL extensions. If nothing went wrong during this process it returns `true` which means the window is ready for use otherwise it returns `false`.
The window class provides `Window::setDisplayFunc(void (*func)())` to set the function which is called for rendering the scene. For FreeGLUT this function is directly handed over, for GLFW it is saved and later used. To start the render loop `Window::startMainLoop()` can be called which simply starts the render loop of FreeGLUT or starts the GLFW loop. This function blocks as long as something is rendered and the render loop should run. `Window::close()` the render loop can be exit and the window is closed. To exit and close OpenGL in a clean way `Window::terminate()` is implemented to correctly exit and close the window and OpenGL context. During the render process it is necessary to swap the buffers of the used window this can be done with `Window::swapBuffers()`.
In order to provide user input via keyboard to for example change the render mode or shader used the window class implements its own keyboard callback supporting the keys on the keyboard. The keyboard callback can be set and assigned to the window by calling `Window::setKeyCallBack( Window *instance, void (*func)(int))` which takes a function pointer to the method which should be called when a button on the keyboard was pressed. The called function has only one parameter which reflects the pressed key. The internal implementation maps either the keys detected by GLFW or the ones detected by FreeGLUT to the ones provided by the window class.

4.2.5. Mesh Loading and Drawing

Mesh Loading

In order to provide a way to easily load 3D models and meshes for rendering a Mesh loading class was implemented. Meshes themselves consist of different data which is needed for rendering. The important ones for this application are the vertices of the mesh, the normals, the texture coords and the texture itself. Besides this information also the material information

is read which includes the texture information. The mesh loading supports only 3D models in OBJ file format since it can be exported from nearly every 3D software. The signature of the Mesh class constructor is as follows: `Mesh::Mesh(const char * fileName, const char *matDir)`, the first parameter is the path of the 3D model to load and the second parameter indicates the folder where the materials can be found since textures and so on are often separated into subdirectories.
First of all when initializing the 3D mesh the parameters are hand over to the Tiny Obj Loader which loads the file and parses the content of it. The Tiny Obj Loader returns two vectors of information, the first one is a vector of type `tinyobj::shape_t` which consists of all 3D information for one shape. The second vector consists of type `tinyobj::material_t` which holds all relevant material information such as the texture, the ambient color, the diffuse color and so on. Next step during initialization is to iterate over all materials in vector two and load the diffuse texture into an OpenGL texture. This is done with the help of the DevIL image library. First of all, the texture is loaded with origin in the lower left corner (OpenGL standard), if the image is loaded successfully the image information is converted to RGBA colors with unsigned byte data type. Texture filtering is set to `GL_NEAREST` and texture wrapping is set to `GL_CLAMP_TO_EDGE` for both axis (s and t).
After the OpenGL texture is generated and configured the image data which was loaded from DevIL is loaded into the OpenGL texture and the mip maps are generated. The final step in this process is to save the OpenGL texture id in the struct `MaterialBuffer` and put this struct into a vector of `MaterialBuffer`. After all materials have been loaded the shapes of the 3D model are iterated. For each shape the positions, normals and texture coordinates are saved in a vector of `glm::vec4` for the positions and normals and in a vector of `glm::vec2` for the texture coordinates.
When all 3D information was read to the vectors for each of the shapes a separate vertex array object is generated. The vertex array object functions as single reference to a single shape. After the vertex array object is generated for each of the vectors of one shape an array buffer is generated with OpenGL. The array buffers are initialized and filled with the data from the vectors. Each of the array buffers receive a specific vertex attribute pointer according to their data. For example, the array buffer for 3D vertices get the attribute pointer zero, the normal array buffer one and so on. This is very helpful when it comes to drawing the mesh with an shader since shaders in GLSL support location binding for input data, this makes it possible to get rid of specifying for each shader during rendering which data should come to which input variable. Besides the array buffers for vertices, normals and texture coordinates the indices of the shape are copied into an element array buffer. This is an optimization since indexing 3D data is more efficient than saving the 3d vertices for each triangle again. The final step when processing a shape is that all OpenGL ids for all the buffers etc. are saved to a struct with type `ShapeBuffer` which is then put to a vector of the same type. Also the `ShapeBuffer` holds the information which material is used.

Mesh drawing

For drawing the 3D model which was loaded with the Mesh class the function `Mesh::draw(GLuint shaderProgram)` is used. The purpose of this function is to draw the 3D model with the specified OpenGl shader program which comes as parameter. To draw the Mesh, the vector of type `ShapeBuffer` is iterated. For each of the `ShapeBuffer` objects the according material is taken from the vector of type `MaterialBuffer`. With the information

from the MaterialBuffer the texture is set in OpenGL and the texture uniform of the shader program is also set to the texture id.
After the texture has been set the vertex array object of the shape is bound in OpenGL and the triangles are drawn with help of the indices. Finally, the the vertex array object and textures are unbound again to not get in any conflict with the upcoming drawings. If the vertex array objects are not unbound correctly it can happen that different vertex array objects conflict each other which results in rendering issues. Therefore, it is necessary that the buffers etc. are unbound after each usage.

4.2.6. Pipeline and Matrix Stack

Since OpenGL has no matrix stack in the recent versions it was necessary to implement this functionality again with help of the GLM library which provides implementations for vectors and matrices. The Pipeline class provides the same functionality as the OpenGL calls for example `glTranslate` and `glRotate`. In order to provide this functionality, the class holds three vectors of type `glm::mat4`, one for the model matrix, one for the view matrix and one for the projection matrix. The reason for the vectors is to implement the `glPushMatrix` and `glPopMatrix` functionality since C vectors can handle the adding and removing of items. Besides the vector for the different matrices the class holds also three instances of `glm::mat4` for the model view matrix, the model view projection matrix and one instance for the normal matrix. This is done for optimization purpose, that the vertex shader does not need to compute these matrices over and over again for each vertex.
During initialization of the pipeline all vectors and all matrixes become initially an identity matrix set. The pipeline has the functionality to switch the matrix stack used with `Pipeline::matrixMode(int m)`, the method switches the currently used stack to either `MODEL_MATRIX`, `VIEW_MATRIX` or `PROJECTION_MATRIX` this method reflects the function `glMatrixMode` from OpenGL. The method `Pipeline::loadIdentity()` sets the current matrix of the selected matrix stack to an identity matrix like `glLoadIdentity` in OpenGL does. Also to provide rotation, scaling and translation the Pipeline implements `Pipeline::translate(float x,float y,float z)`, `Pipeline::scale(float x,float y,float z)`, and for each axis a rotation function for example `Pipeline::rotateX(float angle)` all of these function multiply the according transformation to the currently selected matrix stack which can be either the view matrix or the model matrix. The projection matrix is not supported for these since it makes no sense to rotate the projection matrix.
Besides the transformations also a complete `Pipeline::multMatrix(glm::mat4 mult)`. To bring back `glPushMatrix` and `glPopMatrix` the pipeline implements the methods `Pipeline::pushMatrix()` and `Pipeline::popMatrix()` which both take the current matrix and put this matrix into the vector again to emulate the matrix stack for `glPushMatrix` or remove the top matrix to emulate `glPopMatrix`. In order to set a correct projection matrix, the pipeline class implements two functions one for orthographic projection and one for perspective projection these are manipulating the projection matrix directly and can be called with `Pipeline::ortho( float left, float right, float bottom, float top, float near, float far)` and `Pipeline::perspective(float angle, float aRatio, float near, float far)`. One of the most important methods is `Pipeline::updateMatrices(unsigned int programId)` which is needed to apply the current matrix stack to a specific shader program, by setting

the relevant uniforms of the vertex shader. Before the function applies the matrices to the shader it checks if something has changed and computes if necessary the normal matrix, model view matrix and the model view projection matrix before these are used by the specified shader program. Beneath the described functionality the pipeline class provides also getter and setter functions for the build in attributes.

4.2.7. Shader

In non-photorealistic rendering shaders are heavily used which means it is necessary to load shader source code and compile this shader code inside the application with OpenGL. In order to make the handling of multiple shaders easier the `Shader` class provides a simple api to load and compile shaders as well as using them. First of all, the constructor accepts c-strings or string objects. The constructor has also two parameters one for the vertex shader source code filename and one for the fragment shader source code which can be seen here: `Shader::Shader(std::string vertexShaderFile, std::string fragmentShaderFile)`. First of all, after creating a new instance of the Shader class, two OpenGL shaders are created with OpenGL one for the vertex and one for the fragment shader. After this operation was successful the source code files of the shaders is read and being attached to the shaders. In the next step the shaders are getting compiled with the attached source code. Since it can happen that shader code is not correct or not compiling the compile status and info log is read from OpenGL and printed if something has gone wrong with the error messages. The final step during initialization are to create a shader program attach the vertex and fragment shader which were compiled before and finally link them together to get the shader program. The initialization process is finished by binding the fragment data output.
For using the newly created shader program the `Shader` class has one function to enable the shader and use it and one function to disable and no more use the shader. To use the shader the function `Shader::useShader()` can be used and for disabling `Shader::unuseShader()`. Also if necessary the shader program can be retrieved by calling `Shader::getProgram()` which simply returns the OpenGL shader program. Besides providing these convenience methods the class also provides an API for getting uniform locations by name since some shaders also have input data that can be set before using them. To get a uniform location the method `Shader::getUniformLocation(const char *uniformName)` returns the uniform id of the shader or prints a message in the console if the uniform could not be found. This likely happens if a uniform was removed during shader code optimization since it was for example not used. This also helps to find errors inside the shader code.

4.2.8. Render Texture

Some of the non-photorealistic rendering effects need to use a method called render to texture. For render to texture it is necessary to generate a texture which can be used as render target during the render process. For simplification of the application the `RenderTexture` class offers a simple way to create textures for rendering in a specific size as well as adding also a depth buffer to the texture if necessary. First of all, if a new object is initialized by calling the constructor `RenderTexture::RenderTexture(int w, int h, bool withDepthTexture)` a new frame buffer is created. A frame buffer can hold zero, one or multiple textures and also hold zero or one depth buffer. This frame buffer is also used as target when using the render to texture method. If the frame buffer is created a 2D texture

is generated with OpenGL which has width `w` and height `h` the color format of the 2D texture is set to `GL_RGB`. Besides RGB as color format texture filtering is set to `GL_NEAREST` for both `GL_TEXTURE_MAG_FILTER` and `GL_TEXTURE_MIN_FILTER` as well as texture wrapping is set to `GL_CLAMP_TO_EDGE` for the s and t axis of the texture. After this first color buffer is created also second one is generated with the same properties in order to support effects that need two color buffers.
As next step if `withDepthTexture` is set to true a second 2D texture is generated which has the format `GL_DEPTH_COMPONENT24` in order to create a 2D depth texture. For the depth texture also filtering and wrapping is set to `GL_NEAREST` and `GL_CLAMP_TO_EDGE` as well. When the depth texture is generated it is attached to the frame buffer which was created before.
If `withDepthTexture` was set to false in order to not use a separate depth texture a new render buffer is generated with type `GL_DEPTH_COMPONENT` and finally attached to the frame buffer. After the depth component of the frame buffer was created the 2D color texture is set as color attachment zero, and the color attachment zero is used as the only draw buffer of the frame buffer. As a final step the status of the frame buffer is checked in order to find problems which can occur during the initialization process of the frame buffer. For using the created frame buffer, color texture, secondary color texture and depth texture the methods `RenderTexture::getFrameBuffer()`, `RenderTexture::getColorBuffer()`, `RenderTexture::getSecondaryColorBuffer()` and `RenderTexture::getDepthBuffer()` return the OpenGL handle to the according texture. Besides that, the method `RenderTexture::isHasDepthTexture()` returns true if a depth texture was used or false if a render buffer was used for the depth component.

4.3. Rendering

In general, the rendering process is cut into five parts (see figure 4.3). First of all, the transformations are applied to the Matrix stack, like translation, scale and rotation, to position the 3D objects correctly and also apply the transformations from the tracking of the user inside the CAVE. After the transformation the first rendering step is done, which is also not mandatory and optional in case that an outline should be drawn around the object. When the outline Rendering is finished the 3D object is rendered in step three which is the middle part of figure 4.3. The third step is a standard rendering of the 3D object which uses either Phong shading or Toon shading depending on which effect should be produced. Both of these rendering steps (two and three) are rendered to a texture with the help of the `RenderTexture` class. After step three the first post-processing effect is applied which is also optional and adds blur to achieve the focus and context effect, the result of this effect is also rendered to a texture to continue with the fifth step. The fifth step is also the last step and final rendering pass which also applies post-processing effects and finally renders to the screen for displaying.

4.3.1. Outline Rendering

In the outline rendering step the outline is rendered by using three rendering passes of the 3D objects in total to generate the outline around the object. First the 3D object is rendered using a shader that can add an offset to the original model and render only the one solid color for example black, which results in a slightly bigger representation of the original 3D object which can be seen in figure 4.4a. The first rendering step also renders only front

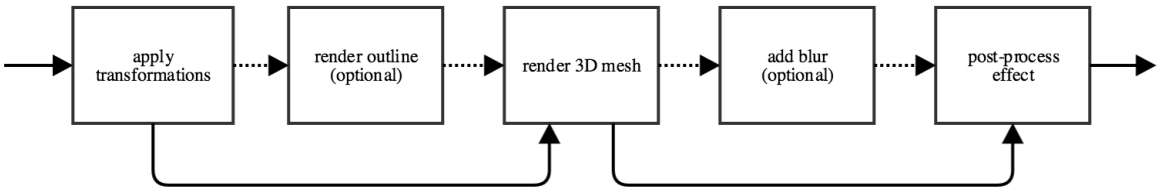
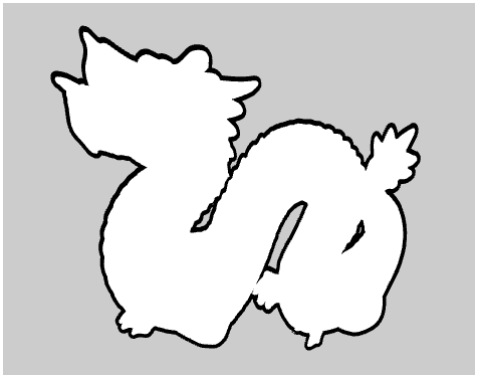


Figure 4.3.: Parts of the rendering process

facing triangles with turned of depth testing. After the first rendering step the same shader is used to render the back faces of the 3D model but this time without any offset, that the 3D object is rendered in the correct size. The second step also uses a different color like white, this results in black line that fits the outside of the 3D object see figure 4.4b. The final step is to render the 3D object in a usual way with enabled depth testing. Result of step three is the final image with an added outline which is illustrated in fig 4.4c.

(a) Result of outline rendering step one



(b) Result of outline rendering step two

(c) Final result of outline rendering

Figure 4.4.: The steps of the outline rendering

To add the offset inside the shader program formula 4.1 is used inside the vertex shader to compute the new vertex out of the old one by extending it in the correct direction using

the normal of the vertex.

$$vertex' = vertex + normal * offset \tag{4.1}$$

4.3.2. Blur Effect

The blur effect which is used in the optional step four of the rendering process is able to blur a texture by using a standard Gauss filter to achieve a Gaussian blur effect. Step four needs in total two rendering passes to generate the blur effect. One pass is used to add blur horizontally and the second one is used to blur in vertical direction. To save additional rendering passes which would be needed to blur multiple objects a mask is used to blur everything needed in one step (horizontal and vertical). Starting in rendering step three a second color buffer is used to generate the mask by adding white at positions that should be blurred and black at positions that should not. The blur shader reads the second color buffer used for the mask and decides depending on the color if the pixel is blurred or not. This optimization is only possible because the blur effect is applied in image space and not in object space. The results of the blur effect are rendered into a texture which is used by the post-processing effects. An illustration showing the blur effect can be seen in figure 5.4.

4.3.3. Post-Processing

The post-processing step consists of different effects that can be applied in this step, the effects are limited to one effect for this step. In the following sections each post-processing effect is described, by explaining the effect itself and the implementation.

Grey scaling

The grey scaling shader used the image data from the texture and produces a grey scaled image only. The implementation of the shader uses the dot product of the rgb color vector and a vector which is used to weight the different color channels see equation 4.2 and get the intensity. The result of the dot product is used for each color channel (rgb), which is the final fragment color.

$$i = \begin{pmatrix} r \\ g \\ b \end{pmatrix} \bullet \begin{pmatrix} 0.29 \\ 0.59 \\ 0.12 \end{pmatrix} \tag{4.2}$$

$$color = \begin{pmatrix} i \\ i \\ i \end{pmatrix} \tag{4.3}$$

Sephia

The sephia effect is similar to the grey scale effect but does more computation to achieve the sephia effect. First of all, the sephia effect also computes the intensity similar to the grey scale effect. Then the intensity is used to interpolate between the two sephia colors which is noted in equation 4.5. The result of this interpolation is used as final fragment color.

$$i = \begin{pmatrix} r \\ g \\ b \end{pmatrix} \bullet \begin{pmatrix} 0.3 \\ 0.59 \\ 0.11 \end{pmatrix} \tag{4.4}$$

$$color = \begin{pmatrix} 0.2 \\ 0.05 \\ 0.00 \end{pmatrix} \cdot (1 - i) + \begin{pmatrix} 1 \\ 0.9 \\ 0.05 \end{pmatrix} \cdot i \tag{4.5}$$

Color Selection

With the color selection shader it is possible to define a selected color in RGB color and render everything expect this color in greyscale, which is helpful to highlight color parts of an image. In general, the shader uses the selected color and checks if the current color is the same. If the colors are the same then it renders the color that was found, if not it uses grey scaling in the same way it was described in section 4.3.3. In detail the shader transforms the selected color from RGB color space to HSV (hue, saturation and value), also the color which comes as input from the texture is transformed to HSV color space. In HSV color space it is easier to detect similar colors, since color (hue) is separated from the saturation. The hue has a range from zero to 360 where zero and 360 are the same color (red). The shader adds a delta to the filter color and checks if the current color is in this delta. If yes it sets the fragment color to the color from the texture, if not it applies grey scaling to the current color and returns it as fragment color. Since the values for hue reach from zero to 360 it is necessary to check while applying the delta that it does not exceed 360 or is below zero by either subtracting 360 or adding 360. An example of the color selection where the selected color is gold can be seen in figure 5.3 from the concept section.

Edge enhancement

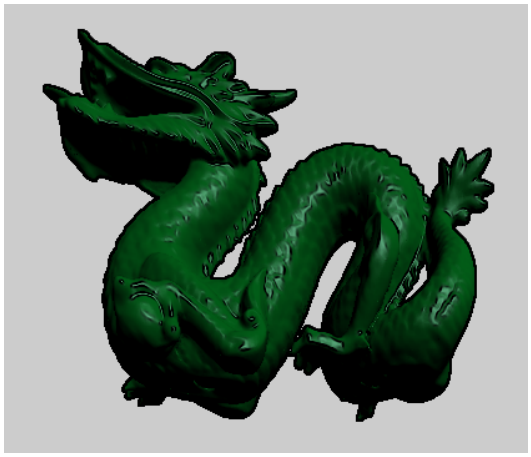
Edge enhancement is used to highlight edges inside the image to gain better visibility of important image information. The edge enhancement consists of two approaches that can be selected. The first is a Sobel edge filter and the second is a Frei Chen edge filter algorithm. More details about the Sobel filter and the Frei Chen filter can be found in section 2.3.2, which describes the scientific papers about both topics. Both of the edge enhancement filters work similar expect the chosen filter algorithm which can be the Sobel filter or the Frei Chen filter. Inside the shader the support for three processing modes is implemented. The first mode renders the plain result of the filter algorithm to the screen, this results in a black image where the edges are drawn in white (figure 4.5a). The second mode is the opposite of processing mode one, it displays the image in white where the edges are drawn in black (figure 4.5b). The third and last processing mode is able to apply the second mode onto the original image (figure 4.5c). This third mode is the mode used by the benchmark scenarios.

4.4. Measurement

In order to be able to measure on each render node the frame rate, the measurement is done by implementing the `FpsLogger` class. The class is able to log each frame with its timestamp. By logging each frame with the timestamp the analysis of the results can be done offline, since

(a) Edge enhancement mode one

(b) Edge enhancement mode two



(c) Edge enhancement mode three

Figure 4.5.: Processing modes of edge enhancement shaders

it is possible to compute the longest time span needed to compute an image as well as the shortest and average. The Measurement uses milliseconds as time unit. The class `FpsLogger` has methods to start (`FpsLogger::startLogging()`), stop (`FpsLogger::stopLogging()`) and reset (`FpsLogger::resetLogging()`) the logging. Besides that the class offers a methods which is called for each frame and saves the frame number plus timestamp in memory (`FpsLogger::newFrame()`). When the measurement is stopped the results can be written from memory to disk as a simple list of frame number and timestamp.

5. Measurements and Results

This chapter lists and analyses the results of the measurements and shows the visual results. Since this thesis consists of different algorithms, these are as well compared using the measurement results. The visual results will be presented as 2D images since 3D images from stereo renderings can not be printed. Section 5.2 will list the measured results and shortly describe from which scenario these results are. In section 5.1 the Benchmark Models and Scenarios are presented in detail. Section 5.3 discusses the results and analysis them.

5.1. Benchmark Models and Scenarios

For benchmarking the effects and the rendering it is necessary to have pre-defined scenarios that can be used to compare different effects and combination of effects. The pre-defined scenarios will be defined in the following section. To see how the effects are performing with different amounts of triangles three different models are used which will be described as well in this section. The amount of triangles ranges from 69.451 triangles up to 1.087.716 triangles per benchmark model and in total 2.289.997 triangles for all three models.

5.1.1. Normal Shading and Toon shading

The scenarios also consist of the normal shading and the toon shading scenarios without any post processing effect, to be able to compare post-processing effects with the standard shading procedures and to be able to see how much computation is needed for the post-processing effects additionally to the standard rendering process.

5.1.2. Greyscaling and Sephia

The scenario for greyscaling and sephia are measured to see how for example color selection performs compared to simple greyscaling and sephia effects, which is needed to see what is needed to for example check the color against the selected color.

5.1.3. Edge Enhancement with Standard Shading

The edge enhancement scenario with standard shading is able to benchmark the combination of rendering the scene with phong shading and in addition to that enhance the edges and structure of the model with edge highlighting. Besides that, the scenario has two variants, the first is with turned off outlines visible in figure 5.1, the second variant has additional outlines.

5.1.4. Edge Enhancement with Toon Shading

The scenario edge enhancement with toon shading is comparable to the scenario edge enhancement with standard shading. The difference is that the shading model is toon shading.

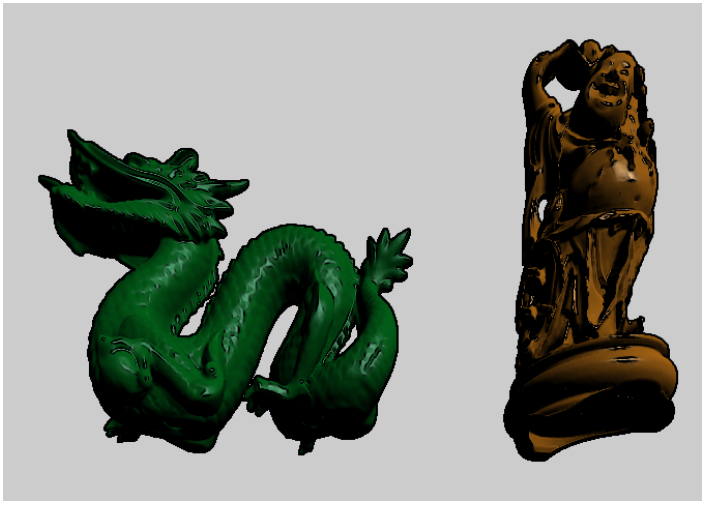


Figure 5.1.: Example enhanced edges with standard shading

The scenario has also two variants, one with extra outline and one without. The combination of toon shading and edge enhancement is visible in figure 5.2.

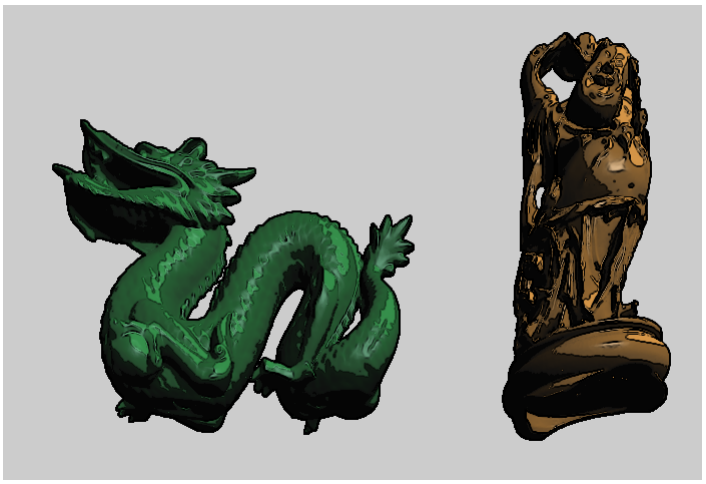


Figure 5.2.: Example enhanced edges with toon shading

5.1.5. Color Selection

The color selection scenario is used to showcase and benchmark the post-processing effect which is able to render the scene in greyscale. The exceptions in this post-processing effects are the parts of the scene which have the selected color. In case of matching color between selected and scene color the scene is rendered normally. Figure 5.3 shows on the left the normal scene and on the right the scene with color selection set to an orange brown color that fits to the color of the Buddha statue.

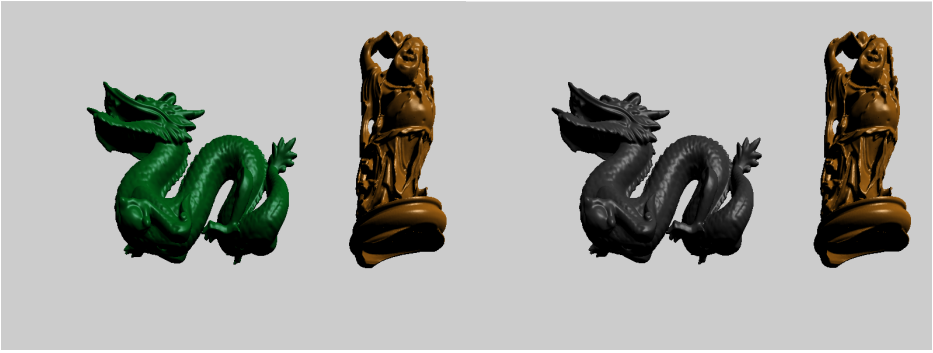


Figure 5.3.: Example of color selection

5.1.6. Focus and Context

For evaluating the focus and context effect the scenario consists of two models, where one is in focus and one is the context. The scenario benchmarks the selective blur of the scene by rendering one object normally and the other blurred. Figure 5.4 shows a rendering where the focus is on the Buddha statue.

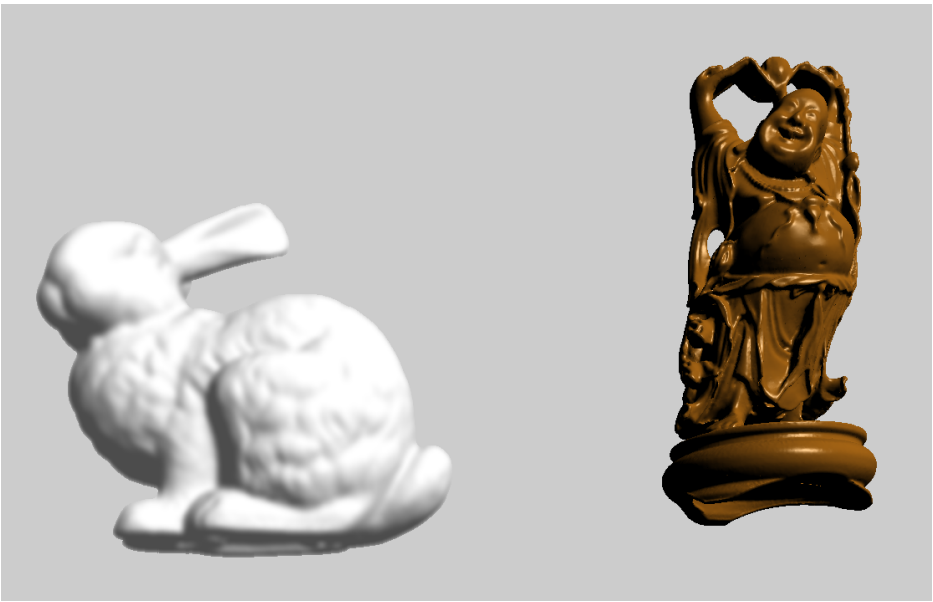


Figure 5.4.: Example scenario of two objects with focus on the Buddha statue

5.1.7. Stanford Bunny

Source	Stanford University Computer Graphics Laboratory
Scanner	Cyberware 3030 MS
Number of scans	10
Size of scan	362,272 points (725,000 triangles)
Size of reconstruction	35947 vertices, 69451 triangles

5.1.8. Stanford Dragon

Source	Stanford University Computer Graphics Laboratory
Scanner	Cyberware 3030 MS + spacetime analysis
Number of scans	70
Size of scan	2,748,318 points (about 5,500,000 triangles)
Size of reconstruction	566,098 vertices, 1,132,830 triangles

5.1.9. Stanford Buddha

Source	Stanford University Computer Graphics Laboratory
Scanner	Cyberware 3030 MS + spacetime analysis
Number of scans	60
Size of scan	4,586,124 points (9,200,000 triangles)
Size of reconstruction	543,652 vertices, 1,087,716 triangles

5.1.10. Limitations

The scope of the thesis will be limited to effects that can work on any kind of 3D data that fit to the OpenGL framework for rendering. All effects that do need pre-computation on the 3D data will not be discussed and evaluated since one of the goals of this thesis is to evaluate plug and play non-photorealistic effects. These plug and play effects need to be generic enough to work on 3D data without pre-conditions, like adding more information out of heavy computations or any other type of pre-processing. The result should be a toolbox that can be used generally for VR and visualization. One more limitation is that the algorithms should support a high number of vertices. The reason for a high count of vertices is that many 3D object that will be visualized have a high resolution and therefore a high number of vertices that need to be displayed. As example for a high resolution 3D objects the models of the Stanford 3D Scanning Repository [The] are used see section 5.1 for more details about the used models, including number of vertices of the 3D scans and 3D models used for OpenGL.

5.2. Measurement Results

This section lists the measurement results and shortly describes them. The results are presented in a table that lists the average values computed over all render nodes. The detailed measurement results can be found in the appendix.

The list of scenarios is as follows:

1. Reference Measurement: Phong shading, no post-processing, Stanford Dragon and Buddha.

2. Reference Measurement: Phong shading, post-processing (render to texture), Stanford Dragon and Buddha.

3. Reference Measurement: Toon shading, no post-processing, Stanford Dragon and Buddha.

4. Reference Measurement: Toon shading, post-processing (render to texture), Stanford Dragon and Buddha.
5. Greyscale Measurement: Phong shading, post-processing (greyscaling), Stanford Dragon and Buddha.
6. Greyscale Measurement: Toon shading, post-processing (greyscaling), Stanford Dragon and Buddha.
7. Color filter Measurement: Phong shading, post-processing (Color filter), Stanford Dragon and Buddha.
8. Color filter Measurement: Toon shading, post-processing (Color filter), Stanford Dragon and Buddha.
9. Sephia Measurement: Phong shading, post-processing (Sephia), Stanford Dragon and Buddha.
10. Sephia Measurement: Toon shading, post-processing (Sephia), Stanford Dragon and Buddha.
11. Edge Enhancement Measurement: Phong shading, post-processing (FreiChen), Stanford Dragon and Buddha.
12. Edge Enhancement: Toon shading, post-processing (FreiChen), Stanford Dragon and Buddha.
13. Edge Enhancement Measurement: Phong shading, post-processing (Sobel), Stanford Dragon and Buddha.
14. Edge Enhancement: Toon shading, post-processing (Sobel), Stanford Dragon and Buddha.
15. Silhouette Measurement: Phong shading, silhouette, post-processing (Sobel), Stanford Dragon and Buddha.
16. Silhouette Enhancement: Toon shading, silhouette, post-processing (Sobel), Stanford Dragon and Buddha.
17. Focus + Context Measurement: Phong shading, post-processing (Blur), Stanford Dragon and Buddha.
18. Focus + Context Enhancement: Toon shading, post-processing (Blur), Stanford Dragon and Buddha.
19. Combination Measurement: Phong shading, silhouette, post-processing (Blur + Edge Enhancement), Stanford Dragon and Buddha.
20. Combination Enhancement: Toon shading, silhouette, post-processing (Blur + Edge Enhancement), Stanford Dragon and Buddha.
21. Combination Measurement: Phong shading, silhouette, post-processing (Blur + Edge Enhancement), Stanford Dragon, Buddha and Bunny.

22. Combination Enhancement: Toon shading, silhouette, post-processing (Blur + Edge Enhancement), Stanford Dragon, Buddha and Bunny.

Szenario	Time per frame in ms			Framerate per second		
	min	max	avg	min	max	avg
1	16,00000	17,20000	16,66641	58,13953	62,50000	60,00094
2	15,80000	17,40000	16,66633	57,47126	63,29114	60,00122
3	15,80000	17,30000	16,66640	57,80347	63,29114	60,00097
4	15,40000	17,80000	16,66643	56,17978	64,93506	60,00084
5	15,60000	17,80000	16,66640	56,17978	64,10256	60,00095
6	15,90000	17,30000	16,66642	57,80347	62,89308	60,00089
7	16,00000	17,20000	16,66639	58,13953	62,50000	60,00098
8	15,80000	17,30000	16,66649	57,80347	63,29114	60,00064
9	16,00000	17,50000	16,66631	57,18955	64,50000	60,00130
10	16,00000	17,50000	16,66631	57,18955	62,50000	60,00132
11	16,00000	17,20000	16,66636	58,13953	62,50000	60,00112
12	15,90000	17,10000	16,66640	58,47953	62,89308	60,00095
13	16,00000	17,10000	16,66640	58,47953	62,50000	60,00096
14	16,00000	17,20000	16,66646	58,13953	62,50000	60,00073
15	32,10000	34,40000	33,33262	29,06977	31,15265	30,00064
16	31,70000	35,20000	33,33278	28,40909	31,54574	30,00049
17	15,80000	17,20000	16,66640	58,13953	63,29114	60,00095
18	16,00000	21,40000	16,66839	46,72897	62,50000	59,99381
19	33,00000	34,00000	33,33263	29,41176	30,30303	30,00063
20	33,00000	34,00000	33,33277	29,41176	30,30303	30,00050
21	32,10000	34,90000	33,33277	28,65330	31,15265	30,00051
22	32,80000	34,10000	33,33274	29,32551	30,48780	30,00053
23	16,00000	17,20000	16,66639	58,13953	62,50000	60,00098
24	15,80000	17,30000	16,66649	57,80347	63,29114	60,00064

Table 5.1.: Average measurement results

5.3. Discussion of Results

From the measurement results in section 5.2 the first thing to notice is that there is no real difference between phong and cartoon shading since the results differ only a bit from each other. The reason for this is that both are using the same rendering procedure but only differ in the used shader, since shaders are optimized for these computations it does not make a huge difference if there is distinguished between the angle or not. Next fact which is nearly not noticeable is between using render to texture or not, both phong and cartoon shading do not differ much when using render to texture or not. This fact was unexpected since an additional rendering pass is needed to render the texture to the screen and show the final result. The only reason that explains this behavior is that the graphics hardware has all needed information in memory and passes these information further to the renderbuffer by using a simple pass through post-processing shader. The same applies to

greyscaling, sephia, color filter, edge enhancement and the blur post-processing shaders, the measurement results of these are all nearly identical to the pass-through shader since they do only a little bit more computations but not so many and those are using optimized math operations that are supported by the graphics hardware. A noticeable drop in the frame rate can be observed as soon as the silhouette rendering is added, the frame rate drops here by approximately 50% when for example comparing the reference values from table 5.1 and row one which lists the phong shading and the values from table 5.1 row fifth-teen which lists the results from silhouette rendering. When analyzing the algorithm that is used for computing the silhouette around the object it is easy to see that two additional rendering passes are needed to generate the silhouette, and each of these rendering passes operates on 3D mesh data unlike the post-processing effects that only operate in 2D image space. However, all of the implemented algorithms and effects work in real-time with the test scene, where the silhouette needs the most computation time and would also be the first effect to drop the frame rate to much when rendering a more complex scene.
Two of the most useful effects are the color selection effect and the focus and context method which both can be used to highlight objects. Both of these could be used for example for picking in an virtual environment by using the effect on the objects that are currently selected. However, the color selection effect does only work on objects that have a solid color and not on complex objects having multiple colors. As a third highlighting effect the silhouette drawing effect could be used, the advantage is that the border can be drawn in any color but the algorithm itself needs a more expensive computation than for example the focus and context effect or the color selection. In addition to the advantage in selection or picking objects the silhouette contributes to the NPR effects. Besides those effects the edge enhancement is able to draw pictures more artificial and let them look more like a painterly drawing, which is one of the goals in NPR and a nice result.

6. Conclusion

In the research topic of virtual reality (VR) and non-photorealistic rendering (NPR) are a lot of findings, describing algorithms about NPR effects from cartoon shading on 3D data as well as edge detection and enhancement on 2D data. However, the research field combining VR and NPR is not that large and it is needed to be researched, since wide range of application are using VR to visualize. Therefore NPR can help visualizing these by helping with perception as well as enhanced user feedback from user input. This thesis had the goal to evaluate NPR effects with the requirements from VR such as real-time capability and multi display support, besides the VR requirements it should be possible to apply the used algorithms on any kind of 3D data that can be processed with OpenGL. Furthermore, it should compare the performance of different rendering techniques by only using standard software frameworks such as OpenGL.
The prototype application implements a software framework to be able to render on multiple displays at a time where all displays can be connected to different computing machines like it is in case of the CAVE which is a audio-visual automatic virtual environment. Furthermore, the application implements different rendering techniques to achieve effects such as grey scaling, color selection, silhouette drawing, edge enhancement, toon shading, phong shading and the focus and context effect.
One of the outcomes is that post-processing effects like grey scaling, color selection, focus and context or sephia are cheap in computation and can be applied easily without loosing much performance. One other result was that the silhouette effect decreases the performance by approximately 50% which was expected lower. In comparison to the post-processing effects which did not decrease the performance a lot the silhouette effect is the most expensive. This did not had much effect on the fluent amount of frames per second (FPS) which is needed for VR in the test scenarios but it could if the complexity of the scene to render will increase.
In addition to the fact of performance some of the effects can be used to enhance picking in VR. For example the silhouette effect is able to highlight a selected object by drawing an silhouette around. The second effect that can enhance picking is the color selection in case objects having a solid color the picking can select the color and the objects with the same color will be highlighted. An other effect is the focus and context effect that is able to direct the focus of the user to different objects. In addition to the focus effect the focus and context algorithm could also be used for picking by blurring all objects that are not selected. Finally effects such as the edge enhancement can help to improve perception by highlighting things and cartoon shading can help to reduce information in the displayed image which finally help to improve perception.

6.1. Future Work

Although this thesis covers standard NPR effects and describes there usage it is still a limited set of possibilities to bring NPR into VR. Future work could consist of adding more effects

in order to improve user perception as well as user input. Also the implementation could be bundled into a software framework to easily apply the current effects in any kind of VR display or environment. One example for porting the effects could be to use a head-mounted display (HMD) instead of using a CAVE. Besides that the application only implemented possible effects and algorithms that fulfill the requirements of VR it is very likely that in the future improved algorithms of effects are researched that could not be applied in the scope of this thesis. It is also very likely that with hardware that is getting better it is also possible to implement those algorithms that are to expensive in computation now. In addition to the further development of the prototype application, the used effects could be implemented in current applications that are used for visualization in order to improve perception and user input methods. Also in adtition to the usage on complete meshes the methods could be applied only on parts of the mesh for example only a door in a room.

Appendix

A. Measurements and Results

Node	Time per frame in ms			Framerate per second		
	min	max	avg	min	max	avg
N01	16,00000	17,00000	16,66641	58,82353	62,50000	60,00094
N02	16,00000	17,00000	16,66641	58,82353	62,50000	60,00094
N03	16,00000	17,00000	16,66641	58,82353	62,50000	60,00094
N04	16,00000	17,00000	16,66641	58,82353	62,50000	60,00094
N05	16,00000	17,00000	16,66641	58,82353	62,50000	60,00094
N06	16,00000	17,00000	16,66641	58,82353	62,50000	60,00094
N07	16,00000	18,00000	16,66641	55,55556	62,50000	60,00094
N08	16,00000	17,00000	16,66641	58,82353	62,50000	60,00094
N09	16,00000	18,00000	16,66641	55,55556	62,50000	60,00094
N10	16,00000	17,00000	16,66641	58,82353	62,50000	60,00094

Table 1.: Reference results for phong shading with buddha and dragon

Node	Time per frame in ms			Framerate per second		
	min	max	avg	min	max	avg
N01	15,00000	18,00000	16,66633	55,55556	66,66667	60,00122
N02	15,00000	18,00000	16,66633	55,55556	66,66667	60,00122
N03	16,00000	17,00000	16,66633	58,82353	62,50000	60,00122
N04	16,00000	17,00000	16,66633	58,82353	62,50000	60,00122
N05	16,00000	18,00000	16,66633	55,55556	62,50000	60,00122
N06	16,00000	17,00000	16,66633	58,82353	62,50000	60,00122
N07	16,00000	17,00000	16,66633	58,82353	62,50000	60,00122
N08	16,00000	17,00000	16,66633	58,82353	62,50000	60,00122
N09	16,00000	17,00000	16,66633	58,82353	62,50000	60,00122
N10	16,00000	17,00000	16,66633	58,82353	62,50000	60,00122

Table 2.: Reference results for phong shading and render to texture with buddha and dragon

Node	Time per frame in ms			Framerate per second		
	min	max	avg	min	max	avg
N01	16,00000	17,00000	16,66640	58,82353	62,50000	60,00097
N02	16,00000	17,00000	16,66640	58,82353	62,50000	60,00097
N03	16,00000	18,00000	16,66640	55,55556	62,50000	60,00097
N04	16,00000	17,00000	16,66640	58,82353	62,50000	60,00097
N05	16,00000	17,00000	16,66640	58,82353	62,50000	60,00097
N06	15,00000	18,00000	16,66640	55,55556	66,66667	60,00097
N07	15,00000	18,00000	16,66640	55,55556	66,66667	60,00097
N08	16,00000	17,00000	16,66640	58,82353	62,50000	60,00097
N09	16,00000	17,00000	16,66640	58,82353	62,50000	60,00097
N10	16,00000	17,00000	16,66640	58,82353	62,50000	60,00097

Table 3.: Reference results for cartoon shading with buddha and dragon

Node	Time per frame in ms			Framerate per second		
	min	max	avg	min	max	avg
N01	16,00000	17,00000	16,66649	58,82353	62,50000	60,00064
N02	16,00000	17,00000	16,66649	58,82353	62,50000	60,00064
N03	16,00000	18,00000	16,66649	55,55556	62,50000	60,00064
N04	13,00000	20,00000	16,66622	50,00000	76,92308	60,00161
N05	16,00000	17,00000	16,66649	58,82353	62,50000	60,00064
N06	15,00000	18,00000	16,66649	55,55556	66,66667	60,00064
N07	16,00000	17,00000	16,66649	58,82353	62,50000	60,00064
N08	16,00000	17,00000	16,66649	58,82353	62,50000	60,00064
N09	14,00000	20,00000	16,66622	50,00000	71,42857	60,00161
N10	16,00000	17,00000	16,66649	58,82353	62,50000	60,00064

Table 4.: Reference results for cartoon shading and render to texture with buddha and dragon

Node	Time per frame in ms			Framerate per second		
	min	max	avg	min	max	avg
N01	16,00000	18,00000	16,66640	55,55556	62,50000	60,00095
N02	16,00000	18,00000	16,66640	55,55556	62,50000	60,00095
N03	15,00000	18,00000	16,66640	55,55556	66,66667	60,00095
N04	16,00000	17,00000	16,66640	58,82353	62,50000	60,00095
N05	16,00000	17,00000	16,66640	58,82353	62,50000	60,00095
N06	15,00000	19,00000	16,66640	52,63158	66,66667	60,00095
N07	16,00000	17,00000	16,66640	58,82353	62,50000	60,00095
N08	16,00000	18,00000	16,66640	55,55556	62,50000	60,00095
N09	15,00000	18,00000	16,66640	55,55556	66,66667	60,00095
N10	15,00000	18,00000	16,66640	55,55556	66,66667	60,00095

Table 5.: Greyscale post-processing using phong shading and rendering the buddha and dragon

Node	Time per frame in ms			Framerate per second		
	min	max	avg	min	max	avg
N01	16,00000	17,00000	16,66640	58,82353	62,50000	60,00095
N02	16,00000	17,00000	16,66640	58,82353	62,50000	60,00095
N03	16,00000	18,00000	16,66640	55,55556	62,50000	60,00095
N04	16,00000	17,00000	16,66649	58,82353	62,50000	60,00064
N05	16,00000	17,00000	16,66640	58,82353	62,50000	60,00095
N06	16,00000	17,00000	16,66640	58,82353	62,50000	60,00095
N07	16,00000	17,00000	16,66640	58,82353	62,50000	60,00095
N08	16,00000	17,00000	16,66640	58,82353	62,50000	60,00095
N09	16,00000	18,00000	16,66640	55,55556	62,50000	60,00095
N10	15,00000	18,00000	16,66649	55,55556	66,66667	60,00064

Table 6.: Greyscale post-processing using cartoon shading and rendering the buddha and dragon

Node	Time per frame in ms			Framerate per second		
	min	max	avg	min	max	avg
N01	16,00000	17,00000	16,66639	58,82353	62,50000	60,00098
N02	16,00000	17,00000	16,66639	58,82353	62,50000	60,00098
N03	16,00000	18,00000	16,66639	55,55556	62,50000	60,00098
N04	16,00000	17,00000	16,66639	58,82353	62,50000	60,00098
N05	16,00000	17,00000	16,66639	58,82353	62,50000	60,00098
N06	16,00000	17,00000	16,66639	58,82353	62,50000	60,00098
N07	16,00000	17,00000	16,66639	58,82353	62,50000	60,00098
N08	16,00000	17,00000	16,66639	58,82353	62,50000	60,00098
N09	16,00000	17,00000	16,66639	58,82353	62,50000	60,00098
N10	16,00000	18,00000	16,66639	55,55556	62,50000	60,00098

Table 7.: Colorfilter post-processing using phong shading and rendering the buddha and dragon

Node	Time per frame in ms			Framerate per second		
	min	max	avg	min	max	avg
N01	16,00000	17,00000	16,66649	58,82353	62,50000	60,00064
N02	16,00000	17,00000	16,66649	58,82353	62,50000	60,00064
N03	16,00000	17,00000	16,66649	58,82353	62,50000	60,00064
N04	16,00000	17,00000	16,66649	58,82353	62,50000	60,00064
N05	16,00000	17,00000	16,66649	58,82353	62,50000	60,00064
N06	16,00000	17,00000	16,66649	58,82353	62,50000	60,00064
N07	16,00000	17,00000	16,66649	58,82353	62,50000	60,00064
N08	16,00000	17,00000	16,66649	58,82353	62,50000	60,00064
N09	14,00000	20,00000	16,66649	50,00000	71,42857	60,00064
N10	16,00000	17,00000	16,66649	58,82353	62,50000	60,00064

Table 8.: Colorfilter post-processing using cartoon shading and rendering the buddha and dragon

Node	Time per frame in ms			Framerate per second		
	min	max	avg	min	max	avg
N01	16,00000	17,00000	16,66631	58,82353	62,50000	60,00130
N02	16,00000	18,00000	16,66631	55,55556	62,50000	60,00130
N03	16,00000	18,00000	16,66631	55,55556	62,50000	60,00130
N04	16,00000	17,00000	16,66631	58,82353	62,50000	60,00130
N05	16,00000	17,00000	16,66631	58,82353	62,50000	60,00130
N06	16,00000	18,00000	16,66631	55,55556	62,50000	60,00130
N07	16,00000	17,00000	16,66631	58,82353	62,50000	60,00130
N08	16,00000	18,00000	16,66631	55,55556	62,50000	60,00130
N09	16,00000	18,00000	16,66631	55,55556	62,50000	60,00130
N10	16,00000	17,00000	16,66631	58,82353	62,50000	60,00130

Table 9.: Sephia post-processing using phong shading and rendering the buddha and dragon

Node	Time per frame in ms			Framerate per second		
	min	max	avg	min	max	avg
N01	16,00000	18,00000	16,66630	55,55556	62,50000	60,00132
N02	16,00000	17,00000	16,66630	58,82353	62,50000	60,00132
N03	16,00000	17,00000	16,66630	58,82353	62,50000	60,00132
N04	16,00000	17,00000	16,66630	58,82353	62,50000	60,00132
N05	16,00000	18,00000	16,66630	55,55556	62,50000	60,00132
N06	16,00000	18,00000	16,66630	55,55556	62,50000	60,00132
N07	16,00000	18,00000	16,66630	55,55556	62,50000	60,00132
N08	16,00000	17,00000	16,66630	58,82353	62,50000	60,00132
N09	16,00000	17,00000	16,66630	58,82353	62,50000	60,00132
N10	16,00000	18,00000	16,66630	55,55556	62,50000	60,00132

Table 10.: Sephia post-processing using cartoon shading and rendering the buddha and dragon

Node	Time per frame in ms			Framerate per second		
	min	max	avg	min	max	avg
N01	16,00000	17,00000	16,66649	58,82353	62,50000	60,00064
N02	16,00000	17,00000	16,66622	58,82353	62,50000	60,00159
N03	16,00000	18,00000	16,66622	55,55556	62,50000	60,00159
N04	16,00000	17,00000	16,66649	58,82353	62,50000	60,00064
N05	16,00000	17,00000	16,66649	58,82353	62,50000	60,00064
N06	16,00000	17,00000	16,66622	58,82353	62,50000	60,00159
N07	16,00000	17,00000	16,66622	58,82353	62,50000	60,00159
N08	16,00000	17,00000	16,66649	58,82353	62,50000	60,00064
N09	16,00000	18,00000	16,66622	55,55556	62,50000	60,00159
N10	16,00000	17,00000	16,66649	58,82353	62,50000	60,00064

Table 11.: Edge enhancement with FreiChen edge detector using phong shading and rendering the buddha and dragon

Node	Time per frame in ms			Framerate per second		
	min	max	avg	min	max	avg
N01	16,00000	17,00000	16,66640	58,82353	62,50000	60,00095
N02	16,00000	17,00000	16,66640	58,82353	62,50000	60,00095
N03	16,00000	17,00000	16,66640	58,82353	62,50000	60,00095
N04	16,00000	17,00000	16,66640	58,82353	62,50000	60,00095
N05	16,00000	17,00000	16,66640	58,82353	62,50000	60,00095
N06	16,00000	17,00000	16,66640	58,82353	62,50000	60,00095
N07	15,00000	18,00000	16,66640	55,55556	66,66667	60,00095
N08	16,00000	17,00000	16,66640	58,82353	62,50000	60,00095
N09	16,00000	17,00000	16,66640	58,82353	62,50000	60,00095
N10	16,00000	17,00000	16,66640	58,82353	62,50000	60,00095

Table 12.: Edge enhancement with FreiChen edge detector using cartoon shading and rendering the buddha and dragon

Node	Time per frame in ms			Framerate per second		
	min	max	avg	min	max	avg
N01	16,00000	17,00000	16,66640	58,82353	62,50000	60,00096
N02	16,00000	17,00000	16,66640	58,82353	62,50000	60,00096
N03	16,00000	17,00000	16,66640	58,82353	62,50000	60,00096
N04	16,00000	17,00000	16,66640	58,82353	62,50000	60,00096
N05	16,00000	17,00000	16,66640	58,82353	62,50000	60,00096
N06	16,00000	17,00000	16,66640	58,82353	62,50000	60,00096
N07	16,00000	17,00000	16,66640	58,82353	62,50000	60,00096
N08	16,00000	17,00000	16,66640	58,82353	62,50000	60,00096
N09	16,00000	18,00000	16,66640	55,55556	62,50000	60,00096
N10	16,00000	17,00000	16,66640	58,82353	62,50000	60,00096

Table 13.: Edge enhancement with Sobel edge detector using phong shading and rendering the buddha and dragon

Node	Time per frame in ms			Framerate per second		
	min	max	avg	min	max	avg
N01	16,00000	17,00000	16,66623	58,82353	62,50000	60,00159
N02	16,00000	17,00000	16,66649	58,82353	62,50000	60,00064
N03	16,00000	18,00000	16,66649	55,55556	62,50000	60,00064
N04	16,00000	17,00000	16,66649	58,82353	62,50000	60,00064
N05	16,00000	17,00000	16,66649	58,82353	62,50000	60,00064
N06	16,00000	17,00000	16,66649	58,82353	62,50000	60,00064
N07	16,00000	17,00000	16,66649	58,82353	62,50000	60,00064
N08	16,00000	17,00000	16,66649	58,82353	62,50000	60,00064
N09	16,00000	18,00000	16,66649	55,55556	62,50000	60,00064
N10	16,00000	17,00000	16,66649	58,82353	62,50000	60,00064

Table 14.: Edge enhancement with Sobel edge detector using cartoon shading and rendering the buddha and dragon

Node	Time per frame in ms			Framerate per second		
	min	max	avg	min	max	avg
N01	33,00000	34,00000	33,33262	29,41176	30,30303	30,00064
N02	32,00000	34,00000	33,33262	29,41176	31,25000	30,00064
N03	32,00000	34,00000	33,33262	29,41176	31,25000	30,00064
N04	33,00000	34,00000	33,33262	29,41176	30,30303	30,00064
N05	32,00000	34,00000	33,33262	29,41176	31,25000	30,00064
N06	32,00000	34,00000	33,33262	29,41176	31,25000	30,00064
N07	33,00000	34,00000	33,33262	29,41176	30,30303	30,00064
N08	29,00000	38,00000	33,33262	26,31579	34,48276	30,00064
N09	32,00000	34,00000	33,33262	29,41176	31,25000	30,00064
N10	33,00000	34,00000	33,33262	29,41176	30,30303	30,00064

Table 15.: Silhouette drawing with edge enhancement using phong shading and rendering the buddha and dragon

Node	Time per frame in ms			Framerate per second		
	min	max	avg	min	max	avg
N01	30,00000	37,00000	33,33282	27,02703	33,33333	30,00046
N02	32,00000	35,00000	33,33282	28,57143	31,25000	30,00046
N03	33,00000	34,00000	33,33265	29,41176	30,30303	30,00062
N04	33,00000	34,00000	33,33265	29,41176	30,30303	30,00062
N05	32,00000	34,00000	33,33282	29,41176	31,25000	30,00046
N06	32,00000	35,00000	33,33282	28,57143	31,25000	30,00046
N07	33,00000	34,00000	33,33282	29,41176	30,30303	30,00046
N08	30,00000	37,00000	33,33282	27,02703	33,33333	30,00046
N09	32,00000	35,00000	33,33282	28,57143	31,25000	30,00046
N10	30,00000	37,00000	33,33282	27,02703	33,33333	30,00046

Table 16.: Silhouette drawing with edge enhancement using cartoon shading and rendering the buddha and dragon

Node	Time per frame in ms			Framerate per second		
	min	max	avg	min	max	avg
N01	16,00000	17,00000	16,66640	58,82353	62,50000	60,00095
N02	16,00000	17,00000	16,66640	58,82353	62,50000	60,00095
N03	15,00000	18,00000	16,66640	55,55556	66,66667	60,00095
N04	16,00000	17,00000	16,66640	58,82353	62,50000	60,00095
N05	16,00000	17,00000	16,66640	58,82353	62,50000	60,00095
N06	16,00000	17,00000	16,66640	58,82353	62,50000	60,00095
N07	16,00000	17,00000	16,66640	58,82353	62,50000	60,00095
N08	16,00000	17,00000	16,66640	58,82353	62,50000	60,00095
N09	15,00000	18,00000	16,66640	55,55556	66,66667	60,00095
N10	16,00000	17,00000	16,66640	58,82353	62,50000	60,00095

Table 17.: Focus and context effect using phong shading and rendering the buddha and dragon

Node	Time per frame in ms			Framerate per second		
	min	max	avg	min	max	avg
N01	16,00000	26,00000	16,67046	38,46154	62,50000	59,98635
N02	16,00000	17,00000	16,66631	58,82353	62,50000	60,00127
N03	16,00000	25,00000	16,67046	40,00000	62,50000	59,98635
N04	16,00000	17,00000	16,66631	58,82353	62,50000	60,00127
N05	16,00000	26,00000	16,67046	38,46154	62,50000	59,98635
N06	16,00000	17,00000	16,66631	58,82353	62,50000	60,00127
N07	16,00000	17,00000	16,66631	58,82353	62,50000	60,00127
N08	16,00000	26,00000	16,67046	38,46154	62,50000	59,98635
N09	16,00000	17,00000	16,66631	58,82353	62,50000	60,00127
N10	16,00000	26,00000	16,67046	38,46154	62,50000	59,98635

Table 18.: Focus and context effect using cartoon shading and rendering the buddha and dragon

Node	Time per frame in ms			Framerate per second		
	min	max	avg	min	max	avg
N01	33,00000	34,00000	33,33263	29,41176	30,30303	30,00063
N02	33,00000	34,00000	33,33263	29,41176	30,30303	30,00063
N03	33,00000	34,00000	33,33263	29,41176	30,30303	30,00063
N04	33,00000	34,00000	33,33263	29,41176	30,30303	30,00063
N05	33,00000	34,00000	33,33263	29,41176	30,30303	30,00063
N06	33,00000	34,00000	33,33263	29,41176	30,30303	30,00063
N07	33,00000	34,00000	33,33263	29,41176	30,30303	30,00063
N08	33,00000	34,00000	33,33263	29,41176	30,30303	30,00063
N09	33,00000	34,00000	33,33263	29,41176	30,30303	30,00063
N10	33,00000	34,00000	33,33263	29,41176	30,30303	30,00063

Table 19.: Focus and context effect with silhouette rendering and edge enhancement using phong shading and rendering the buddha and dragon

Node	Time per frame in ms			Framerate per second		
	min	max	avg	min	max	avg
N01	33,00000	34,00000	33,33298	29,41176	30,30303	30,00032
N02	33,00000	34,00000	33,33246	29,41176	30,30303	30,00079
N03	33,00000	34,00000	33,33298	29,41176	30,30303	30,00032
N04	33,00000	34,00000	33,33298	29,41176	30,30303	30,00032
N05	33,00000	34,00000	33,33298	29,41176	30,30303	30,00032
N06	33,00000	34,00000	33,33246	29,41176	30,30303	30,00079
N07	33,00000	34,00000	33,33246	29,41176	30,30303	30,00079
N08	33,00000	34,00000	33,33298	29,41176	30,30303	30,00032
N09	33,00000	34,00000	33,33246	29,41176	30,30303	30,00079
N10	33,00000	34,00000	33,33298	29,41176	30,30303	30,00032

Table 20.: Focus and context effect with silhouette rendering and edge enhancement using cartoon shading and rendering the buddha and dragon

Node	Time per frame in ms			Framerate per second		
	min	max	avg	min	max	avg
N01	32,00000	35,00000	33,33298	28,57143	31,25000	30,00032
N02	32,00000	35,00000	33,33298	28,57143	31,25000	30,00032
N03	32,00000	35,00000	33,33298	28,57143	31,25000	30,00032
N04	32,00000	35,00000	33,33298	28,57143	31,25000	30,00032
N05	33,00000	34,00000	33,33298	29,41176	30,30303	30,00032
N06	32,00000	35,00000	33,33245	28,57143	31,25000	30,00080
N07	32,00000	35,00000	33,33298	28,57143	31,25000	30,00032
N08	32,00000	35,00000	33,33245	28,57143	31,25000	30,00080
N09	32,00000	35,00000	33,33245	28,57143	31,25000	30,00080
N10	32,00000	35,00000	33,33245	28,57143	31,25000	30,00080

Table 21.: Focus and context effect with silhouette rendering and edge enhancement using phong shading and rendering the buddha and dragon and bunny

Node	Time per frame in ms			Framerate per second		
	min	max	avg	min	max	avg
N01	33,00000	34,00000	33,33273	29,41176	30,30303	30,00055
N02	33,00000	34,00000	33,33273	29,41176	30,30303	30,00055
N03	33,00000	34,00000	33,33273	29,41176	30,30303	30,00055
N04	33,00000	34,00000	33,33273	29,41176	30,30303	30,00055
N05	33,00000	34,00000	33,33273	29,41176	30,30303	30,00055
N06	32,00000	35,00000	33,33273	28,57143	31,25000	30,00055
N07	33,00000	34,00000	33,33288	29,41176	30,30303	30,00041
N08	33,00000	34,00000	33,33273	29,41176	30,30303	30,00055
N09	32,00000	34,00000	33,33273	29,41176	31,25000	30,00055
N10	33,00000	34,00000	33,33273	29,41176	30,30303	30,00055

Table 22.: Focus and context effect with silhouette rendering and edge enhancement using cartoon shading and rendering the buddha and dragon and bunny

B. Visual Results

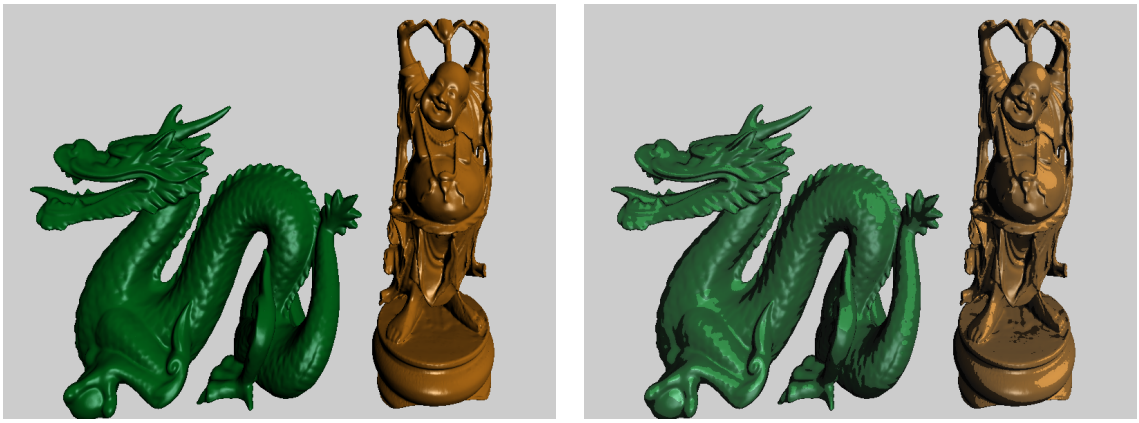


Figure 1.: Phong and cartoon shading

Figure 2.: Greyscaling with phong and cartoon shading

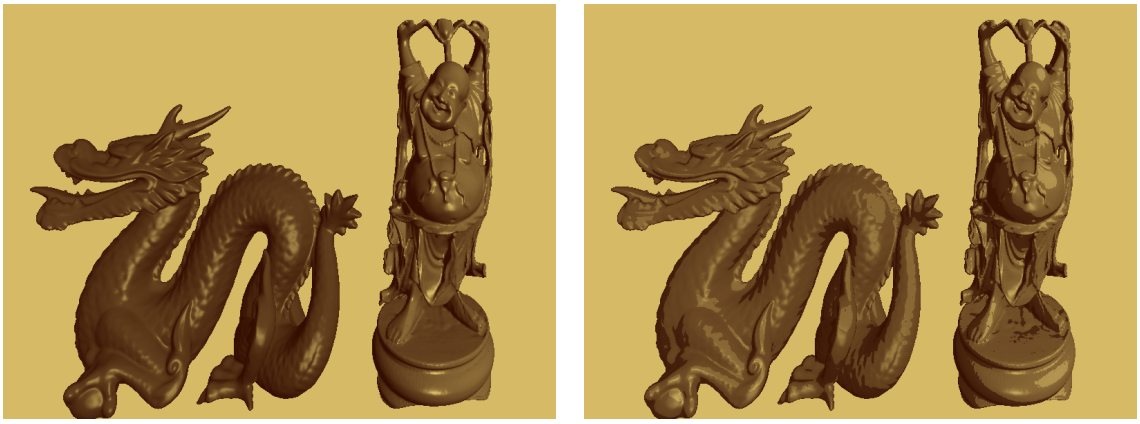


Figure 3.: Sephia with phong and cartoon shading

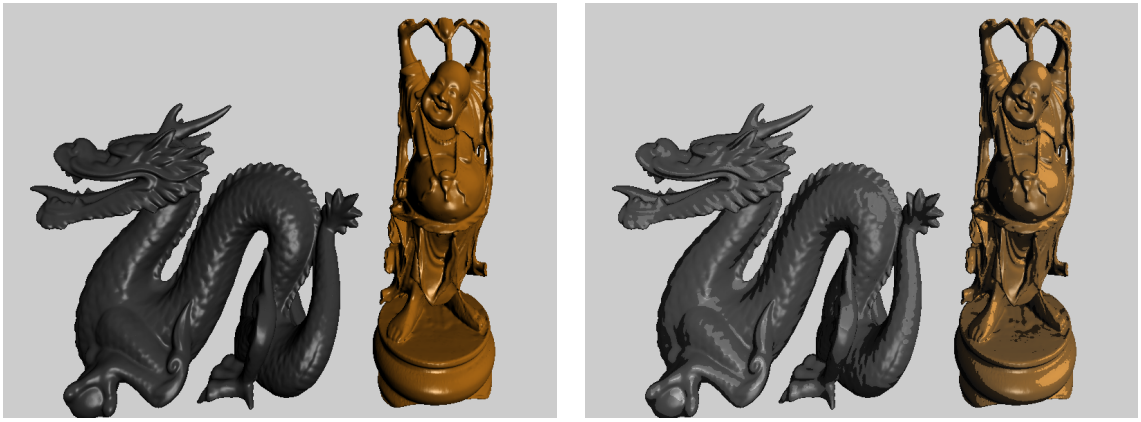


Figure 4.: Color selection with phong and cartoon shading

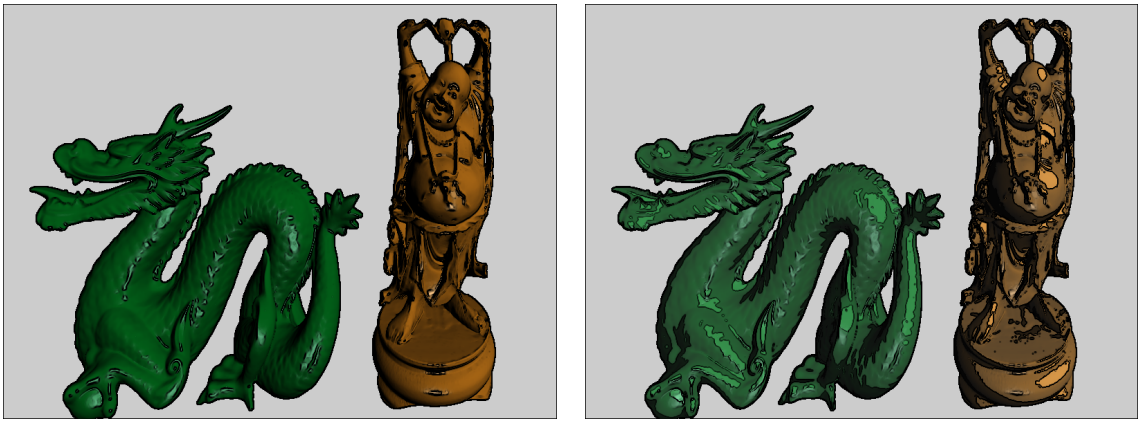


Figure 5.: Edge enhancement (Sobel filter) with phong and cartoon shading

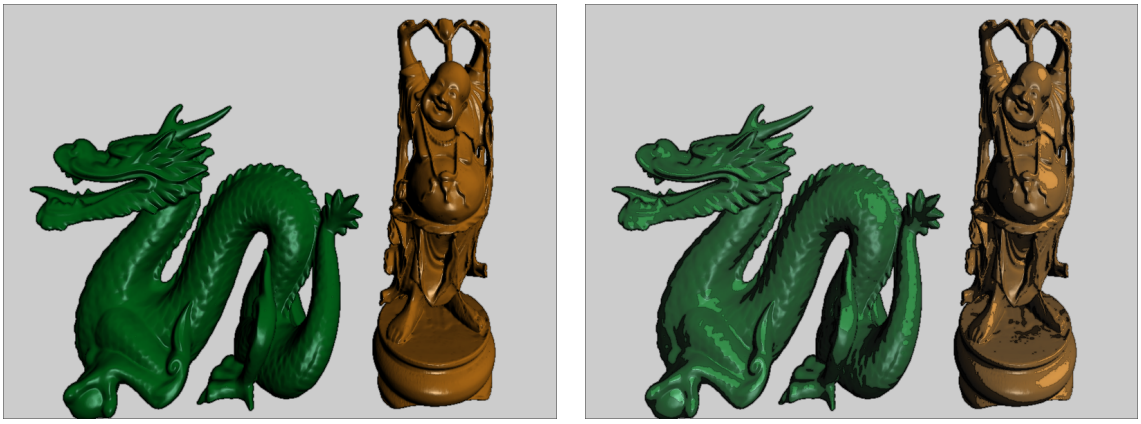


Figure 6.: Edge enhancement (FreiChen filter) with phong and cartoon shading

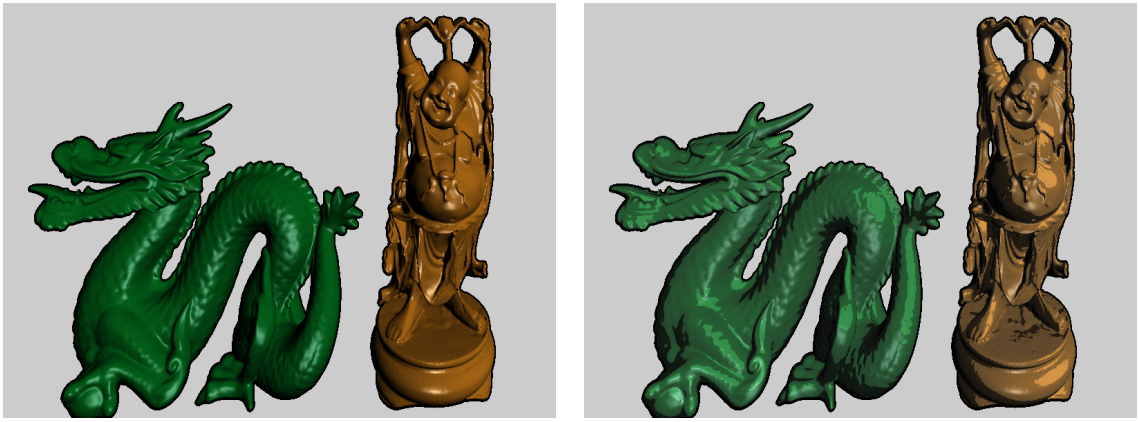


Figure 7.: Silhouette drawing with phong and cartoon shading

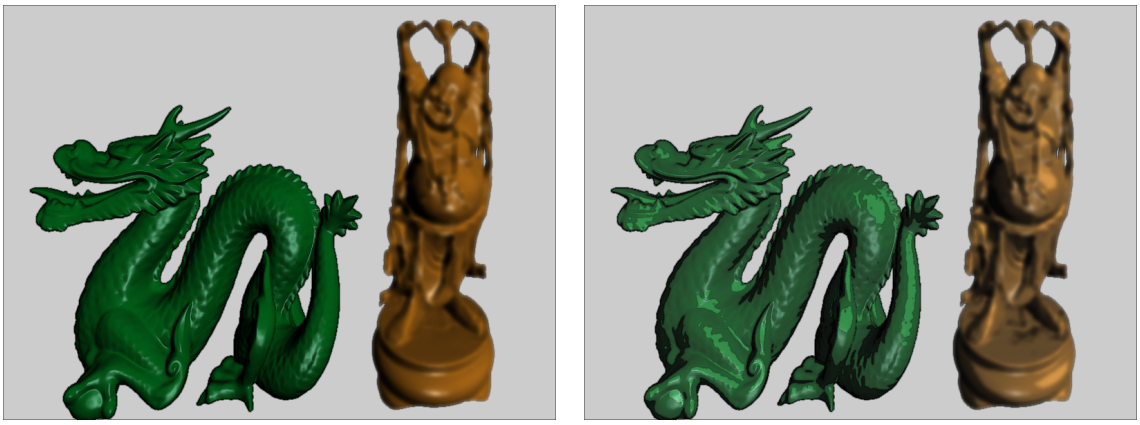


Figure 8.: Focus and context effect with phong and cartoon shading

List of Figures

Bibliography

[CJK11] Chun, Sungkuk, Keechul Jung and Jinwook Kim: *Oil Painting Rendering Through Virtual Light Effect and Regional Analysis.* In *Proceedings of the 10th International Conference on Virtual Reality Continuum and Its Applications in Industry*, VRCAI '11, pages 419–422, New York, NY, USA, 2011. ACM.

[CNSD+92] Cruz-Neira, Carolina, Daniel J. Sandin, Thomas A. DeFanti, Robert V. Kenyon and John C. Hart: *The CAVE: Audio Visual Experience Automatic Virtual Environment.* Commun. ACM, 35(6):64–72, June 1992.

[FC77] Frei, W. and Chung-Ching Chen: *Fast Boundary Detection: A Generalization and a New Algorithm.* IEEE Trans. Comput., 26(10):988–998, October 1977.

[GTDS10] Grabli, Stéphane, Emmanuel Turquin, Frédo Durand and François X. Sillion: *Programmable Rendering of Line Drawing from 3D Scenes.* ACM Trans. Graph., 29(2):18:1–18:20, April 2010.

[Hae90] Haeberli, Paul: *Paint by Numbers: Abstract Image Representations.* SIGGRAPH Comput. Graph., 24(4):207–214, September 1990.

[Hal04] Haller, Michael: *Photorealism or/and Non-photorealism in Augmented Reality.* In *Proceedings of the 2004 ACM SIGGRAPH International Conference on Virtual Reality Continuum and Its Applications in Industry*, VRCAI '04, pages 189–196, New York, NY, USA, 2004. ACM.

[KDMF03] Kalnins, Robert D., Philip L. Davidson, Lee Markosian and Adam Finkelstein: *Coherent Stylized Silhouettes.* ACM Trans. Graph., 22(3):856–861, July 2003.

[KLC07] Kang, Henry, Seungyong Lee and Charles K. Chui: *Coherent Line Drawing.* In *Proceedings of the 5th International Symposium on Non-photorealistic Animation and Rendering*, NPAR '07, pages 43–50, New York, NY, USA, 2007. ACM.

[KLK+00] Klein, Allison W., Wilmot Li, Michael M. Kazhdan, Wagner T. Corrêa, Adam Finkelstein and Thomas A. Funkhouser: *Non-photorealistic Virtual Environments.* In *Proceedings of the 27th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '00, pages 527–534, New York, NY, USA, 2000. ACM Press/Addison-Wesley Publishing Co.

[KLKL13] Kim, Yongjin, Yunjin Lee, Henry Kang and Seungyong Lee: *Stereoscopic 3D Line Drawing.* ACM Trans. Graph., 32(4):57:1–57:13, July 2013.

[LKKL13] Lee, Yunjin, Yongjin Kim, Henry Kang and Seungyong Lee: *Binocular Depth Perception of Stereoscopic 3D Line Drawings.* In *Proceedings of the ACM Symposium on Applied Perception*, SAP '13, pages 31–34, New York, NY, USA, 2013. ACM.

[LKL06] Lee, Hyunjun, Sungtae Kwon and Seungyong Lee: *Real-time Pencil Rendering.* In *Proceedings of the 4th International Symposium on Non-photorealistic Animation and Rendering*, NPAR '06, pages 37–45, New York, NY, USA, 2006. ACM.

[LM02] Lum, Eric B. and Kwan-Liu Ma: *Hardware-accelerated Parallel Non-photorealistic Volume Rendering.* In *Proceedings of the 2Nd International Symposium on Non-photorealistic Animation and Rendering*, NPAR '02, pages 67–ff, New York, NY, USA, 2002. ACM.

[LMHB00] Lake, Adam, Carl Marshall, Mark Harris and Marc Blackstein: *Stylized Rendering Techniques for Scalable Real-time 3D Animation.* In *Proceedings of the 1st International Symposium on Non-photorealistic Animation and Rendering*, NPAR '00, pages 13–20, New York, NY, USA, 2000. ACM.

[MSGS13] Magdics, Milán, Catherine Sauvaget, Rubén J. García and Mateu Sbert: *Post-processing NPR Effects for Video Games.* In *Proceedings of the 12th ACM SIGGRAPH International Conference on Virtual-Reality Continuum and Its Applications in Industry*, VRCAI '13, pages 147–156, New York, NY, USA, 2013. ACM.

[ND04] Nienhaus, Marc and Jürgen Döllner: *Blueprints: Illustrating Architecture and Technical Parts Using Hardware-accelerated Non-photorealistic Rendering.* In *Proceedings of Graphics Interface 2004*, GI '04, pages 49–56, School of Computer Science, University of Waterloo, Waterloo, Ontario, Canada, 2004. Canadian Human-Computer Communications Society.

[RBGV08] Rautek, Peter, Stefan Bruckner, Eduard Gröller and Ivan Viola: *Illustrative Visualization: New Technology or Useless Tautology?* SIGGRAPH Comput. Graph., 42(3):4:1–4:8, August 2008.

[RL13] Rosin, Paul L. and Yu-Kun Lai: *Non-photorealistic Rendering with Spot Colour.* In *Proceedings of the Symposium on Computational Aesthetics*, CAE '13, pages 67–75, New York, NY, USA, 2013. ACM.

[RL15] Rosin, Paul L. and Yu-Kun Lai: *Non-photorealistic Rendering of Portraits.* In *Proceedings of the Workshop on Computational Aesthetics*, CAE '15, pages 159–170, Aire-la-Ville, Switzerland, Switzerland, 2015. Eurographics Association.

[SF68] Sobel, I. and G. Feldman: *A 3x3 Isotropic Gradient Operator for Image Processing.* Never published but presented at a talk at the Stanford Artificial Project, 1968.

[SLKD15] Semmo, Amir, Daniel Limberger, Jan Eric Kyprianidis and Jürgen Döllner: *Image Stylization by Oil Paint Filtering Using Color Palettes.* In *Proceedings of the Workshop on Computational Aesthetics*, CAE '15, pages 149–158, Aire-la-Ville, Switzerland, Switzerland, 2015. Eurographics Association.

[Sut65] Sutherland, Ivan E.: *The Ultimate Display.* In *Proceedings of the IFIP Congress*, pages 506–508, 1965.

[The] *The Stanford 3D Scanning Repository.* `http://graphics.stanford.edu/data/3Dscanrep/`. (Accessed on 04/27/2016).