



# Realistic Rendering of Environmental Conditions in a Construction Machinery Simulator Applying GLSL Shaders

Behboud Kalantary (Fraunhofer-IGD),
Gino Brunetti (Fraunhofer-IGD),
Alvaro Segura (VICOMTech),
Aitor Moreno(VICOMTech),
Dr. Ulrich Hofmann (Universität zu Lübeck)

January 23, 2006

Supported by


**EC - Contract n° COLL-CT-2003-500452**
**EUROPEAN COMMISSION**
**COLLECTIVE RESEARCH PROJECT**
**"Versatile augmented reality simulator for training in the safe use of construction**

**machinery”**

# Abstract

Machinery specific simulators are designed for realistic and immersive “look and feel” of environments and behaviors of the real artifacts they resemble. But often only the “feel” component of such simulators is credible due to a lack of a realistic rendering. Current simulators use the graphic cards fixed function pipeline, and do not use the second processor unit, the Graphics Processing Unit (GPU) they have access to. Most of their high quality rendering is done by applying texture mapping techniques.

In this work it is shown how to improve the rendering of specific materials and environments applying the new OpenGL shading language (GLSL) defined in the OpenGL 2.0 standard. Special attention is given to the rendering of weather conditions and soil as they contribute most to the environment perception in a construction machinery simulator, which will be the test environment of the developments shown in this work.

The presented approach is implemented as part of the framework VAR-Trainer [VT05], which is an European funded Research and Development project whose main objective is the development of an advanced simulator platform for heavy construction machinery. VAR-Trainer uses the open source scene graph and renderer library OpenSG [Ope05c] that is based on OpenGL. Two methods for rendering a scene are implemented in this work: shader-based and texture mapping techniques. Using the two methods the assumption that a higher level of realism is gained using shaders is evaluated and a perception and importance metric is developed. The result for a construction site scenario is that soil and atmospheric effects are the most relevant visual effects to be included , followed by rain and clouds rendering, whereas the rendering of a realistic sky could be neglected, if, for instance, general performance criteria have to be considered.

# Contents

# List of Figures

# Listings

# List of Tables

# Chapter 1.

# Introduction

Several driving simulators for training people in certain environments exist. Examples are simulators for construction machinery environments, military environments, urban environments, etc. But looking at their rendering output, most of them provide a poor visualization even for those parts most relevant for the respective environment and simulation. It should be assumed that for every simulator certain special parts exist, which have to be concentrated on. While an urban vehicle simulator must focus on weather conditions, buildings, vehicles and humans, a construction machinery simulator has to focus on rendering soil, weather conditions, workers and maybe other construction machines.

The importance of a high visualization quality in the case of weather conditions, for instance, can be derived from its role in the real scene to be simulated. Weather conditions are direct and indirect components of any outdoor environment. A rainy day, for instance, changes the lighting situation directly. Indirectly it suggests that the user has to take special care when operating in a rainy situation.

## 1.1. Work Description

The goal of the presented work is to improve the visual output for a construction machinery environment, which is the VAR-Trainer simulation environment [VT05]. VAR-Trainer is a project of the European Commission that has been initiated as a response to the high accident rate within the construction sector. The goal of the simulator is to provide an environment for construction machinery operators, such that they can train critical situations and learn to react properly to them, without endangering the trainee. To make the simulation experience as realistic as possible a high quality visualization of the exterior environment is crucial. Thus, main focus is on rendering weather conditions and soil to build a credible construction site.

The developed system should have controllability over several conditions that are responsible for a specific weather condition: rain intensity, cloud cover, amount of fog and haze, wind

strength and direction and the position of the sun (time of day). For a human observer the soil should be granular enough to be realistic and it should change its color when the soil is from a certain depth. Keeping the performance in mind, the developed system should run with an interactive framerate, i.e. it should allow a real-time rendering with at least 25 frames per second (fps).

The assumption of this work is that a higher level of realism can be obtained using shader technology on a Graphics Processing Unit (GPU) for rendering weather conditions and soil, while leaving the resources of the Computer Processing Unit (CPU) to the simulation application. Hardware shaders permit manipulation of the transformation and lighting of every single vertex, and the color of every single pixel. To achieve independence from a specific GPU and operating system the OpenGL shading language (GLSL) [Ros04] is used. It is defined in the new OpenGL 2.0 [OPE05a] specification for programming the GPU.

As a visualization framework the graphics system OpenSG [Ope05c] is used to keep platform independence and extensibility.

Moreover, GLSL has a C-like syntax which enables to program the GPU with a high level programing language rather than using a low level assembler language. Choosing OpenSG as rendering framework has many advantages. It allows to extend the rendering on a cluster of computers, and it allows to render in stereoscopic view by rendering into a quadbuffer, displaced color buffers or displaced multi-screen output, just to mention a few possibilities that are useful for rendering an immersive scene.

## 1.2. Summary of Main Results

As main achievement of this work we combine and improve existing methods used to visualize soil, atmospheric effects, clouds and rain. Soil is an adapted method of [Don05] for rendering high frequency, granular surfaces. The color of soil varies with its horizon and thus from which depth it is. This information can either be stored in a height map or in one of the texture coordinate attributes every vertex can have. This information is normalized and used to look-up in a pre-designed color gradient texture to simulate smooth change in the color.

The atmospheric effect realized is an adaption of [O'N05], which is capable to render the sky, sun and atmospheric effects like fog, haze and atmospheric perspective. We improved the method for the needs of VAR-Trainer, and added more possibilities, like position of the sun, intensity of haze at the horizon, etc. to influence the result.

The clouds are based on the idea of [Dub05], but also improved for the specific needs of a construction machine simulator. The clouds are 2.5 dimensional and built using noise textures of different octaves. By adding weighted and scaled octaves together, subtracting

low values from the result and exponentiating all together, the output results in a credible cloud texture. In order to improve the outcome we introduce a new way of shading the clouds with simple average operations with other clouds textures.

Furthermore, we realized an adaption of the method from [Wan05], which renders rain by mapping a texture on a double cone transforming the texture coordinates with a matrix. In addition, we implemented a method to render rain drops on virtual windows by masking a rain drops texture with a noise texture.

In order to evaluate the results of this work and the initial assumptions we conducted a series of experiments. The experiments consists of two setups: immersive and non immersive. Thus we not only compare shader vs. non shader (texture mapping) but also immersive shader vs. immersive non shader and vise versa. Based on the evaluation, we build a metric to decide which shader is responsible for the highest credibility, and which shader can be switched off on a less performant GPU.

Stokes references in [SFWG04] many papers describing perceptual metrics that can be used to establish stopping criteria for high quality rendering systems and using perceptual metrics to optimally manage resource allocation for efficient visualization. Those metrics are gathered either trough perceptual experiments where people make subjective decisions on visualizations or through the difference of an image to a *gold standard* image [PF03]. The use of such a metric is to render highest possible quality image under given constraints. Particularly, Ahumada [Ahu93] and Ramasubramania [Ram00] provide good reviews of computational image quality metrics developed by computer graphics and vision science fields.

Figure 1.1 describes schematically the assumption. On the right side, the static scene maps the input textures directly on the mesh, using texture coordinates (uv-space coordinates) [TMR99a]. On the left side the textures are used to generate procedural texture mapping [TMR99b] and textures on the GPU. While the scene on the right is static, the scene on the left provides direct manipulation of factors like: wind direction, wind strength, sun position, cloud cover, etc.

In the last part of this work we evaluate the performance of the shaders and the improved perception of shaders versus textures. The result is that most shaders use less than 20 cycles per-fragment and fulfill the requirements of a real time application. Moreover, the test application reaches over 50 fps[1] with all shaders visible.

We also evaluate the results in a subjective perceptional manner and compare the solutions of the rendering with real photos. The result is that the level of photo realism can not be reached fully, but some effects can be simulated in real-time and have a credible look compared to the reality. The perceptional evaluation shows that in some cases a shader does

[1]On a PC with Pentium 4 with 3 GHz, 1GB RAM and Geforce 6800 GT (PCI-Express).

not absolutely achieve a better result than a simple texture mapping. The experiments also show that there is a small difference between immersive and non-immersive visualization.

Moreover, the results of the experiments allow for building a perceptional metric of the importance of the implemented shaders. This metric can be used to decide which shader could be replaced by simple texture mapping, e.g. because of high GPU load. The ranking of the shaders based on the metric is:

1. Soil and Atmospheric Effects
2. Rain
3. Clouds
4. Sky

## 1.3. Organization of This Document

The work is organized as follows. Firstly, in chapter 2, we review the state-of-the-art in the visual capabilities of driving simulators, particularly the visualization of construction machinery simulators. Further, we review existing techniques, which can be used to visualize materials often seen in a construction machinery environment, namely soil, grass, wood, rusted steel, dirt and concrete. Then we review analytical models and techniques used to approach the visualization of certain weather conditions.

In chapter 3 we discuss how soil and weather conditions can be approached by choosing techniques described in chapter 2, but using these techniques in another context and improved for our requirements. Additionally, mathematical concepts of every shader, and how we improved them making simple assumptions, are described.

Chapter 4 gives an introduction into a GPU architecture, particular the Nvidia NV40 chip [NVI05a], and GPU programing using GLSL. Then, we explain the OpenSG library and how to use it as a back-end for shaders. Afterwards, all shaders are reviewed in detail and explained.

In chapter 5 we initially overview the results of the realized shaders and discuss their performance. Afterwards, the perceptional evaluation experiment is declared. As a result of the experiments the importance metric for the implemented shaders is presented.

Finally, chapter 6 discusses the major results and conclusions of this work, including considerations on future research in visualizing realistic materials and weather conditions for a construction machinery simulator.

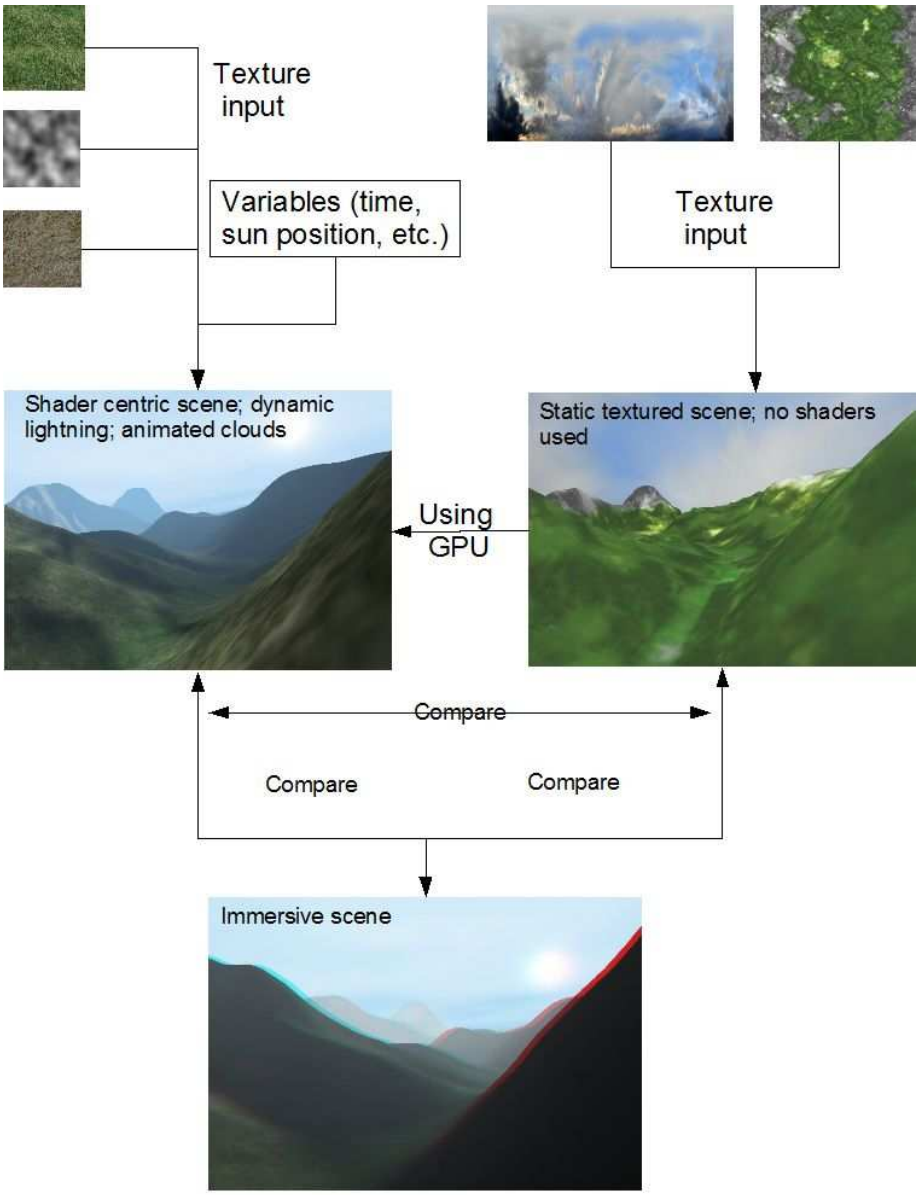


Figure 1.1.: Schematic view of the assumption in this work. Using the GPU we can reach higher level of realism. By evaluating the propose we get a metric on the importance of every effect.

# Chapter 2.

# State of the Art

This chapter reviews previous visualization attempts in driving simulators, particularly visualization in car simulators, truck simulators and construction machinery simulators. The reason why we also review related work in other simulators than construction machinery simulators is that most vehicle simulators attempt to simulate outdoor sceneries. Outdoor sceneries often include visualization of weather conditions and typical materials like: grass, sand, soil, concrete, rust, etc. Further, we review previous techniques that can be used to visualize materials often seen in a construction machinery environment. Then we review analytical models and techniques used to approach certain weather conditions. In the last section we give a brief overview of previous perceptual evaluation experiments, used to gather metrics for predictions on practicable quality under certain constraints.

## 2.1. Visualization in Driving Simulators

This section gives a brief overview on existing driving simulators. We refer the interested reader to [INR05] for a more complete list of driving and other vehicle simulators. Unfortunately, most simulators do not provide specifications about their visualization module, why we included a screen shot either taken from the web, from executable demos or video demos to have a subjective impression.

### 2.1.1. Car Simulators

**Northeastern University** The Virtual Environments Laboratory [Nor05] at Northeastern University in Boston, Massachusetts, is building on an OpenGL-based driving simulator for some of their projects. Current research projects focus on development of virtual reality driving simulators for elderly drivers and rehabilitation assessment research. Figure 2.1 shows the graphical output of the Simulator. The visualization dos not seem to use shaders to improve the perception of the environment or specific

details. Thus, the renderer used in this project mostly applies textures to surfaces to improve the visual impression. State of the art PC-hardware is used for rendering.

Figure 2.1.: Northeastern University Simulator at Virtual Environments Laboratory in NU Boston (USA), with head mounted display (HMD).

**University of Zaragoza** The University of Zaragoza develops on SIMUSYS [SIM05b], which is a simulator taking visualization of different weather conditions into account. In Figure 2.2 four scene renderings with different weather conditions are shown.

The 3D engine used in SIMUSYS has the following features:

- OpenGL-based;
- Dynamic stencil shadows;
- Environment map reflections;
- 3D sound generation;
- Frustum culling optimization.

However, the engine does use the fixed function pipeline and texture mapping to realize the effects of changing weather, displaying shadows and reflections. Standard PC-hardware is used to render a scene.

**STSoftware Simulation Technology** ST Software [STS05] provides a number of tools to develop a driving simulator and its environment. Figure 2.3 shows a driving scenario. The 3D engine uses rendering technology based on the OpenSceneGraph [Ope05b] rendering toolkit and enables real-time frame rates on standard PC hardware, e.g. up to 60 fps for average complex city databases including 20 or more simulated cars. Although OpenSceneGraph is capable of shaders, the renderer seems to use just standard texture mapping.

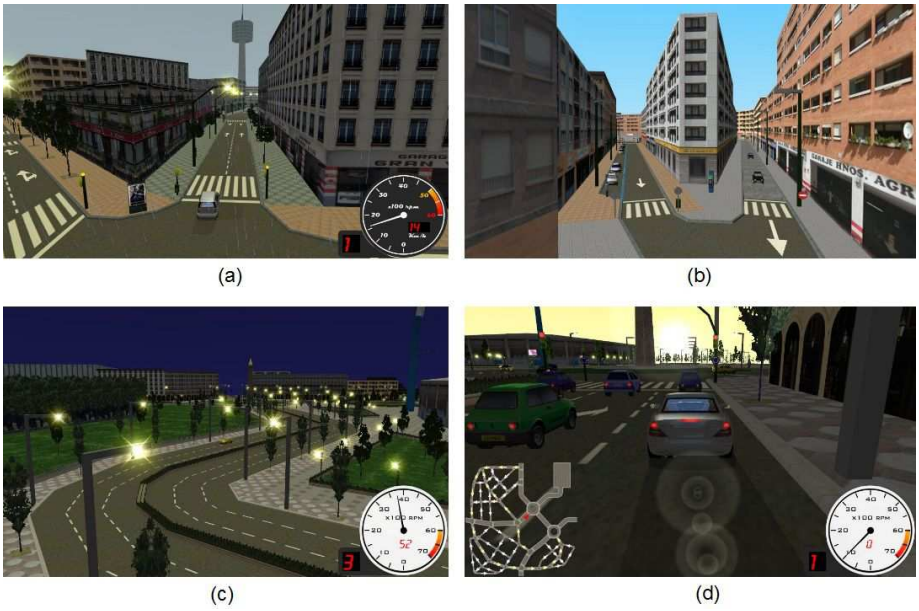


Figure 2.2.: SIMUSYS Dynamic Driving Simulator. Features dynamic configuration of weather and day time: (a) rainy, (b) sunny, (c) night, (d) dawn scene.

**VTI Simulators** The Simulator III [VTI05] at the Swedish National Road and Transport Research Institute can be fully adapted for cars, trucks and in the near future also for train simulators. Figure 2.4 shows two typical scenes from their simulator. The snow can be dynamically added to the scene.

The renderer uses texture mapping to achieve improved visual perception. The simulator renders with state of the art PC-hardware on a 120 degree screen. Additionally, the three rear mirrors display correct rear views.

**E-Com Simulation & Training Systems** One of the products of the company is the Civilian Driving Simulator. Their Visualization module is a PC-based visualization system [Sim05a]. Figure 2.5 is a rendering of a rainy and foggy scene that are results of texture mapping techniques using the fixed function pipeline.

**VSTEP Virtual Safety Training and Education Platform** VSTEP [VST05] has developed a driving simulator that allows driving students to experience real-life traffic situations. VSTEP's 270-degree simulator uses PC-based hardware for rendering. Figure 2.6 shows four different weather conditions that can be manipulated dynamically. However, the simulator uses texture mapping techniques for all renderings.

Figure 2.3.: STSoftware Simulation Technology. Car Driving Simulation Software. Use of open source OpenSceneGraph 3D graphics libraries [Ope05b] for rendering tasks.

Figure 2.4.: VTI Simulators can change weather conditions dynamically.

Figure 2.5.: E-Com Simulation & Training System scene shows a rainy environment.

Figure 2.6.: VSTEP Car Simulator can change weather conditions and day time.

### 2.1.2. Truck, Bus and Train Simulators

**LANDER Simulation & Training Solutions** LANDER [LAN05] designs and develops PC-based simulators with motion platform. It builds visualization software for trains, tramways, high speed trains, diesel locomotives, buses, cars, trucks, cranes, trolleys and heavy machines. No specification for the visual system are published (Figure 2.7), but the renderer seems to use texture mapping techniques.

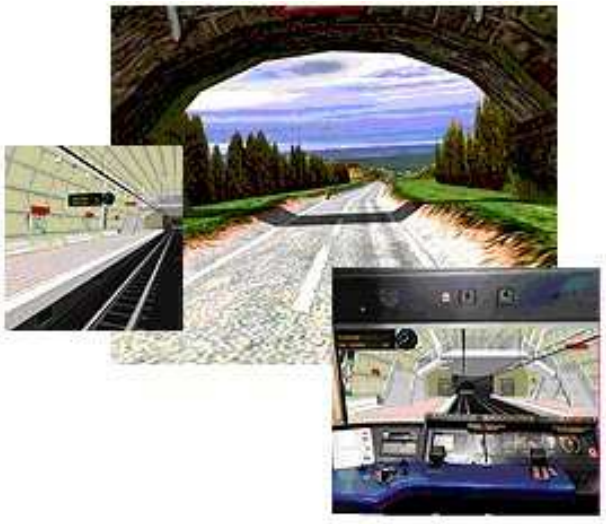

Figure 2.7.: LANDER Simulation & Training Solutions.

**TRUCKSIM** TRUCKSIM is dedicated to the development of the first full mission Truck Driver Training Simulator in the UK [TRU05]. No specifications about their visualization module are published. However, the renderer seems to use texture mapping to provide details. Figure 2.8 shows a visual output of the capabilities of the renderer.

Figure 2.8.: TRUCKSIM.

### 2.1.3. Construction Machinery Simulators

**CMLABS** One of the products of CMLABS is Vortex [Phy05]. Vortex' main goal is to simulate accurate physical behavior of machines, robots and environments. Figure 2.9 shows a rich textured excavator. The underlying hardware is PC-based. Texture mapping is used to render the environment and the machines. Vortex uses high detailed textured machines to improve the perception.

Figure 2.9.: CMLABS Vortex development platform for simulating physical behaviors.

**Immersive Technologies Advanced Training Simulators** Immersive Technologies has developed a mining truck simulator environment. The simulator makes use of special hardware to generate the images over the 180 degree field of view or 4.5 meter wide wrap around screen [Tec05]. Figure 2.10 shows some mine environments and machines, which are enriched with high detail textures using texture mapping techniques.

**SARTURIS** SARTURIS is a research project founded by the Bundesministerium für Bildung und Forschung (BMBF). The visual part of the interactive machine simulator visualizes construction machinery, particularly the process of excavating [SAR05] (Figure 2.11). The renderer uses PC-based hardware and texture mapping with simple textures.

**S.H.E. Project** The S.H.E. (Simulateur de Excavatrice Hydraulique) [Okt05] project has been an European Project funded in the EC ESPRIT program. Its aim was to develop a prototype for a fully interactive training simulator for operators of hydraulic excavators. The system uses a SGI ZX10VE with the 3Dlabs Wildcat 4210 and great OpenGL performance. The renderer used texture mapping techniques with very simple textures as shown inFigure 2.12.

Figure 2.10.: Immersive Technologies mining truck simulator.

Figure 2.11.: SARTURIS.

Figure 2.12.: A typical construction machinery site rendered in the S.H.E. Project.

## 2.2. Materials Present in a Construction Site Environment

In this section we review previous work for visualizing some often seen materials at a construction site, which are soil, grass, wood, rust, dirt, and concrete.

### 2.2.1. Soil

Soil is the most often seen material in a construction site environment. Typically it is rendered using texture mapping [TMR99a], which is one of the primary techniques to improve the appearance of objects rendered. It is typically used to provide color detail for complex surfaces by modifying the surface color. For example an image of concrete supplied by a texture can make a flat polygon appear as if it is made of concrete.

For rendering soil, however more surface details are necessary, as soil is often perceived as a complex, coarse and granular surface. But, instead of adding more micro polygons to the mesh with the displacement mapping technique [Coo84], J. Blinn introduced bump mapping [Bli78], which simulates wrinkles or the bumps in a surface without the need for geometric modifications to the model on a per fragment level [TMR99c].

Later, Tomomichi Kaneko et al. [Tom01] introduced parallax mapping which results in a better approximation of the parallax observed on uneven but smoothly varying hight fields. However, parallax mapping or bump mapping alone can not simulate occlusions, silhouettes, or self shadowing, and it is also not suitable for high frequency details. Horizon maps [Max88] provide a solution for shadowed bump-mapped surfaces, which can be implemented using graphics hardware [SC00].

Relief mapping [POC05] uses a combination of linear and binary search on ray intersection with a hight map, which produces correct self occlusion, shadows and all standard per pixel lighting and filtering effect. Unfortunately, to resolve small details it is necessary to increase the number of steps in the linear search which then becomes too expensive.

View Dependent Displacement Mapping (VDM) [WWT$^+$03] is a technique to treat displacement mapping as a raytracing problem by precomputing the distances from each displaced point to a reference surface. This technique simulates correct self occlusion, shadows, all standard per pixel lighting and filtering effects and silhouettes. Unfortunately, VDM and its extension Generalized Displacement Mapping (GDM) [XXS$^+$04] require a five dimensional representation of the data-set and thus significant amount of storage. Figure 2.13 shows the difference between the techniques described.

Per-pixel displacement mapping with distance functions [Don05] handles small details using sphere tracing [Har96] and a precomputed distance map using euclidean distance


Figure 2.13.: Comparison of different rendering techniques: (a) bump mapping, (b) horizon mapping, (c) conventional displacement mapping, and(d)view-dependent displacement mapping with self-shadowing [WWT+03].

mapping [Dan80]. This technique simulates correct self occlusion, shadows, all standard per-pixel lighting and filtering effects, but it requires newest graphics hardware with Shader Model 3.0 capability (Figure 2.14).

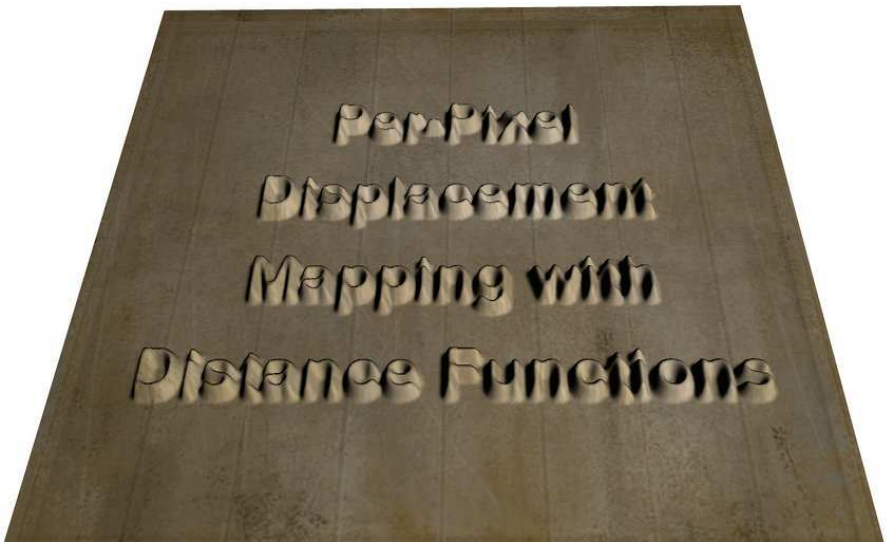


Figure 2.14.: The effect of per-pixel displacement mapping [Don05].

### 2.2.2. Grass

Traditionally, grass has been rendered by applying grass textures to the surface with texture mapping [TMR99a], which gives a nice approximation if the viewpoint is far away or the visualized grass tend to be short. Rendering static grass stalks can be done by billboarding [TMR99d] or using the widgets method [Bro05]. Both methods result in

aliased edges which can be overcome using latest hardware with per-primitive super-sample and multi-sample modes for antialiasing primitives with transparent fragments.

Volumetric texture rendering technique as discussed by Perbet and Cani [PC01] can be used to visualize animated grass, if high amount of texture memory usage is acceptable. Bakay ,Lalonde and Heidrich introduced a way to have real time animated grass [BLH02], which requires high amount of vertices on the surface for realistic look of thin and short grass stalks.

Some very recent approaches for rendering animated and dynamic grass using recent hardware shaders was introduced by Pelzer [Pel04] and Wang et al [WWZ+05]. In [Pel04] the vertices of the grass widget are offset to form a wind direction. Wang et al [WWZ+05] introduced a skeleton-based modeling and rendering method to simulate the grass wagging in the wind (Figure 2.15).

Figure 2.15.: A meadow under whirlwind [WWZ+05].

### 2.2.3. Wood, Rust, Dirt and Concrete

Materials like above mentioned materials can be approximated using texture mapping [TMR99a]. Using new shader capabilities, Nvidia [NVI05a] and ATI [ATI05] introduced Software Development Kits (SDKs) with many procedural methods [TMR99b] for rendering wood, rust, dirt, concrete and many other materials. Most of them are using combinations of multi texturing, (dynamic) bump mapping, bidirectional reflectance distribution functions and other lighting techniques [TMR99e]. Nvidia presented in *Ogres and Fairies: Secrets of the Nvidia Demo Team* [NVI05b] techniques their demo team used to create the *Time Machine* demo, where different materials get affected by weathering when time passes by (Figure 2.16 and Figure 2.17).

Figure 2.16.: Nvidia - time machine demo showing rust and dirt shader [NVI05a].

Figure 2.17.: Nvidia - time machine demo showing wood shader [NVI05a].

## 2.3. Weather Conditions

In this section we review previous work in real time and interactive visualizing of weather conditions. To visualize different weather conditions on our planet, it is necessary to visualize a sky with one sun, clouds and rain. Additionally, atmospheric light scattering needs to be visualized in order to support more photo realism.

### 2.3.1. Sunlight and Atmospheric Effects

For a day light outdoor scene the major light source is the sun. As the sun light travels trough the atmosphere and hits particles, it changes wavelength and thus color. Hoffman and Preetham [HP03] presented a real time approach for simulating Areal Perspective and physically-based skylight for clear day skies(Figure 2.18). Although their model can be implemented on modern GPUs and performs fast due to simplifications on the scattering equation described by Preetham, Shirley and Smits [PSS99] (Figure 2.19), it produces an unnaturally bright area opposite the sun direction, an unnatural and huge sun, and it is limited to a constant density atmosphere. Nielson [Nie03] expands the above mentioned method to consider the density change in the atmosphere.

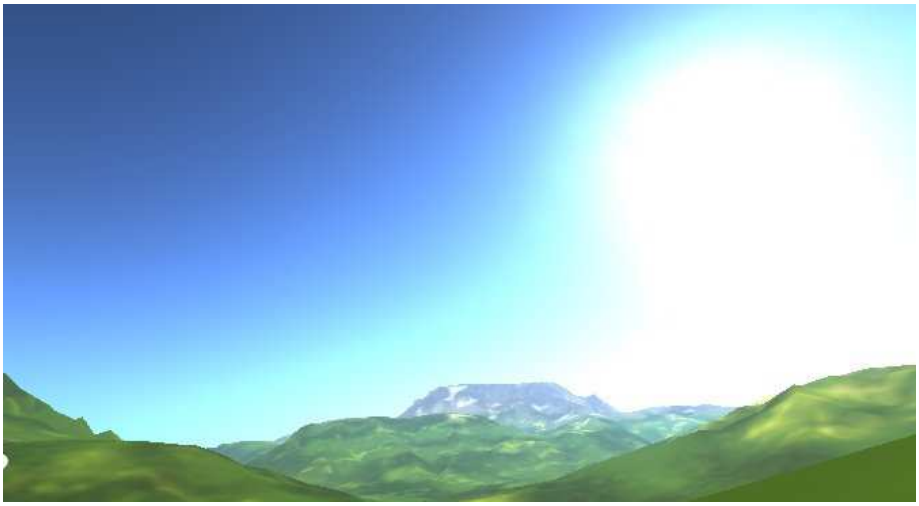

Figure 2.18.: Sky rendered using *Rendering Outdoor Light Scattering in Real Time* presented by Hoffman and Preetham [HP03].

Accurate atmospheric scattering as presented by O'Neil [O'N05] is an approach for rendering a planet with its atmosphere not only from a certain height, but for example, also from space. The model implements the atmospheric scattering algorithm described by Nishita et al. [NSTN93] entirely on the GPU and performs at interactive rates on modern graphics cards (Figure 2.20).

Figure 2.19.: Sky rendered using *A Practical Analytic Model for Daylight* presented by Preetham [PSS99].

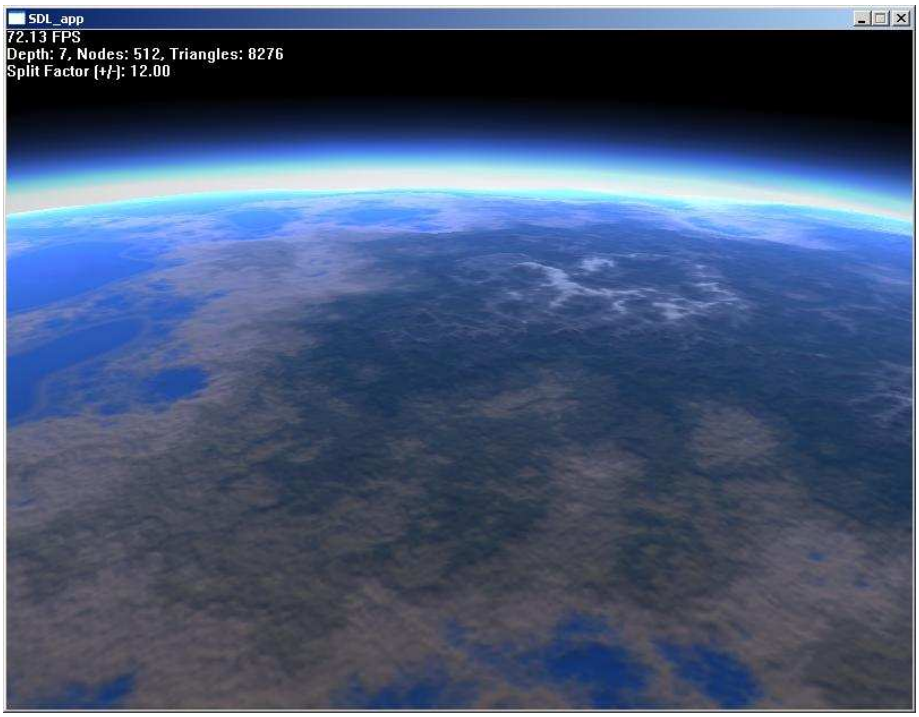


Figure 2.20.: Sky rendered using the *Accurate Atmospheric Scattering* presented by O'Neil [O'N05].

### 2.3.2. Clouds

Clouds play an important role in achieving a dynamic and realistic outdoor scene, though in a construction machinery environment most workers are concentrated on their machine and the ground. Nevertheless, without clouds a rainy scene would not be credible.

Isidoro and Riguer [IR02] described a method based on texture perturbation effects for rendering animated clouds using the GPU (Figure 2.21). Though their method has good performance, it does not create realistic shaded clouds, and the cloud cover can not be parameterized. Gehling [Geh05] uses texture pertubation and a shading and lighting model described by Laeuchli [Lae05], which applies per-pixel lighting by calculating the angle between the sun and the current texel in tangent space (Figure 2.22). Unfortunately, this lighting model does not take into account any of the physical aspects of real cloud lighting.

Figure 2.21.: Clouds rendered using texture perturbation on the GPU [IR02].


Figure 2.22.: Clouds rendered using texture perturbation and shading [Geh05].

Real-time cloud rendering as presented by Harris and Lastra [HL01] takes multiple anisotropic scattering of light into account and builds volumetric clouds from texture splatted particles. Later, Harris et al. [HBSL03] introduced a model for realistic and animated clouds with fluid motion and thermodynamic forces using entirely the GPU (Figure 2.23).

Figure 2.23.: Clouds rendered using fluid motion and texture splatted particles with multiple scattering for shading [HBSL03].

Another approach to realistic and fast cloud rendering was described by Wang [Wan03], where the shading model and form of clouds is more artistic driven rather than solving light scattering and noise equations (Figure 2.24). Additionally, animation of cloud formation and dissipation is discussed.

Figure 2.24.: Clouds defined by a pre-design step which then are rendered using texture splatted particles [Wan03].

In [Dub05] Dube presents a procedural algorithm to generate clouds from noise functions with parameterizable cloud cover, and a 2.5 dimensional tracing in texture space for accurate shading. Although, the algorithm produces realistic clouds, it needs very recent graphics cards for interactive frame rates. Also, this method does not produce volumetric clouds.

### 2.3.3. Rain

Rain is in many ways responsible for adding more realism to an outdoor scene. Precipitation can be visualized by rendering a large scaled particle system. Reeves discussed in [Ree83] how to design such a particle system, modeling a class of fuzzy objects. Furthermore, a modern approach for a GPU-based particle engine was described by Kipfer et al. in [KSW04]. However, particle-based rain is sometimes an expensive effect when the dense of precipitation and thus the required number of particle increases.

Starik and Werman presented in [SW99] a postprocess effect for adding rain to a video without the knowledge about the 3D scene. They describe a repetitive 2D random spatial pattern to create the impression of rainfall as, apparently, humans eye can not see the individual drops. In Wang [Wan05] discussed a model for rendering a texture mapped double cone which surrounds the camera. By using hardware translation on the texture matrix it is possible to ensures a realistic effect of precipitation.

ATI [ATI05] presented a new demo named *Toy Shop*, which renders highly realistic and accurate rain using about 300 shaders. On the time this work is written no specifications about the shaders has been published.

## 2.4. Summary

In this chapter we first reviewed some visualization attempts for driving simulators, particularly visualization in car simulators, truck simulators and construction machinery simulators. It shows that most simulators do not use the advantage of shaders even when they have access to the required hardware. To train a worker in a construction machinery environment, we assume that it is essential having a credible environment where the worker can focus on his tasks as if he/she is immersed in a virtual world that causes psychological reactions to him like in a real world. This implicates that a simple and rudimentary visual impression would cause the worker to play a game and not be concentrated and focused on his/her tasks. The VAR-Trainer [VT05] project provides the platform of a shader centric architecture for use in a construction machinery site, and thus provides best possibility to build a rich and detailed environment with focus on photo realism.

In the second section of this chapter we reviewed previews work in rendering specific materials seen in a construction site environment. All materials might be approximated by simple texture mapping. However, to achieve more visual realism shaders shall be used. The materials to be rendered are:

- **Soil** is the most common material in a construction machinery site and thus the most important. Here per-pixel displacement mapping technique provides high frequency details with best performance results when recent hardware is used. If silhouettes are an important factor for a coarse and rough surface, which is when the viewers distance to the surface is small and the angle to the surface results in a view where the silhouette can be perceived exactly, VDM or GDM method should be used.
- **Grass** can surround a construction site environment or be a part of it in primary stages. As grass is not viewed from a close distance, (the workers position is defined be in a machine to about 2-3 meters above the ground) it is not necessary to visualize grass stalks.
- **Wood, Rust, Dirt and Concrete** will not be part of this work as techniques described in 2.2.3 are sufficient enough to get nice results.

In the third section of this chapter we reviewed weather conditions as they are an essential part of a day light outdoor scene. There exist mathematical models for *atmospheric scattering* as well as good approximations this models. The method O'Neil [O'N05] describes can produce nice results but is just suitable for rendering planets. *Rendering Outdoor Light Scattering in Real Time* presented by Hoffman [HP03] results in some artifacts.

**Clouds** can be rendered in 3D, 2D and 2.5D which is a faked 3D effect. As the workers will, hopefully, be most of the time concentrated on their work on the ground, clouds do not have to be 3D. A 2.5D effect like presented by Dube [Dub05] gives nice results but is to this time computationally expensive for the GPU.

Adding **rain** to a scene can add realism to a *bad weather scene* and force the worker react more responsible and with the utmost caution. Realizing a particle system for the rain on the GPU is possible and efficient but in Starik [SW99] subjects could not differentiate real rain scenes with post-processed mixed rain scenes. Thus, a 2D effect like described in [Wan05] is sufficient and realistic to add rain to a scene. Additionally, adding rain drops splashing on the machines window to create a more credible rain can improve the realism.

# Chapter 3.

# Approaches for Soil and Weather Condition Rendering

In the previous chapter we reviewed existing simulators for construction machinery. The observation is that most of them provide functional realism but lack in photo realism. To achieve a more realistic environment we reviewed some techniques and methods that can be used to render specific materials and conditions we are interested in.

In this chapter we want to give a better understanding of how we use the techniques reviewed in the previous chapter to approach soil and weather conditions. In every section we present the real world model of such a material and describe the physical, analytical model that exists today, if there is one, to calculate the effect.

In the first section we give a better understanding of soil and how it is defined and perceived. Then we describe a method that consists of several techniques to approach the appearance of soil.

The second section is about weather conditions and which factors play a role for certain weather conditions. In every subsection we discuss those factors and describe the existing analytical models to approach those effects.

## 3.1. Soil Rendering

### 3.1.1. Soil in the Real World

Soil is defined as a thin surface overlying the bedrock of most of the land area of the Earth. Weathering causes rock to be broken down and with material exchange through interaction with the environment, soil develops. Soil contains water, minerals, organisms and gases.

The formation of soil is complex, varies and do not merely consist of random assemblages of particles. *Horizons* in soil are known as a vertical section through the soil. The developed

soil depends on many local factors and the time over which the soil has been forming. Not every horizon will appear in every soil; some soils may have few or very indistinct horizons while others have clear, well-defined horizons [Wik05] [Mic05].

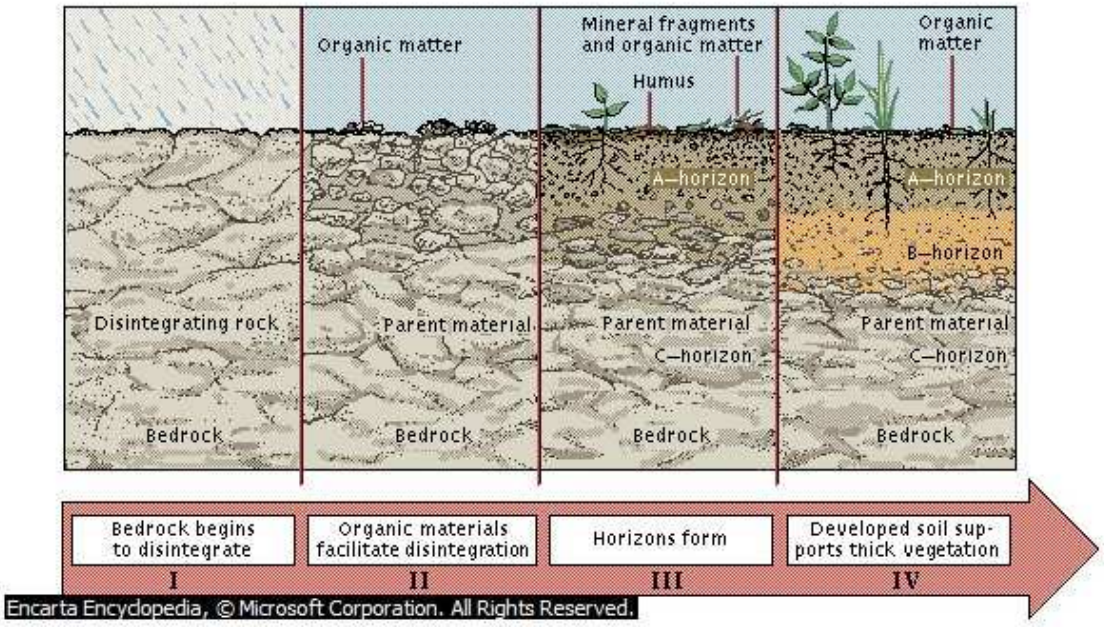


Figure 3.1.: Stages of Soil Formation [Mic05].

In Figure 3.1 the different stages of soil formation are represented. In the first stage (I) the bedrock starts to disintegrate as it is subjected to environmental forces like freezing-thawing cycles, rain, etc. From broken down rocks follows the parent material and in turn the smaller mineral particles (II). The organisms in an area contribute to soil formation by causing it to disintegrate as they live and adding organic matter to the system when they die. As soil continues to develop, layers called horizons form (III). The A horizon, nearest the surface, is usually richer in organic matter, while the lowest layer, the C horizon, contains more minerals and still looks much like the parent material. The soil will eventually reach a point where it can support a thick cover of vegetation and cycle its resources effectively (IV). At this stage, the soil may feature a B horizon, where leached minerals collect.

Summarized, there are some factors that influence soil formation:

- Parent material
- Climate
- Biotic factors; influence of plants and animals
- people

- topography (relief)
- time

Now it is important to know how soil looks like. In fact, it is important to have knowledge about the texture of soil which are mainly sand, loam, and clay as the three types of soil in terms of soil's texture. In Figure 3.2 sandy and loam soil are represented.


Figure 3.2.: Sandy Soil (right hand) and Loam Soil (left hand) [Mic05].

As the type of soil changes with the depth, it is necessary to have a look at the profile of soil. Research on soil types showed that a number of major soil types with distinctive profiles exist. These are found over wide areas, generally relate to the world's biomes, and are closely linked to the climate and the climax vegetation of the region. [Mic05].

In Figure 3.3 a profile of an idealized soil shows different horizons, each with distinct physical and chemical characteristics. In a non idealized profile clear horizons are prevented by the activity of soil organisms, particularly earthworms.

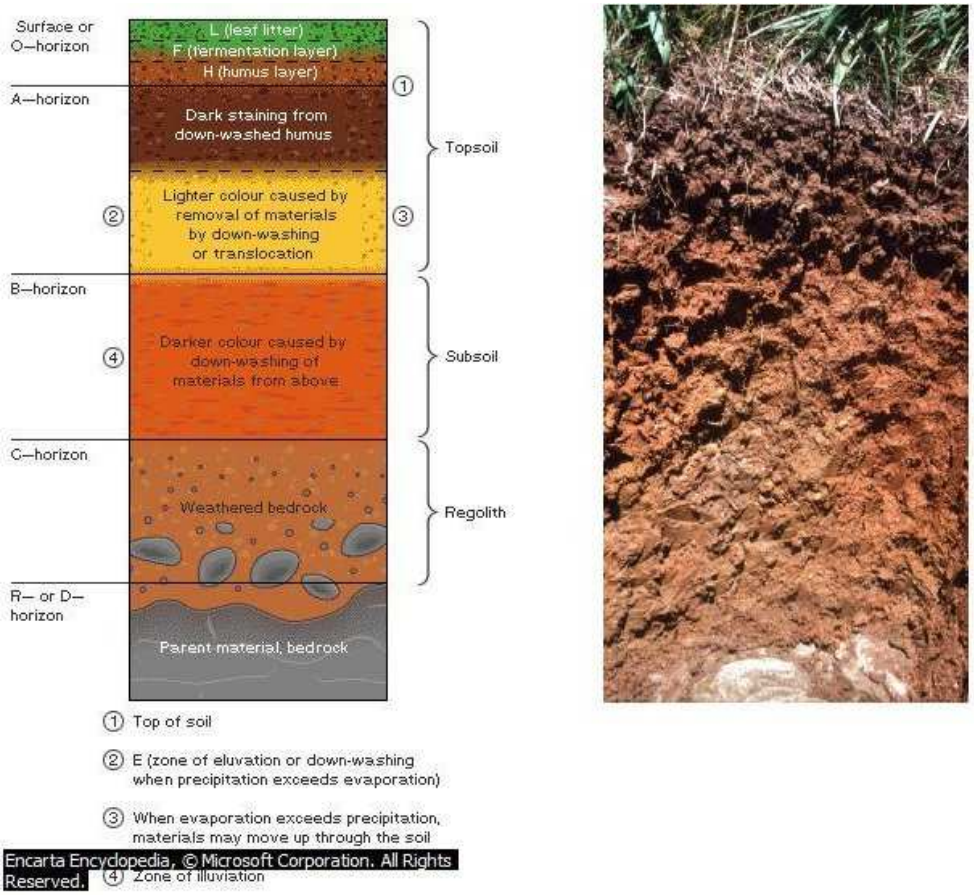


Figure 3.3.: Profile of an Idealized Soil [Mic05] and a photograph of real soil profile.

### 3.1.2. Soil in the Virtual World

As showed in the last section, soil has two major characteristics: first it is either coarse granular like a big piece of clay or it is fine like sand, and second it has different color on different layers in every horizon. Initially, we focus on the granularity of soil surfaces and then we show an approach on how to render dynamic, colored horizons for a realistic soil profile.

For simulating the granularity of a surface we apply the method described by William Donnelly [Don05] called *Per-Pixel Displacement Mapping with Distance Functions*. Using this method we are able to simulate high frequency details, like seen with a sandy surface, as well as coarse surfaces. To understand why the algorithm works, it is useful to firstly describe another approach for simulating low frequency details, called *Parallax Mapping* first presented by Tomomichi Kaneko et al. [Tom01] and extended later by Terry Welsh [Wel04] to *Parallax Mapping With Offset Limiting*. Some background information about *tangent space* and *bump mapping* is needed for describing *parallax mapping*.

**Tangent space** defines a coordinate system that is oriented relative to a given surface. It has three axes, $\vec{T}$, $\vec{B}$ and $\vec{N}$, the tangent, binormal and normal, respectively. The normal $\vec{N}$ lies perpendicular to the surface while the tangent $\vec{T}$ and binormal $\vec{B}$ lie in the plane of the surface. The tangent space rotates with a curved surface as the transformation applies per vertex. Now, to apply bump mapping, the light source direction $\vec{L}$ is transformed into tangent space at each vertex. To find the tangent space vectors at a vertex $\vec{N}$ is set to the vertex normal and $\vec{T}$ set to the gradient in the object coordinate system. The cross product of $\vec{N}$ and $\vec{T}$ gives the binormal $\vec{B}$ and thus an orthonormal basis. With an orthonormal basis it is possible to transform from one space to another, i.e. transform into tangent space:

$$\begin{pmatrix} x' \\ y' \\ z' \\ w' \end{pmatrix} = \begin{pmatrix} T_x & T_y & T_z & -L_x \\ B_x & B_y & B_z & -L_y \\ N_x & N_y & N_z & -L_z \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ w \end{pmatrix} \tag{3.1}$$

In Figure 3.4 can be seen what happens when a texture is mapped onto a flat polygon instead of on the real surface. The observer sees the actual texel, which is the flattened texel from A.

The **Parallax mapping** technique gives an approximation of a corrected texture coordinate by making a look-up in a height map and offsetting texture coordinates either away from the eye when it corresponds to a high area, or toward the eye if it is a low area. The offset calculation follows these steps:

1. Transform the eye vector into tangent space and normalize it.

2. Get the height value for the current texel from height texture.
3. Scale and bias the height value in order to map the height value from $\{0.0, 1.0\}$ to a better representation of the surface (see [Wel04]).
4. Offset is then a vector parallel to the polygon surface from given surface point to eye vector.

In Figure 3.5 the corrected texel would be calculated by:

$$T_n = T_o + (h_{sb} \cdot V_{x,y}/V_z) \tag{3.2}$$

and in Figure 3.6 the offset is calculated with a limitation [Wel04]:

$$T_n = T_o + (h_{sb} \cdot V_{x,y}) \tag{3.3}$$

Thus, combining parallax mapping and bump mapping gives a good result for self shadowed surfaces with depth but unfortunately the calculated offset is not correct and can not handle high frequency details.

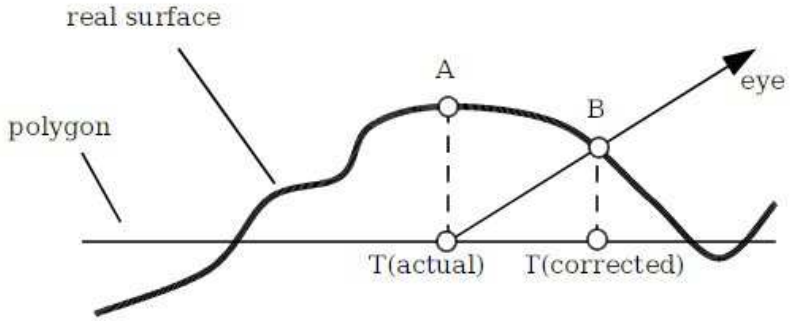


Figure 3.4.: The actual texel observed by the viewer is incorrect. Optimal solution would be the corrected texel as if it would have been projected to the real surface rather than the flat polygon [Wel04].

In the last chapter we reviewed some more advanced methods for calculating the real offset. One performant method was about finding the accurate intersection point from the eye vector with the height map using ray tracing. Donnelly [Don05] uses sphere tracing and a preprocessed volume texture for accelerating the tracing and to prevent loosing high frequency details, which would probably be skipped if the step size of the tracer is too big. Figure 3.7 shows an example where the tracer skips the peak in the height map.

The volume texture contains for each point $p$ in the tangent space and a surface $S$ the shortest distance:

$$dist(p, S) = min\{d(p, q) : q \epsilon S\} \tag{3.4}$$

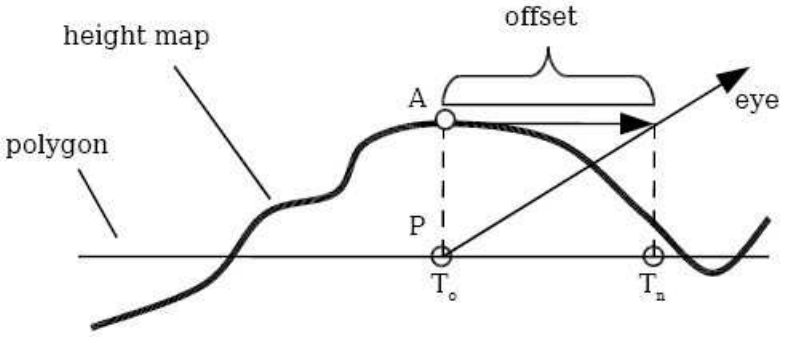


Figure 3.5.: Calculating the correct texel with an offset [Wel04].

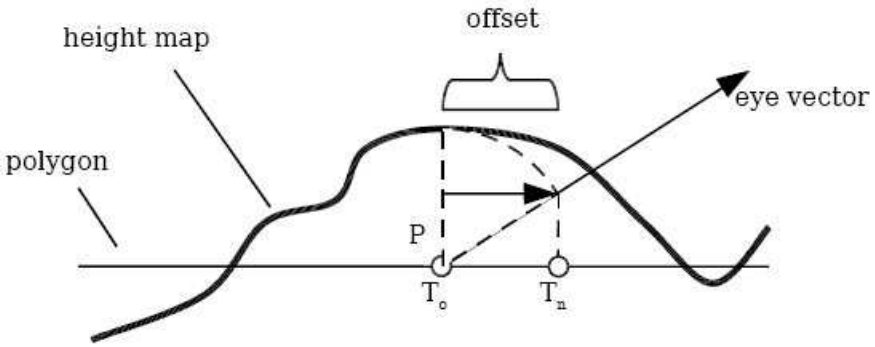


Figure 3.6.: More accurate with offset limitation [Wel04].

where $d$ is a function returning the distance from point $p$ to point $q$ in tangent space. Figure 3.8 shows a slice trough the 2D hight map and the 3D distance map. With the distance map it is possible to get the correct step size for the ray tracer at each point in tangent space. Donnelly [Don05] has a good example for this: Suppose there is a ray with origin $p_0$ and a normalized directional vector $v$. Using the distance information the new position $p_1$ is:

$$p_1 = p_0 + dist(p_0, S) \cdot v \tag{3.5}$$

and the position $p_2$ is:

$$p_2 = p_1 + dist(p_1, S) \cdot v \tag{3.6}$$

for the position $p_n$ it is:

$$p_n = p_{n-1} + dist(p_{n-1}, S) \cdot v \tag{3.7}$$

With enough samples this method converges very fast toward the closest intersection of the ray with the surface, which is what is needed for calculating the correct offset value. In Figure 3.9 the algorithm is illustrated and shows how fast it can converge. In fact 8-16 samples are sufficient for most cases and is recommended by Donnelly [Don05]. The distance

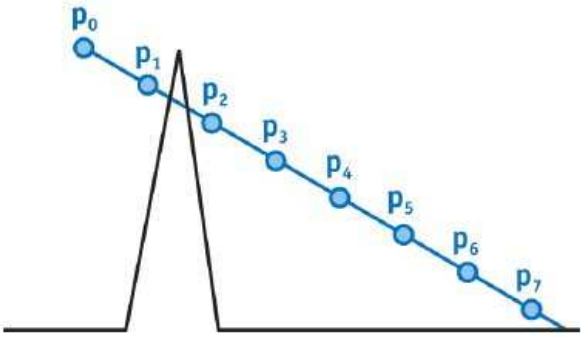


Figure 3.7.: The tracer misses the detail because of too large step size [Don05].

map can be computed with *Euclidean Distance Mapping* presented by Danielsson [Dan80] and runs with $O(n)$ where $n$ is the number of pixels in the height map.

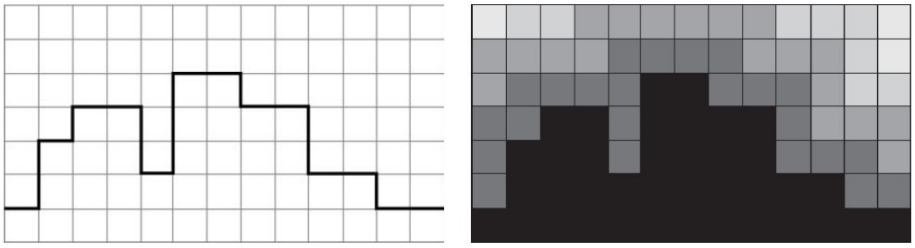

Figure 3.8.: A slice through the 2D height map and the corresponding 3D distance map [Don05].

Once having simulated the granularity of soil with the method discussed above it is necessary to apply a sophisticated material to it.

Coloring the soil depending on the depth can be done in two ways: Firstly by defining a height map with height values from the surface and second using texture coordinates to store the height value. In the first approach a texture has to be regenerated every time the surface is modified, and for large surfaces the amount of storage and transfer throughput can be a bottleneck.

In the second approach used in this work, the $y$ position of the vertices has to be normalized and stored as texture coordinates. Once the height values are in the range between $0.0, 1.0$ they can be used to look-up in a color ramp to get the actual color. Again, all that has to be defined is a normalization factor, which can be different for every mesh, and one or more color ramps/gradients, which define the color of the horizons. Figure 3.10 illustrates the look-up with a height map.

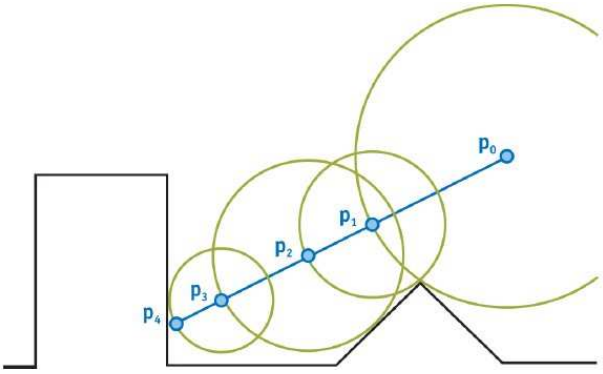


Figure 3.9.: The sphere tracing algorithm converges fast toward an intersection point with the surface [Don05].

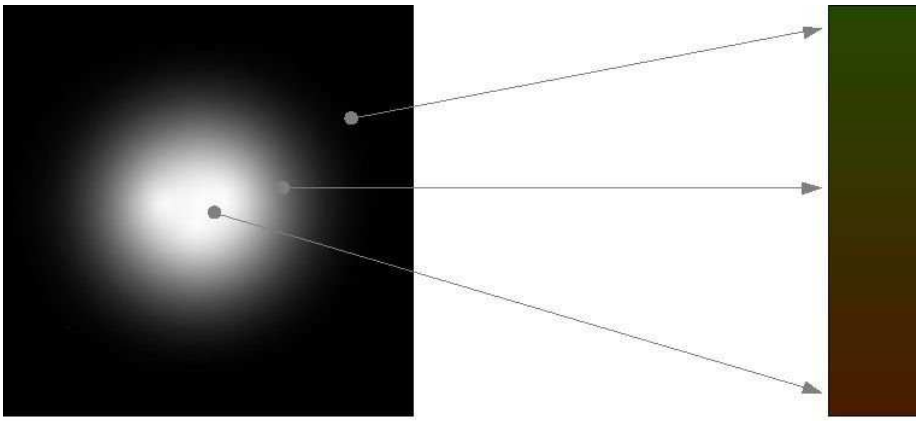

Figure 3.10.: Look-up from a height map into a color ramp.

## 3.2. Weather Condition Rendering

Weather is a composition of some physical characteristics in our atmosphere. Those characteristics can be summarized as:

- sunlight and atmospheric scattering (including fog and dust),
- clouds
- rain

This section introduces mathematical methods and approaches based on observations to visualize those effects.

### 3.2.1. Sunlight and Atmospheric Scattering

Accurate atmospheric scattering is maybe the most important impression of any outdoor scene. It is responsible for the appearance and quantity of light reflected from the scene that reaches the eye, or how it is perceived from the sky. The human eye contains two different types of photo **receptor cells** (neurons) [Wik05] that are responsible for converting light into signals that can be interpreted by the human brain from a combination of red, green and blue color and a luminosity. Rod cells are the type of cells that function in less intense light and are primary source of visual information in dark scenes, whereas cone cells only function in relatively bright light and are mainly responsible for the human color perception. Normally, there are three kinds of cones which have different response curves (see Figure 3.11).

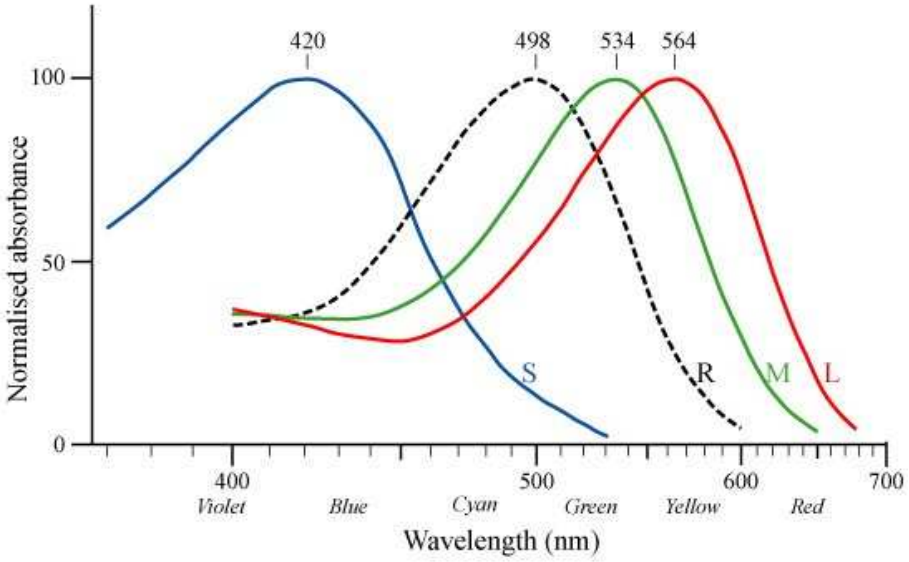


Figure 3.11.: Spectral absorption curves of the short (S), medium (M) and long (L) wavelength pigments in human cone and rod (R) cells [Wik05].

**Sunlight** contains light that is in the range of visible wavelengths of human perception (400 - 700 *nm*) and is a combination of all visible wavelengths in nearly equal intensities. When the sun is close to the zenith, the peak intensity of the sun's electromagnetic energy lays slightly in yellow. That's why the sun appears white and yellowish on a sunny and clear day.

**Atmospheric scattering** will both add and remove light and color from a viewers *eye ray*. This atmospheric effect causes distance objects getting brighter and loose their contrast which is a very important aspect for the human ability to guess distances [Nie03]. Different particles in the atmosphere scatter light in different ways, which can also result in fog or dust, if the particle size is large and the appearance high. Two common models for

scattering in the atmosphere are *Rayleigh Scattering* and *Mie Scattering*, where the latter can be approximated by the *Henyey-Grenstein Phase Function*.

**Rayliegh Scattering** is the model of atmospheric scattering, which refers to scattering caused by molecules scattering light more heavily at shorter wavelengths (Figure 3.12). The phase function $\beta^{\lambda}(\theta)$ describing the amount of light at a given wavelength $\lambda$ scattered in a given direction $\theta$ is given by [Nie03]:

$$\beta^{\lambda}(\theta) = \frac{\pi^2(n^2-1)^2}{2N\lambda^4}(1+\cos^2\theta) \tag{3.8}$$

where $\theta$ describes the angle between the view angle and the sun direction, $n$ is the refractive index of air, $N$ is the molecular density and $\lambda$ the wavelength of light. The most important part of the Rayleigh scattering phase function is the $\frac{1}{\lambda^4}$ dependency of wavelength as this results in a property that shorter wavelengths are scattered much more than longer wavelength, which is in turn the main reason why the sky is blue. The color of the sun and sky turn to yellow, orange or red at sunset, because as light travels far through the atmosphere, the probability that longer wavelengths get scattered away before the light ray reaches the eye is much higher.

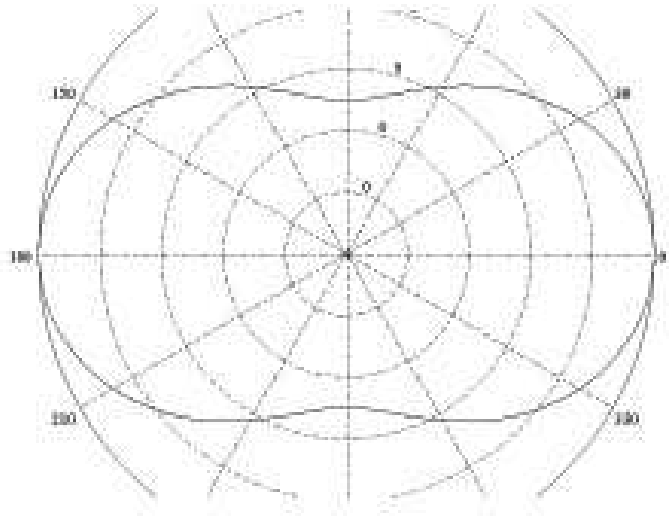

Figure 3.12.: Rayleigh phase function in polar coordinate space.

**Mie Scattering** is used for calculating scattering for larger particles ($r \geq \frac{\lambda}{10}$) [Nie03], such as dust and pollution (aerosols), but is applicable for simulating scattering caused by particles of any size. If assuming a certain size of the particles, the Mie scattering function

can be written as [Nie03]:

$$\beta(\theta) = 0.434c\frac{2\pi}{\lambda}^{v-2}0.5\beta_M(\theta) \tag{3.9}$$

Where $c$ is the concentration factor that varies with turbidity $T$ and is:

$$c = (0.6544T - 0.6510) \cdot 10^{-16}$$

and $v$ is Junge's exponent with a value of 4 for a sky model [PSS99]. $\beta_M$ describes the phase function and gives the shape of the phase function (Figure 3.13). Preetham [PSS99] gives a table for the *Mie Scattering Phase Function*.

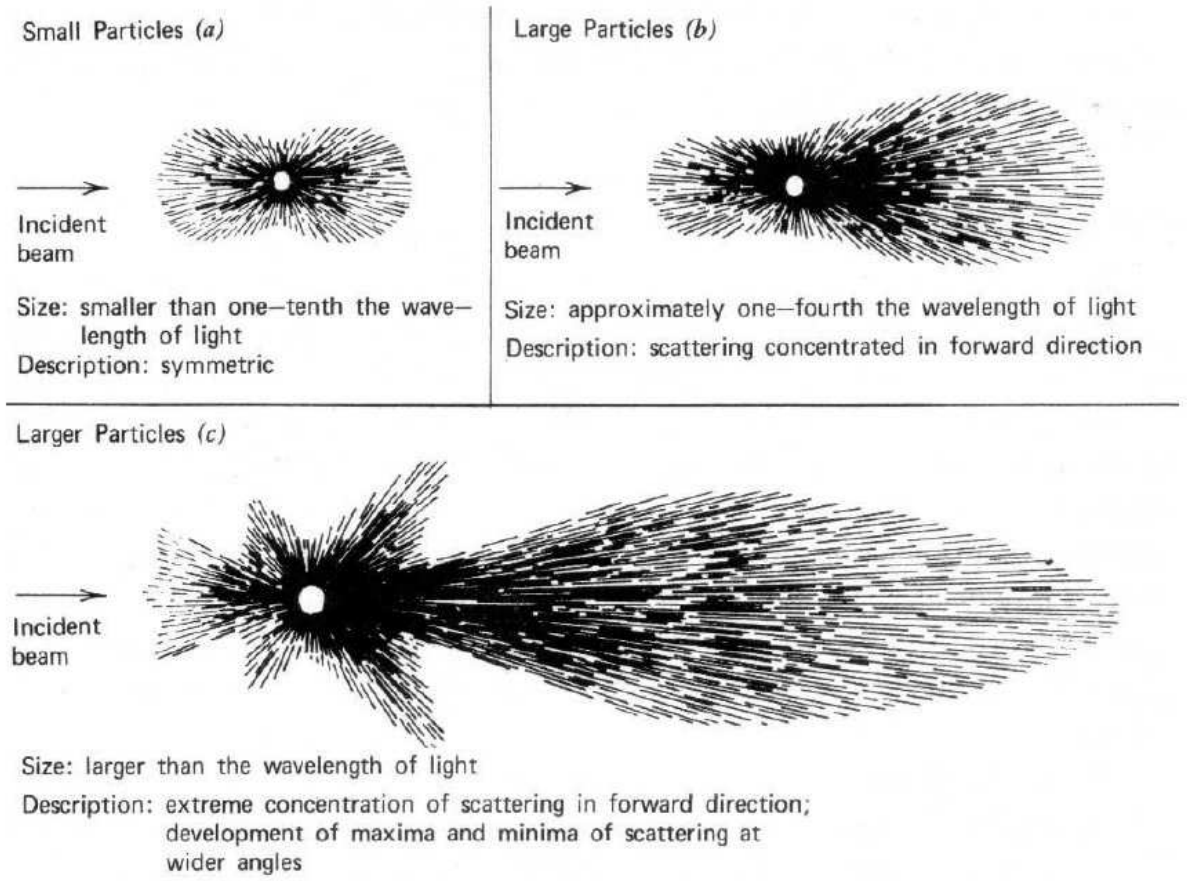


Figure 3.13.: *Mie Phase Function.* The top left image shows that *Rayleigh Scattering* is a subset of *Mie Scattering Phase Function* [Nie03].

Another Phase Function is the **Henyey-Greenstein Phase Function** that is used to approximate the complex Mie theory [Nie03]:

$$\Phi_{HG}(\theta) = \frac{1 - g^2}{4\pi(1 + g^2 - 2g\cos\theta)^{\frac{3}{2}}} \tag{3.10}$$

Where $g$ is the directionality factor. Figure 3.14 shows how the shape of the phase function differs with different $g$ values.

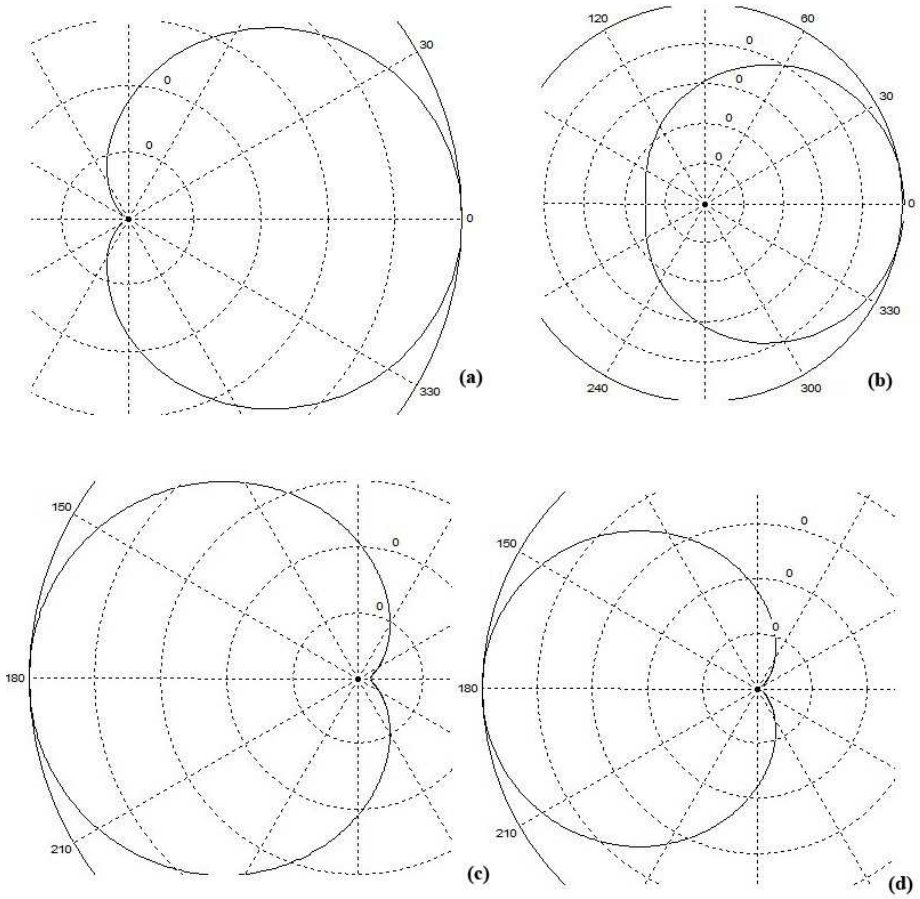


Figure 3.14.: The shape of the *Henyey-Greenstein Phase Function* with different g: g=-0.99 (a), g=-0.2 (b), g=0.9 (c), g=0.5 (d).

In this work we use an adaption of the Henyey-Greenstein phase function previously used in [NSTN93] and [O'N05]:

$$F(\theta, g) = \frac{3(1-g^2)}{2(2+g^2)} \cdot \frac{1+\cos\theta}{(1+g^2-2g\cos\theta)^{\frac{3}{2}}} \tag{3.11}$$

Where $g$ again describes the directionality factor, and by setting $g$ to 0, Rayleigh scattering can be approximated. $g$ is usually set between -0.75 and -0.999 for simulating Mie aerosol scattering.

The integral of the scattering coefficient of all sub elements $ds$ of a given path results in the **optical depth**:

$$T = \int_S \beta(s) ds \tag{3.12}$$

Where $\beta(s)$ is the combined Mie and Rayleigh total scattering coefficients that can vary from day to day and with different altitudes. This can be used to calculate the attenuated spectral distribution arriving at the observer after passing the atmosphere over a path:

$$I = I_0 \cdot e^{-T} \tag{3.13}$$

Where $I_0$ is the incident spectral distribution. The result can be described as a weighted factor based on how many air particles are in the path. Thus, this is where natural phenomenons like fog and dust can be simulated. The fog that can be simulated using the fixed function hardware can be summarized as [HP03]:

$$I = I_0(1 - f) + C_{fog}f \tag{3.14}$$

Where $f$ is the fog factor and $C_{fog}$ is the fog color, which is only an approximation as the multiplicative factor is monochrome and the color of the additive factor or intensity does not change based on the viewing direction.

In [O'N05] the optical depth is extended to an **Out-Scattering Equation**:

$$t(P_aP_b, \lambda) = 4\pi K(\lambda) \int_{P_a}^{P_b} e^{\frac{-h}{H_0}} ds \tag{3.15}$$

Where the integral determines the optical depth across a ray from point $P_a$ to $P_b$ multiplied by the length of the ray (Figure 3.15). The variable $h$ is the height of the sample point and $H_0$ is the scale height, which is the height at which the atmosphere's average density is found. The value of $K(\lambda)$ is the scattering constant and depends on the chosen phase function.

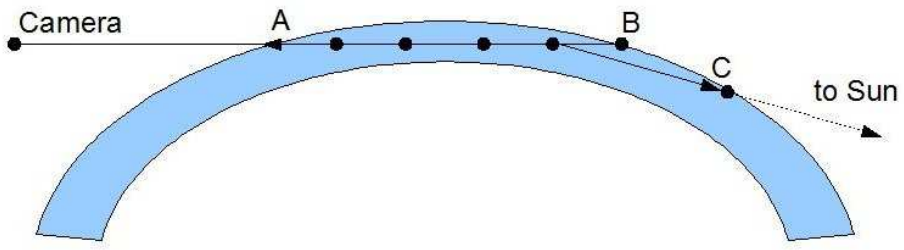


Figure 3.15.: Solving the scattering equations as described in [O'N05] and [NSTN93]: Define a line segmented from point A to B. Approximate the scattering integral by solving the equation at sample positions. In this figure the last sample position is considered. Sunlight comes directly from the sun to that point and gets scattered toward the camera, from which some light is again scattered.

The last thing to be calculated is the **In-Scattering Equation**, which describes how much light is added to a ray through the atmosphere due to light scattering [O'N05]:

$$I_v(\lambda) = I_s(\lambda)K(\lambda)F(\theta, g) \cdot e^{\frac{-h}{H_0}} \int_{P_a}^{P_b} e^{-t(PP_c,\lambda)-t(PP_a,\lambda)} ds \tag{3.16}$$

Where $I_s(\lambda)$ is the intensity of the sunlight and does not have to be dependent on the wavelength, and $F(\theta, g)$ is the *Henyey-Greenstein Phase Function* described in Equation 3.11.

Additionally, the **Surface-Scattering Equation** is used in [O'N05] as a model to calculate the scattered light reflected from surfaces, such as mountains or other objects in the atmosphere. This model takes into account that some of the reflected light will be scattered away on its way to the eye/camera and some extra light is scattered in from the atmosphere:

$$I_v^{'}(\lambda) = I_v(\lambda) + I_e(\lambda) \cdot e^{-t(P_aP_b,\lambda)} \tag{3.17}$$

Where $I_e(\lambda)$ is the amount of light emitted or reflected from a surface and is attenuated by an out-scattering factor. Because the sky is not a surface and thus can not reflect or emit light, $I_v(\lambda)$ is sufficient to render the color of the sky.

In [O'N05] O'Neil discusses how poorly these equations would perform with a brute force rendering algorithm. With five sample points the number of calculations would be about 3.000 per vertex! With some improvements on look-up tables the number decreased to 60, but it still requires newest hardware with Shader Model 3.0. After some graphical analysis O'Neil could eliminate the look-up table and use some polynomial functions (for detail description please see [O'N05]). As for this work it was not important to render atmosphere from the space or different altitudes, we could remove most of the computation responsible for calculating the $h$ and $H_0$ parameters and replaced them with parameters that affect the look of the sky and atmosphere in an intuitive way (see subsection 4.3.1).

### 3.2.2. Clouds

Clouds are visible mass of ice crystals or droplets suspended in the atmosphere. Thus clouds are also particles reflecting and scattering light like smaller particles described above. But clouds scatter all visible wavelengths of light equally and thus are usually white, but they can also appear gray or even black if the density is high enough, so that sunlight can not pass trough. Clouds are quite heavy and keep typically up to several million tonnes of mass. But the single particles are so small and lightweight, that currents can sustain them in the air [Wik05].

Clouds form when invisible water vapor rises, cools in the air, and condenses out of the air as droplets. The form of the clouds varies and depends on the strength of the up lift and air stability. In unstable conditions the clouds are formed vertically, and in stable air conditions horizontal clouds are formed.

Clouds can be classified by their altitudes and shape (see Figure 3.16), and are divided into two general categories: layered and convective, and are named stratus clouds and cumulus clouds. Respectively these two types are again divided into four groups that distinguish in their altitudes [Wik05].

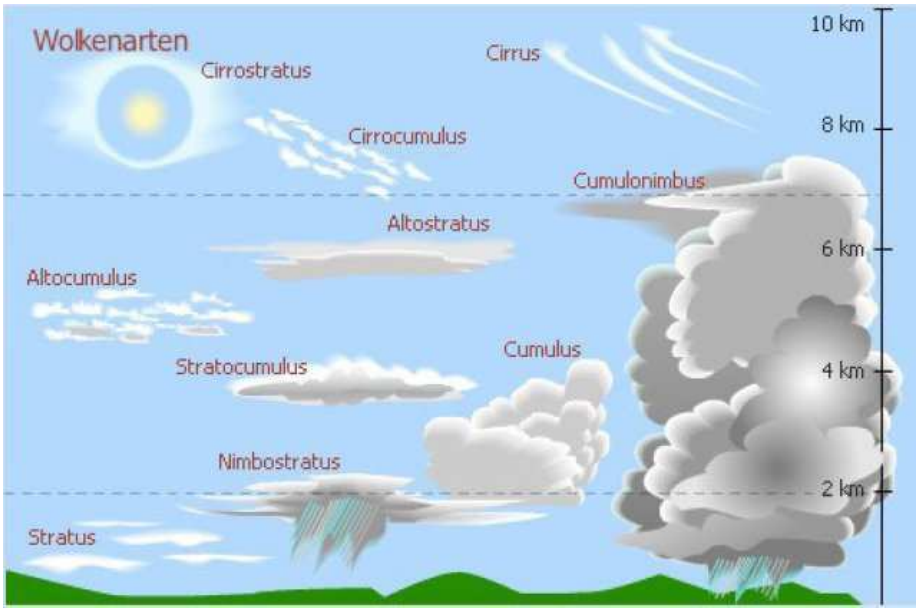


Figure 3.16.: Cloud classification by altitude of occurrence [Geh05].

The color of clouds is mostly white and gets gray and black if the thickness of the clouds is very high. Also, droplets may combine to produce larger droplets, which may themselves combine and form a droplet that is big enough to fall as rain. This accumulation permits most of the sunlight from being reflected back out before it is absorbed which is the reason why the color of the cloud gets gray or black and the surface following the light direction gets darker and shadowed. Other colors are more natural effects of light scattering within the cloud (e.g. builsh-grey clouds) or are the result of reflecting the long and unscattered rays of sunlight due to scattering in the atmosphere (e.g. red, orange or purple clouds in Figure 3.17).

In our case it was not important to have 3D clouds as the viewer will never pass through the clouds or be higher than any cloud layer. Thus approximating clouds with animated noise and correct shading, like described in [Dub05], is the most applicable method. The texture is a weighted octave composition of eight **noise textures** (e.g. perlin noise). The noise textures are generated with random numbers and smoothed with simple neighborhood filtering. The texture size can be $128x128$ for having good performance and still good visual quality. The textures are then added together using different weights and scales to achieve the desired effect. The composition starts with a non scaled first octave containing low frequency data and thus the main shape of the clouds. Then the second layer is scaled

Figure 3.17.: Different cloud colors [Wik05].

to be more repeated but weighted less than the previous layer. As more layers are added, they are scaled to repeat more but weighted less, which gives more and more detail as the higher octaves contain high frequency data:

$$\psi(n) = \sum_{n} texture_{uv*2^n} * \frac{1}{2^n} \tag{3.18}$$

Where $n$ is the number of octaves, $texture$ is a texture look-up and $uv$ is the texture coordinate in UV space. $\psi$ is the composition of all layers.

The appearance of the clouds can be controlled allowing different weights and scales. Increasing weights in higher octaves results in smaller clouds, and a higher repeat factor results in faster changing shapes. By animating/offsetting the high frequency detailed textures more than the low octave ones, the main shape of the cloud changes slowly while the fluffy part of the cloud changes more frequently. Moreover, fading slowly between different noise textures results in credible change of cloud shapes.

The **cloud density** is controlled by subtracting a value from the composition and clamp the result to zero, which removes a certain quantity of noise from the composition:

$$\vartheta(\psi, \xi) = \begin{cases} 0.0 & : \quad (\psi - \xi) \leq 0.0 \\ 1.0 & : \quad (\psi - \xi) \geq 1.0 \\ (\psi - \xi) & : \quad else \end{cases} \tag{3.19}$$

Lastly, the result is exponentiated to get a fluffy or hard shaped cloud layer (see Figure 3.18) [Dub05]:

$$\chi(\eta, \vartheta) = \eta^{\vartheta} \tag{3.20}$$

$\vartheta$ is the density of the clouds, $\xi$ ranges from $[0, 1]$ and gives a rate for how much sky is covered by the clouds, $\chi$ is the exponentiated value of the clouds, and $\eta$ is the sharpness/fluffyness value of the clouds and is in the range [0,1] [Dub05].

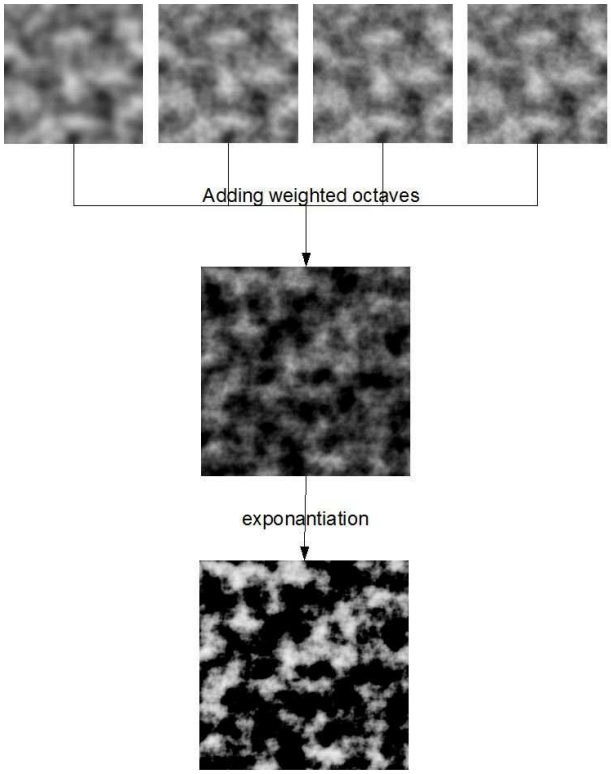


Figure 3.18.: Cloud Rendering: different weighted noise octaves are added together, truncated and exponentiated.

Now, instead of lighting the clouds with correct physical scattering terms, which is difficult to calculate in real time or even can leave much to be desired [HL01], Dube [Dub05] described a way to approximate the desired look. As discussed above clouds are usually white and get darker when small amount of light passes through the cloud. The previously generated density texture can be used as a density field and the sun is proposed to be a point light. Sunlight is then traced trough the density voxels to find the correct darkening (see Figure 3.20).

The reason why point light and not directional lighting, which would be the most physical correct lighting technique, is used is that directional light would cause a flat shading. The problem is illustrated in Figure 3.19. When the light is treated as directional light, the red voxel in cloud A and cloud B are shaded the same way, because they have the same thickness, shape and are in fact 2D. Real clouds are 3D and thus from a viewers point V cloud B would have a different shading i.e. most of the red voxel would be hidden by the side of the cloud. This 3D effect can be approximated by treating the sunlight as a point

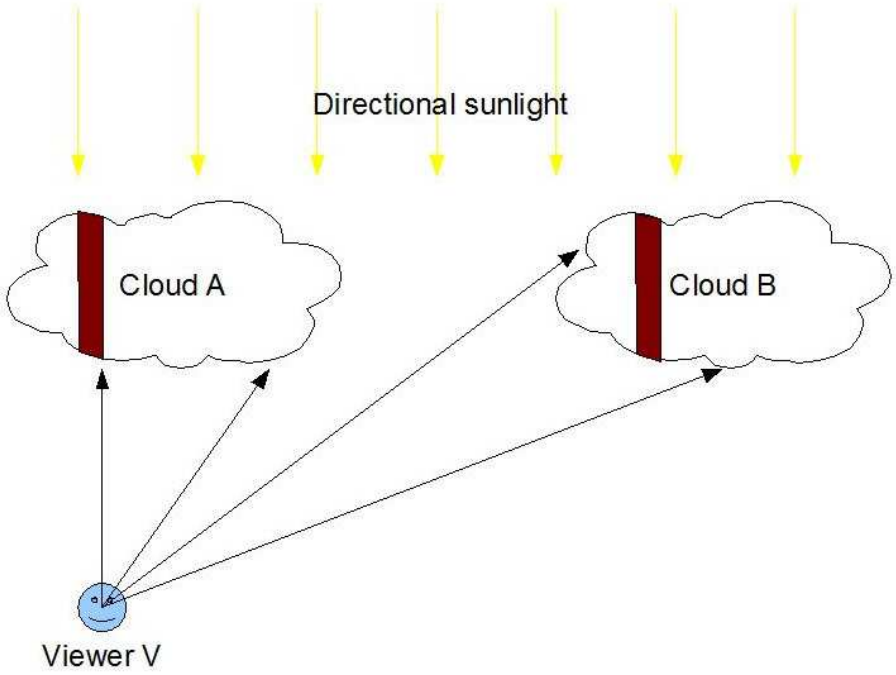


Figure 3.19.: Clouds are 3D and thus directional sunlight would be the physical correct way to shade the clouds.

light, which is illustrated in Figure 3.20. The result is that the voxel that was shaded in both clouds nearly black, as the light had to pass through the whole cloud, is now shaded black in cloud A correctly, but shaded nearly white in cloud B, as it now passes just a small amount of cloud.

Another way to shade the clouds is to use **direct illumination** by calculating the normal at every pixel and do a dot product with the sun direction. Although, this is a bad approximation, it is much faster than the path tracing and the result is fairly good, if the visualization does not care about covering the whole sky with clouds. The problem with this method is shown in Figure 3.21, where the clouds seem to be flat. The reason is the dot product, that creates an even shading based on the normals computed.

In order to improve the shading of clouds, still with keeping the rendering performance in mind, we adopted another approach. The idea is based on using a second cloud texture adding it as a layer to the previously generated $\chi(\eta, \vartheta)$ and multiplying it with $\frac{1}{2}$, which is the average of both cloud layers:

$$result = (\chi(\eta, \vartheta) + extraTexture) \cdot \frac{1}{2} \tag{3.21}$$

The result of our appraoch is shown in Figure 3.22. Unfortunately, this approach does not

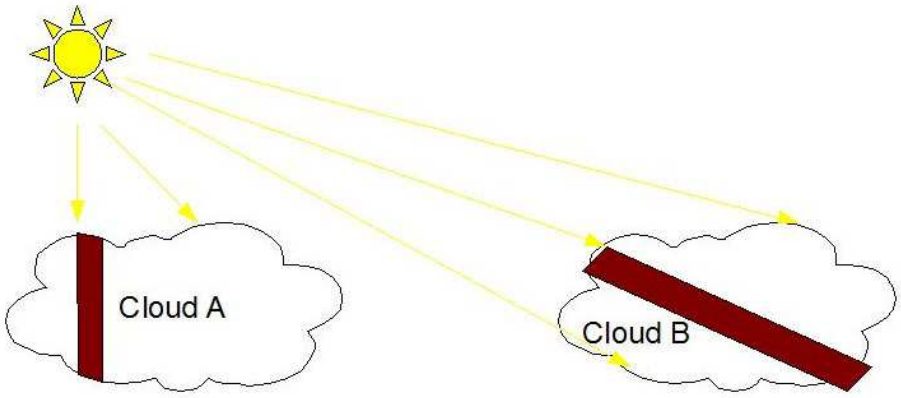


Figure 3.20.: The 3D effect can be approximated by treating the sunlight as a point light source.

generate realistic shading as it never takes the sunlight direction into account. However, this method is fast and fulfills the conditions of the VAR-Trainer project, where the worker is mostly focused on the ground. This method also generates better results for full cloud covered sky.

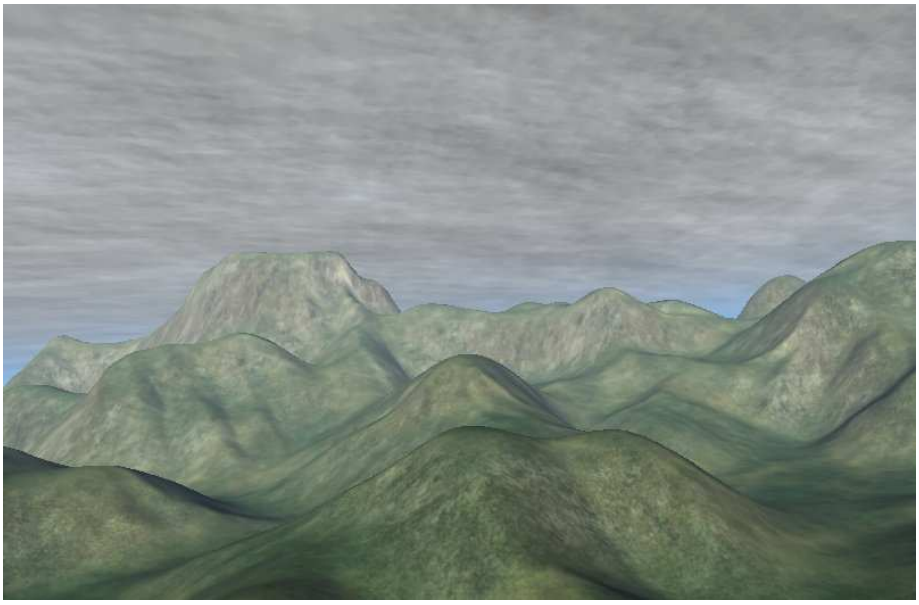

Figure 3.21.: Full covered sky with dot product for shading the clouds.

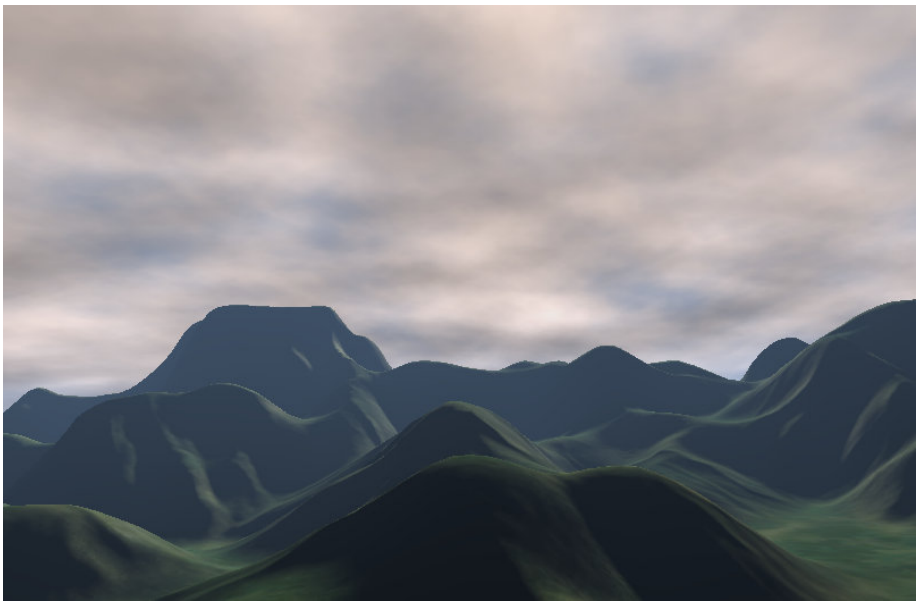

Figure 3.22.: Full covered sky with average of two cloud textures for shading the clouds.

### 3.2.3. Rain

Rain is a result of droplets in clouds that may combine to produce larger droplets, which may themselves combine and form a droplet that is big and heavy enough to fall down as rain. Rain is a form of precipitation, other forms of which are snow, sleet, hail and dew (Figure 3.23). Rain can add a high level of realism to an outdoor scene.

Figure 3.23.: Real rain scene: notice that the scene is just covered with a layer of random streaks, which is often what makes us see rain [SW99].

In [SW99] Starik and Werman showed that rain can be approximated with a 2D post effect for videos. Wang and Wade [Wan05] described a method to produce credible rain for 3D visualization. They came up with a solution that adds layers of textures with different size of rain and number of drops together, and animates them by translating the texture coordinate matrix in the desired direction. This is then mapped on a double cone surrounding the camera. The amount of how much the texture is scrolled in every frame is given by [Wan05]:

$$D_{precip} = S_{streak} \cdot \frac{t_\Delta}{t_{const}} \tag{3.22}$$

Where $t_{const}$ is a fixed frame time and describes how fast the texture should be scrolled (e.g. $\frac{1}{30}$ for 30 Hz. $S_{streak}$ is the factor handling how many rain drops should fall in one frame and $t_\Delta$ is the frame time. The scale of the rain is important to simulate motion blur:

$$E = \frac{S_{streak}}{S_f \cdot (C_{pixel} \cdot S_f \cdot Vec_{precip} + S_{streak})} \tag{3.23}$$

Where $S_f$ is a scale factor, $C_{pixel}$ specifying a world space scale factor for each pixel in the texture and $Vec_{precip}$ originally was designed for simulating camera movement at high speed [Wan05]:

$$Vec_{precip} = (C_f \cdot Vel_{camera} + Vel_{gravity}) \cdot t_\Delta \tag{3.24}$$

Where $Vel_{camera}$ is the camera velocity, $C_f$ is a factor for limiting the tilting of the cone and $Vel_{gravity}$ is the velocity of precipitation due to gravity.

All transformations can be put together in one matrix which then can be multiplied with the texture coordinate to have the desired rain effect:

$$\begin{pmatrix} S_f & 0 & 0 & 0 \\ 0 & \frac{S_{streak}}{S_f \cdot (C_{pixel} \cdot S_f \cdot Vec_{precip} + S_{streak})} & 0 & 0 \\ 0 & S_{streak} \cdot \frac{t_\Delta}{t_{const}} & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

As for the purpose of a construction machinery simulator it is not necessary to take care about high velocity and tilting we simplified the matrix to:

$$\begin{pmatrix} S_f & 0 & 0 & 0 \\ 0 & \frac{S_{streak}}{S_f \cdot (C_{pixel} \cdot S_f \cdot Vel_{gravity} \cdot t_\Delta + S_{streak})} & 0 & 0 \\ 0 & S_{streak} \cdot \frac{t_\Delta}{t_{const}} & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

We use some simplified parameters for the translation and scale as all that has to be transformed by this matrix is the offset in the direction toward the ground and the scale in $X$ and $Y$ direction. This results in the following transformation matrix:

$$M = \begin{pmatrix} \frac{1}{2 \cdot S_f} & 0 & 0 & X_{offset} \\ 0 & S_f & 0 & S_{streak} \cdot t \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{3.25}$$

Where $t$ is the current time and $X_{offset}$ is used to offset rain layers in $x$ direction, which is important for stereo views as we found out that in stereo/immersive view the double cone is visible. Thus, we decided to not only add one double cone to the scene. Adding four double cones where every double cone has double the size than the previous one results in a nice rain effect, even for stereo view.

Another effect, which makes the rain more credible, is rain drops splashing on objects, windows or puddles. We found it reasonable to concentrate on the effect of rain drops on windows as the targeted viewer position will be about three meters up from the ground viewing most of the time out of a windowed cabin. **Rain drops** in nature reflect a high specular shine and have a high transparency that refracts the light. To approximate this effect we first created a texture containing drops having a specular shine (Figure 3.24). Adding this texture to an object, and using the negative $(1 - texture_{color})$ of this texture as an alpha value for blending, gives a nice approximation without taking the effect of

reflection and refraction into account. A random appearance of the drops can be simulated by masking the texture with a random but smooth animated noise texture. By animating the drop texture coordinates toward the ground, a realistic movement and appearance of rain drops is achieved with minimum computation cost.

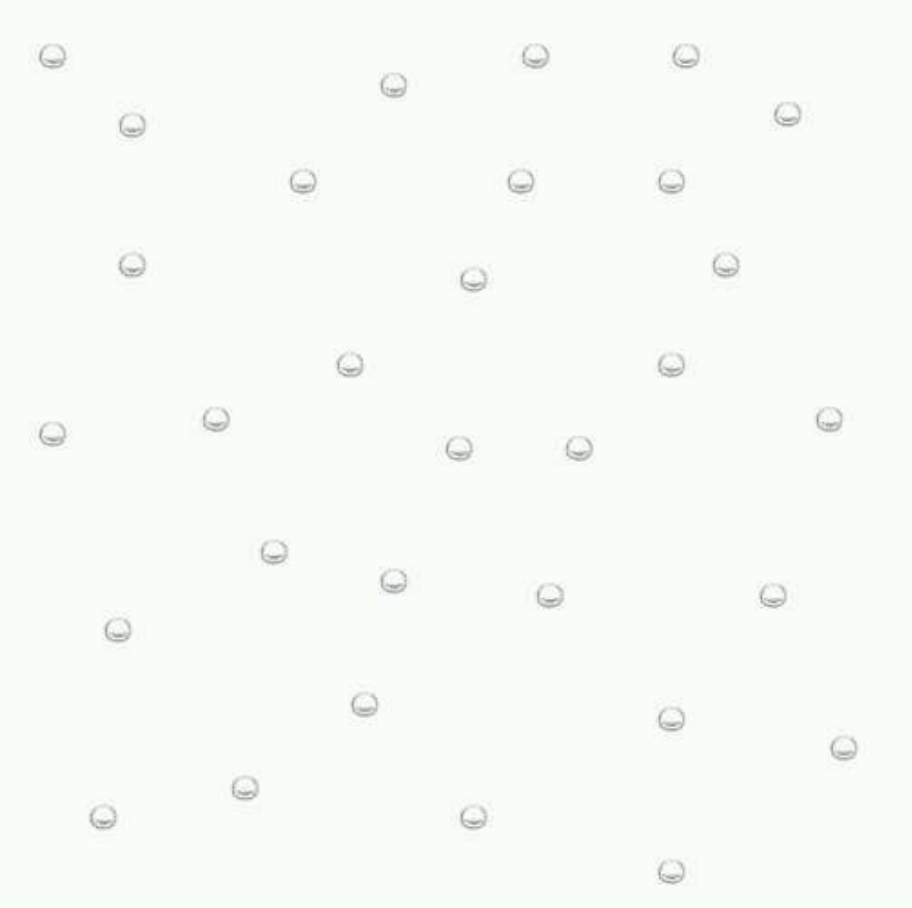

Figure 3.24.: Rain drops texture.

## 3.3. Summary

In this chapter we first described how soil can be approached with some recent techniques and newest hardware. The look of soil has two factors. The first factor is the granularity, which can be simulated with per pixel displacement mapping. This method described in [Don05] uses a sphere tracing technique with a pre-calculated volume texture to generate the correct parallax effect. Adding the bump mapping technique, the soil is able to produce correct self shadows. The second factor is the color of soil, which depends on the region

it comes from and varies in so called horizons. The color can be controlled by some pre-generated color gradients for look-up operations, i.e. depending on a depth value, the color gradient can be used as a look-up table.

The next important outdoor effects are related to the sky. There are many existing models on how to approximate the sky. Unfortunately, some of them are computationally too expensive for real time visualization, as for correct rendering of the sky and atmosphere, a scattering integral which takes all incoming and reflected light into account has to be calculated. For approximations usually two phase functions are used to calculate the scattering term that is used to render the sky, atmospheric effects like fog, dust and pollution. The first one, Rayleigh scattering, is related to small particles in the air and is responsible for the blue sky because it scatters smaler wavelengths. The second one, Mie scattering takes bigger particles into account and scatters all wavelengths equally, but is too expensive to be calculated in real time. Thus, Mie scattering is approximated by various adaption of the Henyey-Greenstein Phase Function. With this two models and some simplifications on the scattering integral made in [NSTN93] and [O'N05], sun, sky and atmospheric effects like fog or dust can be rendered in real time.

Clouds are an important part of a realistic look of the sky and any outdoor scene. Clouds again use the scattering integral as they are an accumulation of water droplets and smaller particles. Clouds can be visualized with animated noise textures mapped on a plane or sphere above the ground. The shading of the clouds in 2.5D is a very expensive operation, as a path tracing has to be done for every cloud texel. The shading can be approximated by direct illumination, which is a dot product between the sunlight direction and the gradient of every cloud pixel. Unfortunately, the direct illumination approach results in flat shaded clouds, if the clouds cover the whole sky. Thus, we came up with another approach where the cloud is shaded with an average of another animated cloud texture. This approach is much faster than all others and results in nice but not accurate cloud shading.

The last part of this chapter described an approach for rendering rain with animated texture coordinates and a rain texture. The mapped rain texture is shifted every frame to simulate falling rain. The object used for mapping is a double cone surrounding the camera, because the shape of a double cone provides a perspective view if the viewer looks up to the sky or down to the earth. Also, by tilting the double cone, velocity can be simulated. Here, we simplified the texture transformation matrix as for the purpose of a construction machinery simulator it is not necessary to take care about high velocity and changing up vector.

Rain drops on windows are simulated by masking a rain drop texture with an animated noise texture. This can lead to a higher level of photo realism of the rain.

# Chapter 4.

# Implementation

In this chapter we firstly give an introduction how to program a GPU with the new OpenGL shading language and how to integrate shaders (i.e. shaders specified in the last chapter) into an OpenSG render context. Focus will be on the Nvidia NV40 GPU series as it is the graphics hardware used in the VAR-Trainer project.

In the second part of this chapter we discuss our implementation, namely the C++ classes and their integration into an OpenSG application.

Lastly, we provide details about the shaders implemented in GLSL for rendering the effects introduced in chapter 3.

## 4.1. Software Environment

### 4.1.1. About GPU Programming

Graphics processor units are widely used today to drive cutting-edge 3D game engines, virtual reality simulations, and film pre-production. GPUs are defined as the microprocessor of a graphics card used in personal computers or in game consoles. In the late 1970s and 1980s the chips had usually no support for drawing shapes, but instead some of them had Bit Block Transfer (BitBLT) support in the form of sprites. The Atari 800 and Atari 5200 used a co-processor called ANTIC, which was an early example of an GPU that could run several operations in a display list and use DMA to reduce the load on the host processor. In the late 1980s and early 1990s general purpose microprocessors were used for implementing GPUs; e.g. Digital Signal Processors like the TMS340 for fast draw calls or PostScript raster image processors for laser printers [Wik05].

The Commodore Amiga and IBM's 8514 graphics system were one of the first systems implementing 2D primitives in hardware. They supported 2D acceleration through a regular frame buffer controller such as VGA. By the early 1990s, when Microsoft Windows

became popular, the interest to high speed, high resolution 2D bitmap graphics became more important than ever for PCs.

In 1991, S3 Graphics introduced the first single chip 2D accelerator, and in 1995 every major chip vendor supported 2D acceleration. In 1993, SGI introduced their RealityEngine™ [Ake93] graphics system, which was the first system primarily designed to render texture mapped, antialiased polygons. Parallel to the Reality Engine two other APIs were designed for rendering raster graphics: DirectX [Dir05] and OpenGL [OPE05a]. Since the advent of DirectX version 8 API and OpenGL 2, GPUs added programmable shading to their capabilities allowing to process each pixel and each vertex through a kernel or small program, allowing different levels of manipulating the transformation, lighting and coloring/shading of the input data.

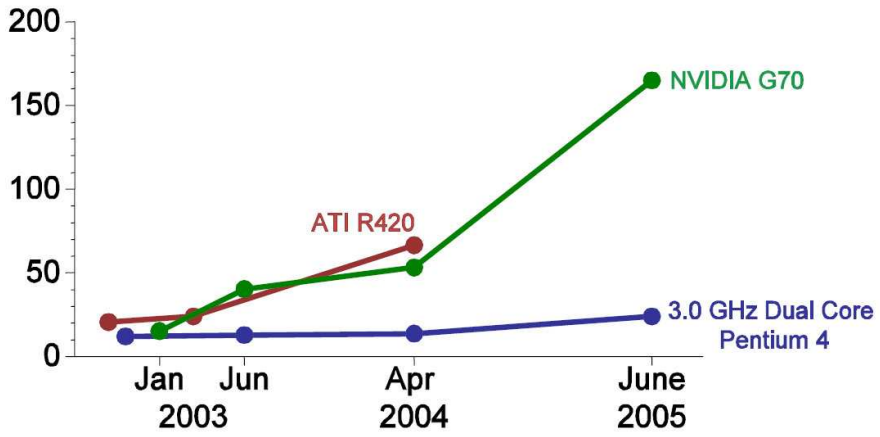


Figure 4.1.: GPU trends by Ian Buck. The metric of the y-axis is in Giga-FLOPS (GFLOPS).

Today's GPUs are designed with a streaming, data-parallel architecture, which is completely different from the serial architecture of today's CPUs. Using the benefits of such a model, GPUs can reach a high level of floating point operations per second (FLOPS) (Figure 4.1). The stream computation model is the basis for programming the GPU. A stream can be defined as an ordered set of data of the same data type [Owe05]. A kernel, which can be a shader program, is able to make computations on the stream. The longer the stream, the more efficient the operation on those. This is because the kernel operates on the entire stream and not on individual elements of the stream. One kernel may project the input stream to another transformed output stream, which may be expanded or reduced in the numbers of stream elements.

The data-parallel paradigm is realized by ensuring that the computations of one stream element cannot affect or is not dependent on another stream element. In currently available GPUs, two models of data-parallelism exist. First, the fragment processors are

single instruction, multiple-data (SIMD) parallel processors, which means that the same instruction sequence is applied to all stream elements. Second, current vertex processors using Shader Model 3.0 are multiple instructions, multiple-data (MIMD) parallel processors, and are therefore able to execute different instructions on all stream elements. However, the current GPUs have more powerful fragment processors as the SIMD architecture is highly efficient and cost-effective [LKO05]. The following code fragment shows the difference between a serial programming paradigm and the streaming, data-parallel paradigm:

Listing 4.1: serial programming paradigm

```
for i:=0 to i<serialData.size() do
begin
        {mainLoop(serialData[i])}
end;
```

Listing 4.2: streaming programming paradigm

```
inputDataStream := getInputStream()
kernel := mainLoop()
outPutDataStream := apply(kernel, inputDataStream)
```

An application designed to work with the streaming programming model is constructed by a chain of kernels and thus represents a functional block diagram where the output of each block is connected to the input of another block. The graphics pipeline represents such an architecture (Figure 4.2) [Owe05] where the only programmable blocks are the vertex and fragment program stages.

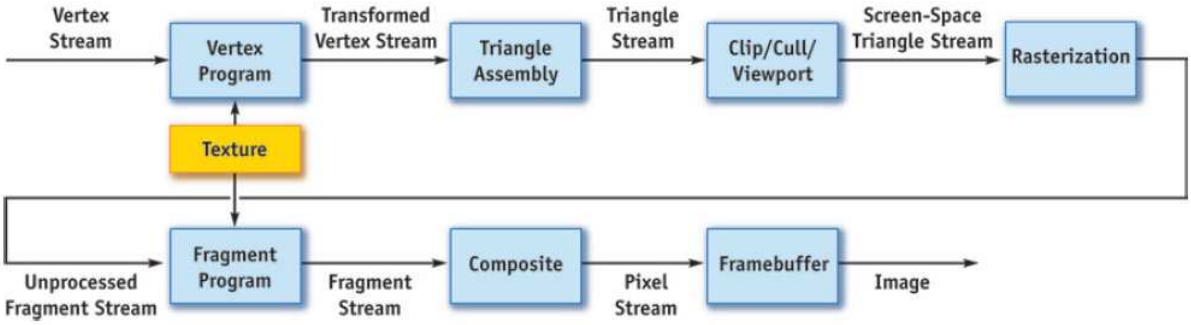


Figure 4.2.: Graphics pipeline as a stream model [Owe05].

Nvidia's Geforce6 series architecture is illustrated in Figure 4.3 as a block diagram. In the first stage, the commands, textures and vertices are sent from the CPU to the GPU through either shared buffers in system memory or local frame buffer memory (Figure 4.4. State changes and initializations, rendering commands, and references to textures and vertices, which have been written by the CPU are parsed and fetched by units and flow downstream to stages, where they are used.

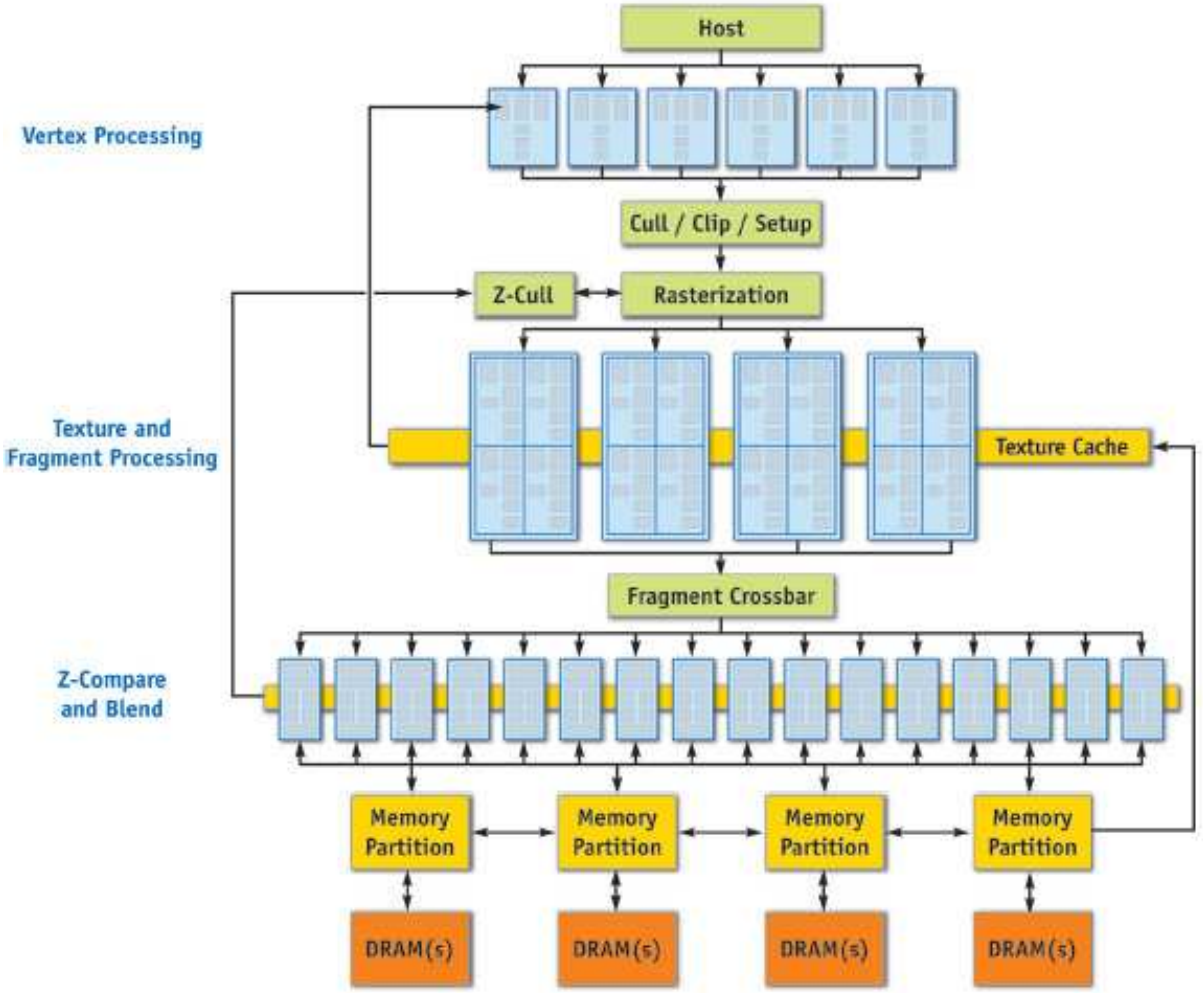


Figure 4.3.: Nvidia's Geforce6 series architecture [KF05].

The vertex processor applies a program or shader to every vertex fetched. It performs transformations, skinning, and any other per vertex operation declared in the shader. Additionally, the vertex processor of Geforce6 series are connected to the texture cache, which is shared with the fragment processor, as the vertex processor is able to fetch texture data. Vertices are then grouped to primitives, which then are culled, clipped and prepared for the rasterization.

The rasterization block is responsible for the calculation and decision which pixels are covered by each primitive. The raterizer uses the z-cull block to determine which pixels are occluded by other objects with a nearer depth value and discards those. Then, each fragment or "candidate pixel" [KF05] is send to the fragment processor where it passes user specified shaders and several tests, and if it gets through all, it will end up in a frame buffer

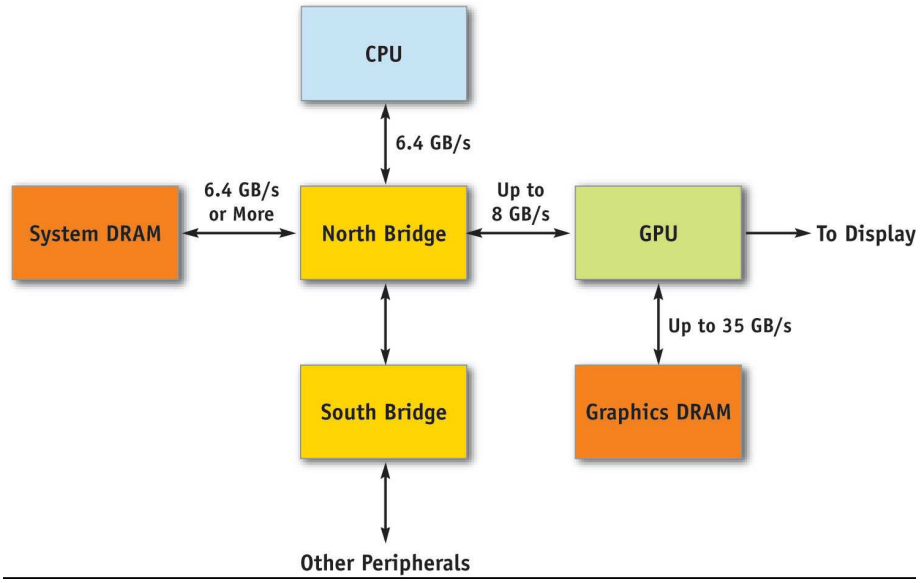


Figure 4.4.: Overall system architecture on a modern PC [KF05].

as a pixel carrying color and depth value. The last block then performs depth testing, stencil operations, alpha blending, and finally writes the pixel to the target.

### 4.1.2. Essentials About GLSL

The OpenGL Shading Language (GLSL), or sometimes called "OpenGL SL" has been approved in June 2003 as an ARB extension to OpenGL 1.5, and added to the OpenGL 2.0 core in September 2004. As shown in the last section, recent trends in GPU hardware architecture has been replacing the fixed function pipeline with programmable stages in areas that have grown in their complexity. In Figure 4.5 the processing pipeline of OpenGL 2.0 is illustrated. Note how it matches the stream model shown in the last section.

The OpenGL Shading Language has been designed to provide such programmability for two stages, namely the vertex and the fragment processor (Figure 4.6 and Figure 4.7). The vertex processor operates on incoming vertex data and is intended to perform operations such as [Ros04]:

- Vertex transformation
- Normal transformation and normalization
- Texture coordinate transformation and generation
- Lighting

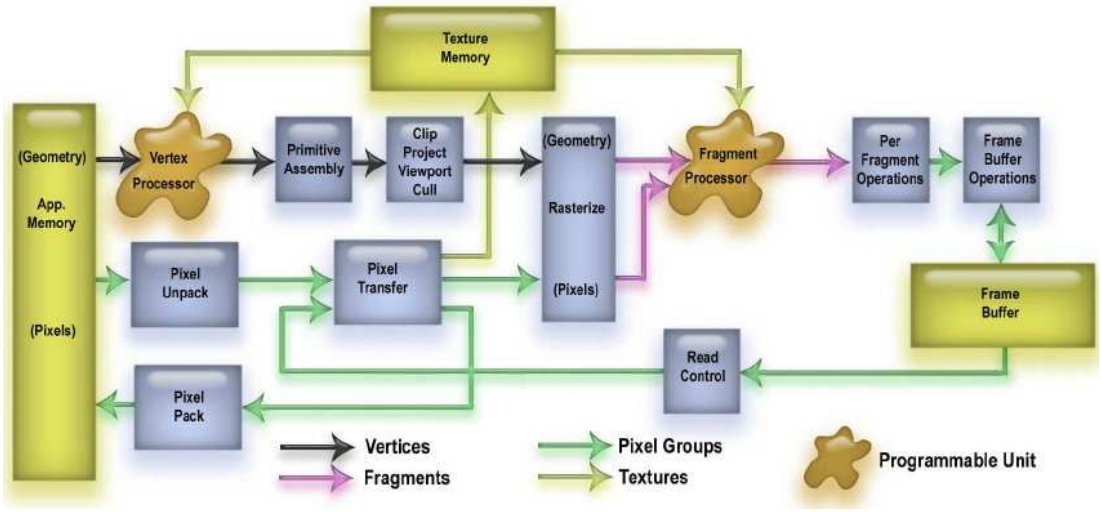


Figure 4.5.: OpenGL 2.0 processing Pipeline [Ros04].

- Color material application

The fragment processor operates on fragment data and is intended to perform operations such as [Ros04]:

- Operations on interpolated values
- Texture access
- Texture application
- Fog
- Color sum

Because of the general purpose programmability of the vertex and fragment processor a wide variety of other computations can be performed on them.

The code containing the GLSL code is called OpenGL shader, and as there exists two different types of processors, there also exists two different shader types: vertex shaders and fragment shaders. OpenGL provides mechanisms to compile and link the shaders to an executable that can run on the programmable units. The GLSL compiler is in the GL driver and part of the core OpenGL API, thus no external compiler tool is required. Because of this tight integration, hardware vendors can exploit their architectures for best possible performance (Figure 4.8). Similar to RenderMan [PIX05] and other shading languages, like CG [NVI05a] or HLSL [Dir05], GLSL has its roots in the C programming language, but enriched with some new types and 3D graphics operators, and reduced other language parts. The removed parts can be summarized into:

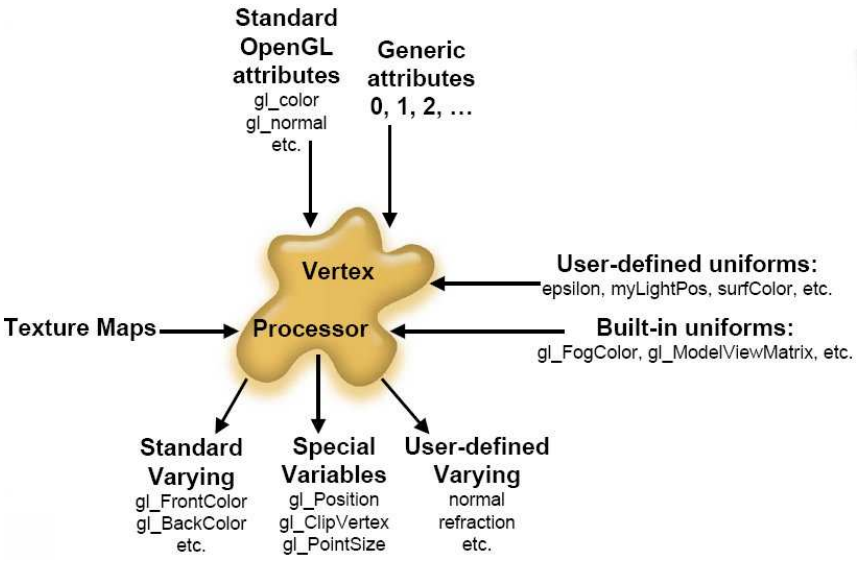


Figure 4.6.: Vertex processor overview [Ros04].

- Pointers and operators related to them
- String data or character and their methods
- File based directives
- Unions and enums
- Bit operators

Some additions made to the C language are:

- Types for two, three and four dimensional vectors and matrices with different constructors for different elements, like *ivec4* for a four component vector of integers. The default constructor, e.g. *vec4* is used for floats.
- Often used linear algebra methods defined for scalar, vector and matrix types, e.g. dot product, normalization, matrix product.
- A set of basic types, called *sampler* for accessing texture memory.
- Call by value-return rather than call by value

Some additions has been borrowed from the C++ language, such as:

- Function overloading
- The concept of constructors
- Variables can be declared when they are needed

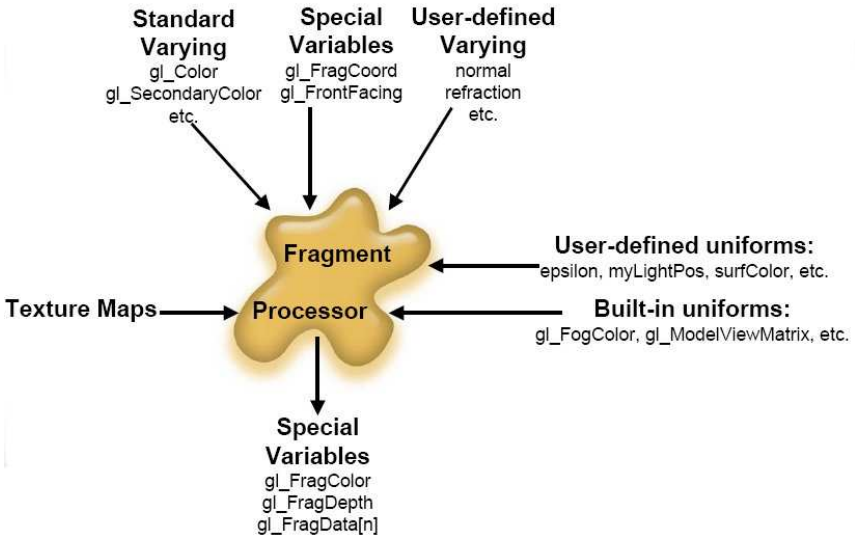


Figure 4.7.: Fragment processor overview [Ros04].

- Functions must be declared before being used

Additionally, three new type qualifiers were introduced to specify the input and output of shaders:

- **Uniform qualifier**, used for infrequently changing information in the vertex and fragment shaders. They can change once per primitive. There are built in uniforms from the OpenGL state machine like: gl_ModelViewMatrix, gl_Point, gl_Fog, etc.
- **Attribute qualifier**, used for frequently changing information, i.e. once per vertex and thus only available in the vertex shader. There are built in attributes like: gl_Vertex, gl_Normal, gl_Color, etc. for reading the traditional OpenGL state.
- **Varying qualifier**, used for interpolated information passed from the vertex shader to the fragment shader. Varying qualifiers are the interface between the vertex and fragment shader. Built in varying variables are for example: gl_TexCoord[], gl_FogFragCoord, etc.

Listing 4.3: A simple vertex shader in GLSL

```
// global declaration of uniform, attribute
// and varying variables
varying vec2 TexCoord;

// every GLSL shader must have a main function that
// is called in the programmable processor

// Vertex Shader main
void main()
{
        // standard transformation of the vertex to the eye space could also be
```

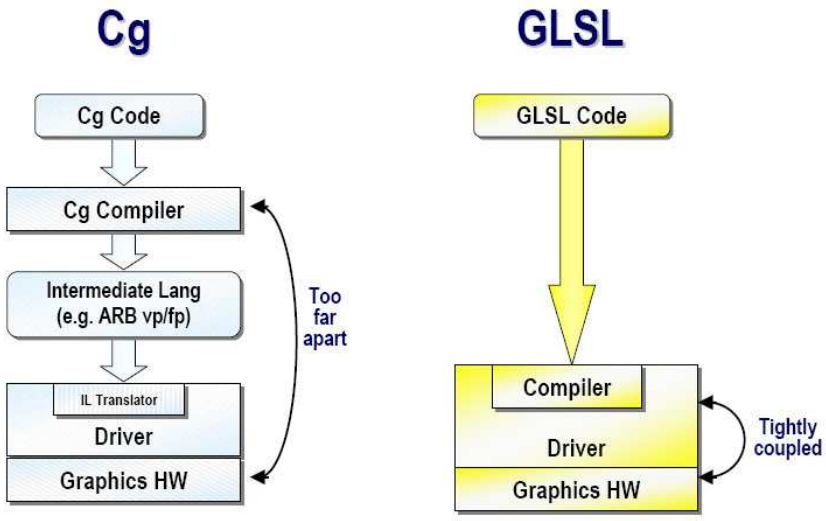


Figure 4.8.: The advantage of the tight integration of the compiler in the driver [Ros04].

```
        // gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
        // but ftransform() keeps inaccuracies with the fixed function pipeline
        gl_Position = ftransform();

        // send the texture coordinates to the fragment shader through the global
        // varying variable declared.
        TexCoord = gl_MultiTexCoord0.xy;
}
```

The example Vertex Shader in Listing 4.3 transforms first the vertices from model space to eye space, using the built-in function *ftransform()*. Then it gets the texture coordinates from the OpenGL state machine as a built-in per vertex attribute and stores them in the varying variable *TexCoord*.

In the Fragment Shader Listing 4.4 the texture coordinates are used to get the texel from a texture (image data), and to multiply the color of that texel with an ambient color component defined by an uniform variable.

Listing 4.4: A simple fragment shader in GLSL

```
// global declaration of variables the varying variable must have the same naming
// convention this holds the interpolated (through rasterizer) texture coordinates
varying vec2 TexCoord;
// uniform variable holding an ambient color
uniform vec4 ambientColor;
// uniform variable holding sampler/texture data
uniform sampler2D myTexture;
// every GLSL shader has to have a main function that
// is called in the programmable processor

// Fragment Shader main
void main()
{
```

```
        // get the texel at the texture coordinate
        vec4 myTexel    = texture2D(myTexture, TexCoord);

        // output this fragment as the texel weighted with the ambient color component
        gl_FragColor = myTexel * ambientColor;
}
```

Please refer to [Ros04] for further informations and tutorials.

### 4.1.3. OpenSG Basics

As the VAR-Trainer's Visualization Module builds on top of OpenSG, we have to make sure that our shaders run in an OpenSG environment. This section is about the basics behind the OpenSG architecture and how a simple scene with a simple GLSL Shader is implemented in OpenSG.

OpenSG is a real time rendering library built on top of OpenGL, and it uses the scene graph metaphor to handle a graph based topology for connecting the nodes/objects together and thus building a knowledge network. OpenGL is a state machine and does not know anything about objects represented in the scene. All OpenGL knows are e.g. vertex positions to build a drawable polygons, as it is just a thin layer over the graphics hardware. A scene graph has knowledge about the topology of a scene, due to it's construction as a acyclic[1] and directed[2] graph. It consists of Nodes, representing information about topology, or any other information a node is designed for, in a graph. This way, the rendering of a scene can be optimized. Imagine a node representing a normalized number on how far the closest, underlying geometry is away from the viewer, where 0 is close and 1 is very far away. Now the rendering command that parses the graph and sends the geometry and other necessary data to the OpenGL state machine can look-up for this node in the tree and, e.g., decide to clip objects that are $> 0.7$. Based on this a decision can be made to use a less complex version of the underlying geometry (basic idea of Level of Detail (LOD)).

A scene graph is an abstraction layer, and in case of OpenSG it is an abstraction layer on top of OpenGL. One big advantage of the scene graph metaphor is the share of data making it possible to build a car with e.g. three meshes: a body, an engine and one wheel transformed to the correct places. In OpenSG data is shared through the concept of *Nodes* and *NodeCores* that is illustrated in Figure 4.9. A Node consists of the following informations: a list of children Nodes, a pointer to the parent Node, a bounding volume, a traversal mask and usually a pointer to a NodeCore. Thus, a Node does not contain

[1] Loops are not allowed in the graph, as they would cause the render command to loop infinitely while it parses the graph.

[2] There is a defined direction of the connections between the nodes, which means that nodes can be grouped together and form a hierarchy of parent and child nodes. This concept allows to manipulate a parent node, which in turn manipulates all child nodes.

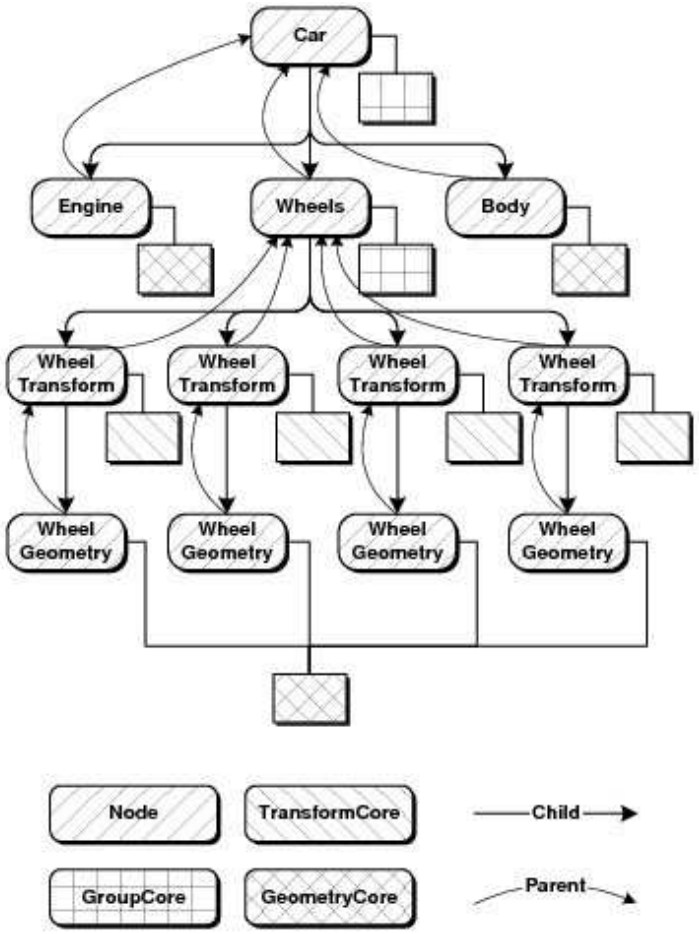


Figure 4.9.: Design of the data sharing concept in OpenSG [Ope05c]. Nodes build the topology, while NodeCores hold relevant data.

any content or data for the scene, but Nodes build the topology of the graph and gives information, for instance, if a node and its children should be traversed or if they should be left out. However, two general types of NodeCores exists, which either hold structural information or drawable data. Groups are the type of NodeCores holding structural information:

- **Group** traverses commands to its children Nodes.
- **Switch** selects one of its children the and passes the commands to it.
- **Transform and ComponentTransformation** define a coordinate system for its children relative to its ancestor.
- **DistanceLOD** switches between geometry versions based on distance to the viewer.
- **DirectionalLight, PointLight and SpotLight** influence all children with light.

Drawables are the type of NodeCores representing actual information about **geometry** and **materials**.

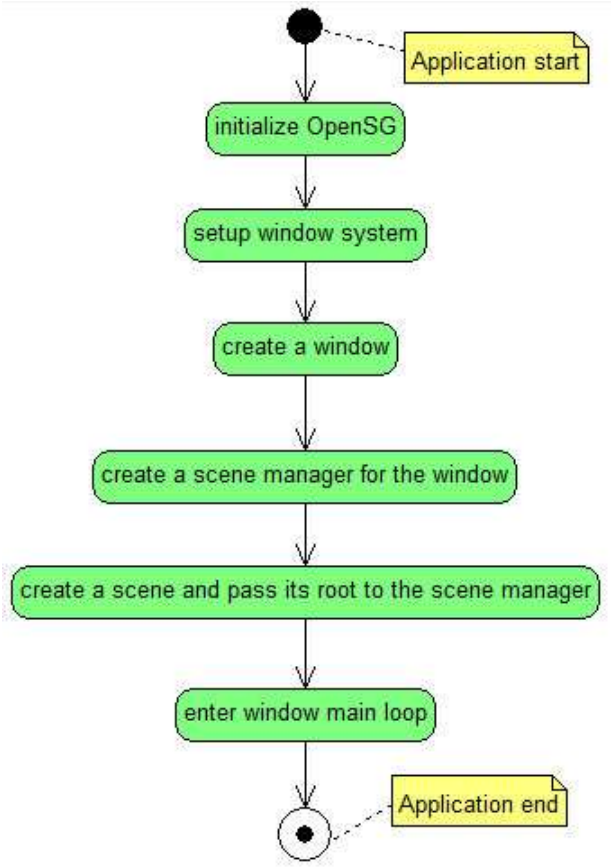


Figure 4.10.: Basic OpenSG application as a state diagram.

In Figure 4.10, a simple OpenSG setup is shown. We refer the interested reader to Listing A.1 in Annex A for a complete source code on how to set-up a simple scene in OpenSG. One important thing to note about OpenSG is its thread-safe usage, which is reached through locking and unlocking the current state that is manipulated using *beginEditCP()* and *endEditCP*. Please see [Ope05c] for more methods and Tutorials on OpenSG and its architecture.

In Figure 4.11, it is shown how the shader introduced in Listing 4.3 and Listing 4.4 are applied to a torus. We assume to have the GLSL shaders saved in files, namely: *simple-vertex-shader.glsl* and *simple-fragment-shader.glsl*. See Listing A.2 in Annex A for the full OpenSG source code.

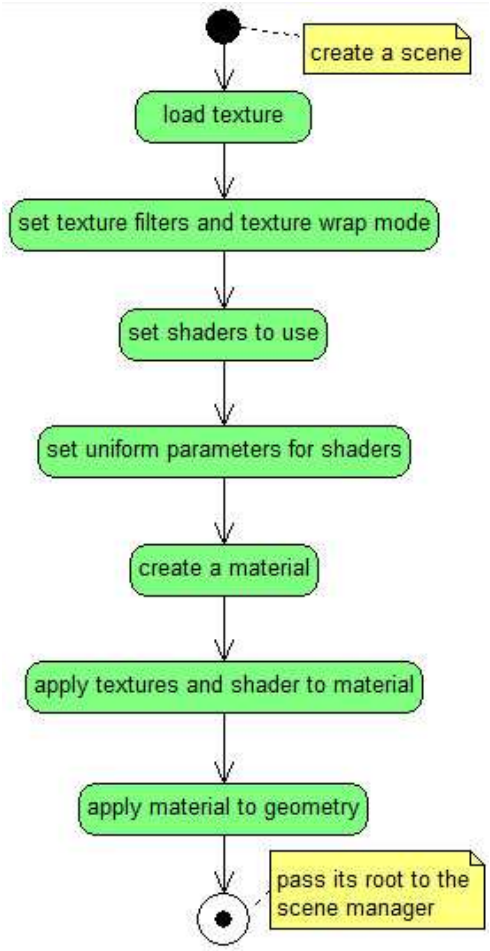


Figure 4.11.: Basic OpenSG scene node including a shader node as a state diagram.

## 4.2. Using OpenSG to Render the Shaders

Managing a scene with hundreds of objects that have to be shaded require a great deal of code. To facilitate the usability of the shaders we realized an own class for each of them. This classes allow to create shader instances that can be attached to the graph. They provide an interface to manipulate certain parameters to control the behavior of the effects. Additionally, all unnecessary computations that can be avoided to be done on the GPU, are done on the CPU and shared between shaders, e.g. sun to camera vector, exposure of the camera, etc. The simplified UML Class Diagram of the shader class is illustrated in Figure 4.12. The complete UML Class Diagrams can be found in Figure B.1, Figure B.2 and Figure B.3. The **SimpleShader** class is the base class for all the shader classes. The Idea behind this class is to have dynamic polymorphism through the usage of virtual functions. This class provides data used in every shader:

- the mesh filename (_shMesh) and the vertex and fragment shader filenames (_shVP, _shFP).
- reference to the shader node (_shChunk), the root graph node organized by this class (_shNode), and a pointer to the material node (_shMaterial).

Notice also the other parameters, which we use for switching between two versions of the scene: one that uses shaders and one that uses texture mapping. We explain the reason for this in the next chapter.

The **Rain** class provides extra data, as it uses extra layers of rain to simulate the effect of rain in a Virtual Reality (VR) set-up. Thus, instancing a Rain object, a layer with some default parameters for the rain effect is generated. Adding more layers to the rain system for a VR set-up is done by calling the *addlayer(*scale*)* method, while the scale parameter changes the scale to the previous layer. The tilting of the double cone can be controlled with the *setRainDir* method to simulate velocity or stormy rain. The double cone itself is loaded from a mesh file and is adapted from the polygon model shown in [Wan05]. This class has an animation of the rain that gets updated by passing a time parameter to the *updatePrecipitation* method. This is done on every new rendered frame. The second part of this class that has to be updated is the matrix of the first double cone. Every time the camera updates the position, the method *updateMatrix* should be called. This ensures the camera to be always surrounded by the double cones as described in the method for the creation of rain simulation presented in subsection 3.2.3 and [Wan05]. The reason why only the first double cone has to be updated is that all other double cones, which are added through the *addlayer* method, use the coordinate system given by the first double cone, using the Transformation NodeCores described in the previous section.

The **Atmosphere** class uses two Design Patterns to reduce complexity. First, it uses the Singleton Pattern, as some data used by the Atmosphere is also used in other classes. Then, it uses the Observer Pattern to provide a communication channel between classes that want to reuse the data and the Atmosphere class. Thus, the Atmosphere class is derived from the Subject class. The Atmosphere class is designed to render the sky and provide data to render atmospheric effects. The reason why we do not render the atmospheric effects in this class, and save all the data to send it to all observers is that the atmosphere just knows about the sky and has no knowledge about the objects on the ground. Parameters that control the atmospheric effect (see subsection 3.2.1) but are not directly sent to the shaders are:

- **Kr** is the Rayleigh scattering constant.
- **Km** is the Mie scattering constant.
- **ESun** controls the energy of the sun.
- **g** is the Henyey-Greenstein phase function parameter.

- **dawn** is a constant controlling the behavior of the sun, when it is near horizon. Some may want to have a dawn effect, where others might want to have still a hazy, white Sun. This parameter ranges from $-1$ to 1, where negative values create a dawn effect and positive values a hazy, white-yellowish sun at the horizon.

- **density** is a factor for the exponential fog attenuation.

As most of the computations depends on the viewers position, the camera position is updated every time the viewer changes the position through the *setCamPos* method.

The **RainDrops** class and the **Clouds** class have both the *setTime* method to update the animation of drops on windows and clouds forming, appearing, and disappearing. More parameters are related directly to the shader and thus explained in the next section.

The class **SimpleNearGround** handles the atmospheric effects related to objects near the ground. It implements the virtual *update* method, used to get the updated parameters from the subject it observes (Observer Pattern [Wik05]). This class is useful when objects without textures have to be added to the scene e.g. a crane with red and shiny, metallic material. Adding an object to the scene using this interface ensures correct atmospheric perspective, fog and lighting. Most of the parameters are related to the *Atmosphere* class and thus discussed in the next section.

**SimpleTexNearGround** handles more complex shading for near ground objects. It provides applying one or two textures and a bump map to the shader through the constructor interface. It is generic in a way that it decides which shader to use when certain parameters are provided.

The **Soil** class has the same decision theory like *SimpleTexNearGround*, but additionally it has to decide between two more versions of the shader to apply either a shader using Shader Model 3.0 (SM 3.0) or Shader Model 2.0 (SM 2.0). For now this decision is based on a single boolean variable, which is by default false (SM2.0), but can be set to true as a constructor parameter.

## 4.3. Shader Implementations

As shown in the last section, all shaders are optimized to make just the necessary parts of the computations in the GLSL Shader. In this section we go through every shader, pick the important parts of the shader and explain how those parts influence the rendering of the effect. Additionally, we explain all input parameters.

### 4.3.1. Atmosphere

This shader needs a sphere or a sky dome as input for the vertex shader. Depending on incoming sun angle and view position, a color is calculated using simplified versions of Equation 3.16 described in subsection 3.2.1. As the interpolation of the sky color is done in the fragment shader, the sky dome does not have to have a complex geometry.

#### Vertex Shader

The vertex shader has different input parameters that are mostly calculated on the CPU and passed to the shader. This is because the shader would calculate those values per vertex and thus more often than needed. The input parameters are:

- **attenuationFactor:** In [O'N05] this value is calculated in the vertex shader, as it depends on the altitude of the viewer. We reduced this to a uniform parameter, which generates the value now depending on the angle to the sun and the *dawn* parameter. Both parameters do not change frequently and indeed not per vertex, which makes is feasible to compute the *attenuationFactor* on the CPU. The value is calculated by:

  $$attenuateFactor = (sunAngle + dawn) \cdot (invWaveLength \cdot (fKr4PI + fKm4PI))$$

  Where

  $$invWaveLength = \frac{1}{wavelength}$$
  $$fKr4PI = Kr \cdot 4 \cdot \pi$$
  $$Km4PI = Km \cdot 4 \cdot \pi$$

- **colorFactor** is dependent on the energy of the sun:

  $$colorFactor = invWaveLength * fKrESun$$

  Where

  $$fKrESun = Kr \cdot ESun$$

- **KmESun** $= Km \cdot Esun$.
- **atmoConst** creates a thin layer of haze at the horizon, where the color of the haze depends on the *dawn* parameter. This parameter is in the range of $[0, 10]$, where zero means no haze and 10 fills almost half of the sky.
- **vViewPosition** is the camera position in model space.
- **upVec** is the world up vector. This gives freedom about system design decisions as some applications use $y$ and others $z$ as their up vector.

- **v3Direction** is a varying variable passing the value of the viewer to the current vertex as an interpolated value to the fragment shader.

In the first line of Listing 4.5 the attenuationFactor gets exponatiated (optical depth) and is dependent on the angle between the *upVec* and the normal of the current vertex. In the second and third line the attenuated color is weighted with the energy of the sun (brighter regions of the sky where the sun is).

Listing 4.5: Optical depth and weighted sky color.

```
1 vec3 v3Attenuate = exp( attenuationFactor*exp2(angle*atmoConst));
2 gl_FrontSecondaryColor.rgb = v3Attenuate * fKmESun;
3 gl_FrontColor.rgb = v3Attenuate * (colorFactor);
```

### Fragment Shader

The fragment shader uses the adapted *Henyey-Greenstein Phase Function* Equation 3.11 to simulate *Mie Scattering* (subsection 3.2.1). The Mie term is also responsible for rendering the sun. The input parameters are:

- **sunDir** is the direction of the sun on a normalized sphere.
- **exposure** controls the overall light in the scene. This is very important for a rainy scene, where the scene should get dark.
- **miePhaseConst** is the constant part of the adapted Henyey-Greenstein Phase function.

In Listing 4.6 the resulting in-scattering term is shown.

Listing 4.6: In-scattering result for the fragment shader (subsection 3.2.1).

```
1 vec3 dir = normalize(v3Direction);
2 float fCos = dot(sunDir, dir)/length(dir);
3 float fMiePhase = miePhaseConst.x * (1.0 + fCos*fCos) /
4                   pow(miePhaseConst.y - miePhaseConst.z*fCos, 1.5);
5 vec4 comp = gl_Color + fMiePhase * gl_SecondaryColor;
```

## 4.3.2. Clouds

The cloud shader creates texture coordinates, animates them and send them through the rasterizer to the fragment shader. The fragment shader then adds details to a low frequency noise texture, and shades them with another texture. Finally, the clouds get faded out at the horizon.

### Vertex Shader

In the vertex shader, first the fade value of the sky dome is generated. Then, texture coordinates for a sky dome are calculated and animated using a wind vector and a time step. The input parameters in the vertex shader are:

- **timeX** is the time step.
- **windVec** is a two dimensional vector giving the direction and strength of wind.
- **timeConst** is a four dimensional vector parameter, which drives the animation for every cloud layer differently.
- **tCloudHeight** is a factor defining the cloud height that is faked through the texture coordinate computation.
- **upVec** defines the world up vector.

In Listing 4.7 the main part of the vertex shader for the clouds is listed.

Listing 4.7: Calculation of the animation and texture coordinates for the clouds vertex shader.

```
1  vec4 t = timeX * timeConst;
2  // compute TextureCoordinates
3  // use preprocessed mesh if possible to avoid these instructions
4  vec3 norm = normalize(gl_Vertex.xyz);
5  vec3 offset = upVec*norm+tCloudHeight;
6  vec2 vTexCoord2;
7  // cannot avoid this as the project staff want to change the upVector as parameter!
8  if(offset.y==0.0)
9       vTexCoord2 = (norm.xz/offset.y)*tCloudHeight;
10 else
11      vTexCoord2 = (norm.xy/offset.z)*tCloudHeight;
12
13 vTexCoord2    = atan(vTexCoord2);
14 // pack them into variables
15 vec4 coord = vec4(vTexCoord2,vTexCoord2);
16 vec4 wind = vec4(windVec, windVec);
17 vTexCoord0 =  coord + wind * t.xyxy*2.5;
18 vTexCoord1 = coord + wind * t.zwzw;
```

### Fragment Shader

The fragment shader generates the clouds by adding different textures with different frequencies together. Based on the covering parameter it calculates how much clouds cover the sky, and finally it shades the clouds with another extra layer of clouds. The input parameters are:

- **octavesScales** is a four component vector, which scales every cloud layer differently.

- **octavesWeight** is also a four component vector, weighting the importance/intensity of the layers.
- **sunColor** is the color of the sun. This way a direct illumination of the clouds, can be simulated.
- **lightMultiplier** changes the intensity of the *sunColor* and can produce dark clouds if set to zero.
- **cloudCover** is a value defining how much sky can be seen. Zero means no sky and above parameters mean less clouds.
- **clouds_sharpness** changes the fluffyness of the clouds.
- **textureMult** is a multiplier for the texture coordinates that allows to change the frequency of visible clouds.
- **octaves** represents a texture holding four cloud textures in different channels (RGBA). The first layer (Red) includes the low frequency noise layer, in the second layer (Green) is the cloud texture with higher octave and so on.
- **extraClouds** is again a texture, which contains four channels of different cloud layers. This is used to shade the clouds.

In line 17 of Listing 4.8 the average of the additional layer of clouds and the computed clouds (*tex*) Equation 3.20 is saved in *shaded*. In line 20 the *lightMultiplier* is applied to it generating a different version of the clouds.

Listing 4.8: Shading the clouds with average function.

```
1  // Sample noise map three times with different texture coordinates
2  vec4 offset = octavesScales * textureMult;
3  float tex = texture2D(octaves, vTexCoord0.xy * offset.x).r * octavesWeights.x;
4  tex += texture2D(octaves, vTexCoord0.zw * offset.y).g * octavesWeights.y;
5  tex += texture2D(octaves, vTexCoord1.xy * offset.z).b * octavesWeights.z;
6  tex += texture2D(octaves, vTexCoord1.zw * offset.w).a * octavesWeights.w;
7  //extra clouds could also use other channels for diff look
8  float clouds_0 = texture2D(extraClouds,vTexCoord0.xy).r;
9  clouds_0 += texture2D(extraClouds,vTexCoord1.xy).r;
10 clouds_0 *= 0.5;
11
12 // cloud covering
13 tex = max(tex - cloudCover, 0.0);
14 // sharpen the clouds
15 tex = 1.0-pow(clouds_sharpness, tex * 255.0);
16 // shade the clouds
17 float shaded = (clouds_0+tex)*0.5;
18 float compose = shaded;
19 // lighten or darken the clouds if needed
20 shaded *= lightMultiplier;
21 // fade the clouds near horizon
22 tex *= fade;
23 float finalComposition = clamp(((compose+shaded)*0.5)-tex, 0.0, 1.0);
```

### 4.3.3. Rain

The rain shader first generates the texture coordinates transformation matrix Equation 3.25 and multiplicates this matrix with the texture coordinates (subsection 3.2.3). Then it maps the rain texture with the transformed texture coordinates on the double cone and applies an intensity value to it.

#### Vetrex Shader

The vertex shader generates, based on the input parameters, a 4x4 animated transformation matrix and multiplicates it with the texture coordinate of the current mesh. The rain shader can be extended to simulate more scene depth by using more than one rain texture movement. In the vertex shader more than one animation has to be calculated, which firstly should depend on how many internal layer you want to use. Secondly, it should depend on the depth of the layer you want to simulate. Based on the desired distance, a slower animation should be calculated as the more far an object is, the slower it seems to move, while near objects pass the view frustum very fast (camera/viewer is fixed). Then the fragment shader adds all layers together.

The input parameters for the vertex shader are:

- **E** is the scale factor manipulating the x-scale. It is procedurally generated and dependent on *scaleFactor*.
- **scaleFactor** is the main control factor for scaling. This can be used to simulate longer streaks of rain drops and simulate motion blur.
- **xFactor** is a factor responsible for an offset in the x direction. This is important if more than one layer is added to the scene to prevent layers to overlay each other when they are animated.
- **dPrecip** is the precipitation factor that controls the amount of rain precipitation.

In Listing 4.9 the built-in attribute for texture coordinates is transformed with the desired transformation matrix (Equation 3.25).

Listing 4.9: Transforming the texture coordinates with a pre-calculated matrix.

```
1 // calculate the texture transformation matrix
2 mat4 M = mat4(E,       0.0,          0.0, xFactor,
3               0.0,    scaleFactor,  0.0, dPrecip,
4               0.0,    0.0,          1.0, 0.0,
5               0.0,    0.0,          0.0, 1.0);
6 // set transformed texture coords
7 coords = gl_MultiTexCoord0 * M;
```

#### Fragment Shader

The fragment shader uses the incoming interpolated and animated texture coordinates for a texture look-up. Then it maps the texture to the mesh, applying an intensity value to it. The input parameters are:

- **rainColor** is a four component vector allowing to set a color and intensity value for the rain.
- **multiplier** is a factor controlling how often the texture repeats over the mesh.
- **rainSampler** is a texture holding texture information in its RGB channels (Figure 4.13.

In Listing 4.10 the texture coordinates are multiplied by a factor to allow factoring of the repitition of the texture.

Listing 4.10: Get on layer of the shifted texture.

```
1 // set the fragment color for each layer
2 vec4 layer = texture2D( rainSampler, coords.st * multiplier);
3 // output colored rain
4 gl_FragColor = layer * rainColor;
```

### 4.3.4. RainDrops

This shader simulates rain drops splashing on transparent objects like windows. It calculates a mask texture based on a random noise texture and multiplicates the mask with a rain drops texture. The result is then animated in the y-direction to simulate gravity influence.

#### Vertex Shader

The only thing the vertex shader does is the animation of the texture coordinate in the y-direction to simulate gravity:

$$Texcoord.y = gl\_MultiTexCoord0.y - timeX;$$

And thus the only input parameter is the $timeX$ value, which defines a time step.

#### Fragment Shader

The fragment shader calculates a mask out of a 3D noise texture and masks in every frame certain regions of the rain drops texture as visible and invisible. The input parameters are:

- **timeX** is the time step and used for the 3D noise texture to get different slides on different frames.
- **multiplier and multiplier2** are responsible for how often the drops texture or the mask texture are repeated. This controls the frequency of the drops.
- **rainSampler** is the drops texture.
- **Noise** is a 3D random noise texture and can have a small resolution (64x64x64) (Figure 4.14).

In Listing 4.11 the random texture is bounded by a smooth function within the values $[0.45, 0.8]$, which creates larger spaces in the noise texture with black and white. The variable *drops* holds the color value of the current drops texture texel, and *mask* the current random noise texture texel.

Listing 4.11: Masking the drops texture with a random texture.

```
1 vec4 drops = texture2D(rainSampler, Texcoord * multiplier);
2 vec4 mask = texture3D(Noise, vec3(Texcoord *multiplier2 , timeX));
3 // mask drops with noise texture
4 finalComp = drops * smoothstep(0.45, 0.8, mask.x);
```

### 4.3.5. Soil and other Near Ground Objects

The soil shader is like all shaders for near ground objects highly related to the atmospheric effects. Thus it generates the atmospheric values like already described for the atmosphere shader, but this time with the use of the current vertex data. Then, in the fragment shader, the atmospheric color is added to the base color, which generates a realistic look of atmospheric perspective. Additionally, the soil shader does a sphere tracing in tangent space to find the correct offset value for the parallax effect [Don05] Equation 3.7.

#### Vertex Shader

The vertex shader has to transform the current eye and light vector into tangent space. The rest is related to the atmosphere shader. The only important input parameter is:

- **displacementDepth,** which is a factor in the range from $[0, 1]$, where 0 means no displacement at all and 1 means full displacement.

In Listing 4.12 the transformation matrix is first generated. This matrix is made up of the normal, binormal/bitangent and the tangent. These three vectors build an orthonormal basis and thus can transform from one space into another. Transformation then is applied through multiplication. Finnaly, the *displacementDepth* is applied. See Listing C.1 for the full shader code.

Listing 4.12: Transformation into tangent space.

```
1        //transformation by orthonormal basis
2    // OSG packs tangent and binormal into texcoord after
3    // calling calcVertexTangents()
4    mat3 tangentMat = mat3(gl_MultiTexCoord1.xyz,
5                           gl_MultiTexCoord2.xyz,
6                           gl_Normal);
7
8    // Transform the eye vector into tangent space.
9    tanEyeVec = eyeVec * tangentMat;
10   // And apply displacement depth
11   tanEyeVec.z *= (-1.0/displacementDepth);
```

### Fragment Shader

The fragment shader traces through the prepared distance map texture to find an intersection with the profile. This is then the correct offset to be used for the parallax effect. Additionally it uses bump mapping to generate self shadowing. In order to generate the correct horizon color, it uses a height map to define how deep the current soil is that has to be rendered. Then, the value is used to look-up in a one dimensional gradient texture that defines the color at the current fragment. As the bump map and the distance map texture are generated from a base texture, this texture is again used as a base color to have a slight diference in the coloring (Figure 4.15). The input parameters are:

- **normalizationFactor** is used to normalize the offset. It is the depth of the 3dTexture divided by the width of the 3dTexture e.g. $\frac{8}{256}$ with 8 slices and $256x256$ texture size.
- **ambient** is a parameter to generate an ambient factor, if needed.
- **textureMult** defines how often the soil texture is repeated over the surface.
- **soilSampler** is a base texture from which all other textures are generated. It is also used as a base color.
- **normalSampler** is the normal map generated from the base texture using e.g. Adobe Photoshop and the Nvidia Tools.
- **gradientSampler** is a one dimensional texture, holding horizon color information.

- **digHeightMapSampler** is the height map, which should be updated very time the surface structure has changed. It is not necessary to use this when the height information is packed into texture coordinates. In fact, using the texture coordinate method would give the better performance as the information can be set when the surface vertex get modified. This is application specific.
- **disMapSampler** is a 3D distance map, holding information about shortest distances to the profile given by the base texture. First we create a displacement map with the Displacement Map Creator by Ryan Clark [Cla05], which we then use in our *distanceMap* generator that is based on the distance map generator from [Don05]. The *distanceMap* generator creates single slices of the 3D volume texture that has to be connected together to a 3D volume texture with the Nvidia texture Atlas tools or other tools capable with this kind of possibility.

In Listing 4.13 the tracing through the 3D distance map texture is shown. The *offset* variable is the normalized eye vector in tangent space, which the fragment shader gets from the vertex shader through a *varying* variable. Finally, the *distance2Trace* is the radius of the sphere used to optimize the tracing algorithm (Equation 3.4). See Listing C.2 for the full shader code.

Listing 4.13: Sphere tracing through the distance texture (Equation 3.7).

```
1 for (int i = 0; i < NUM_ITERATIONS; i++)
2 {
3       distance2Trace = texture3D(disMapSampler, texCoord).r;
4       texCoord += offset * distance2Trace;
5 }
```

## 4.4. Summary

In the first section we shortly reviewed the history of GPUs and focused later on the NV40 GPU series from Nvidia, as it is the hardware used in the VAR-Trainer project. Further, we discussed why the utilization of shader technology is feasible for the purpose of this work when using the computational power of the GPU.

Furthermore, we showed how the OpenGL processing pipeline matches the stream programming model and which parts of the pipeline are the programmable parts of it. The vertex shader block is the first stage in the pipeline that is programmable, and is responsible for vertex, normal and texture transformation, lighting, color material application, and other per vertex related manipulation operations. The NV40 has a MIMD architecture in its vertex block. The fragment shader is the second block in the pipeline that is programmable, and is responsible for operations on interpolated values, texture access and application, fog, color sum and other per fragment related manipulation operations. The NV40 has a highly efficient SIMD architecture in its fragment block.

We also introduced into the scene graph architecture of OpenSG, which is the target implementation API for visualization of all shaders. We showed how OpenSG builds the graph topology with *Nodes* and defines *NodeCores* as containers for the content.

In the second section, we discussed the software engineering part of the classes we realized as an interface to OpenSG. These classes can also be used in the VAR-Trainer project as an interface between its Visualization Module and OpenSG. The classes are mostly designed to reduce complexity of data handling and code writing. Parameters that are not directly related to the shader, but are used to calculate a parameter that will be sent to the shader, are also described.

Finally, the important parts of the shaders and all parameters related to the shaders were discussed in the last part of this chapter. Namely this were the vertex and fragment shaders for atmospheric effects, clouds, rain, raindrops and soil.

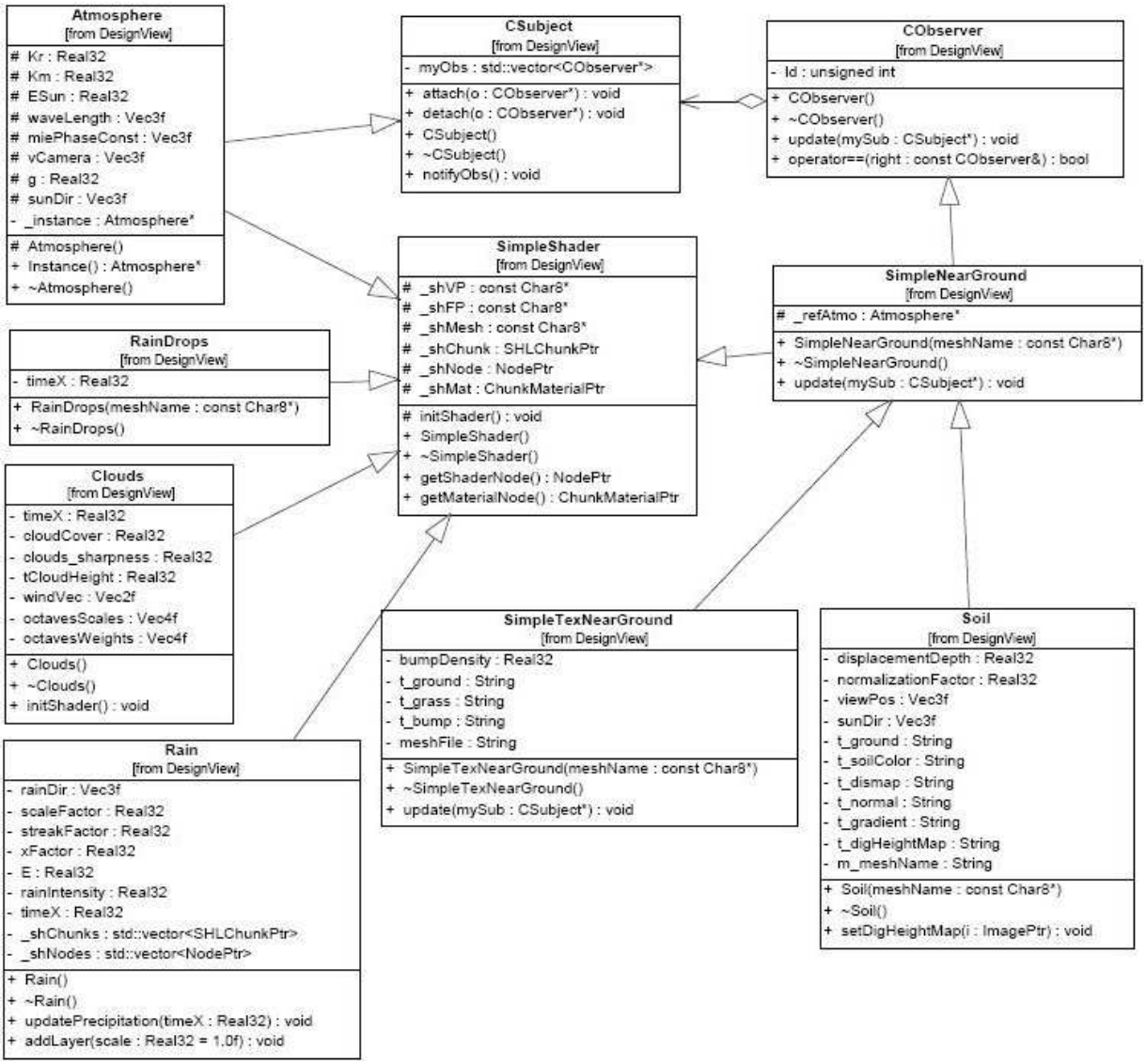


Figure 4.12.: UML Class Diagram of the shader classes.

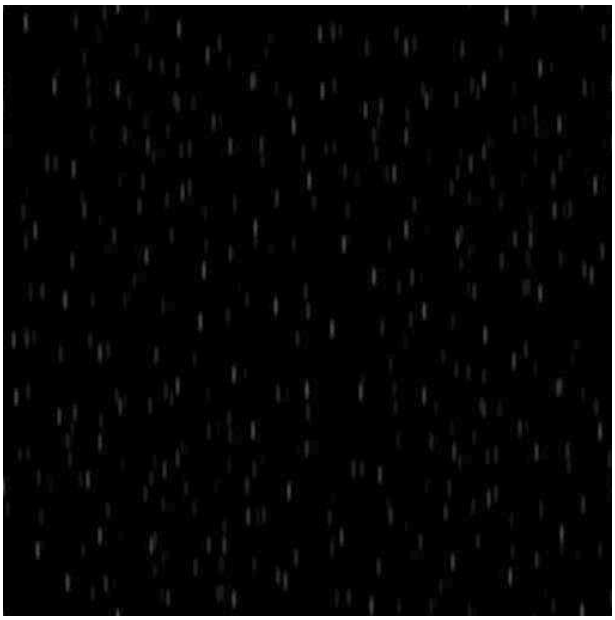

Figure 4.13.: Rain texture: 1-color (one minus color) is the alpha value.

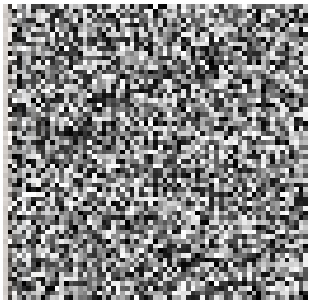

Figure 4.14.: Random noise texture.

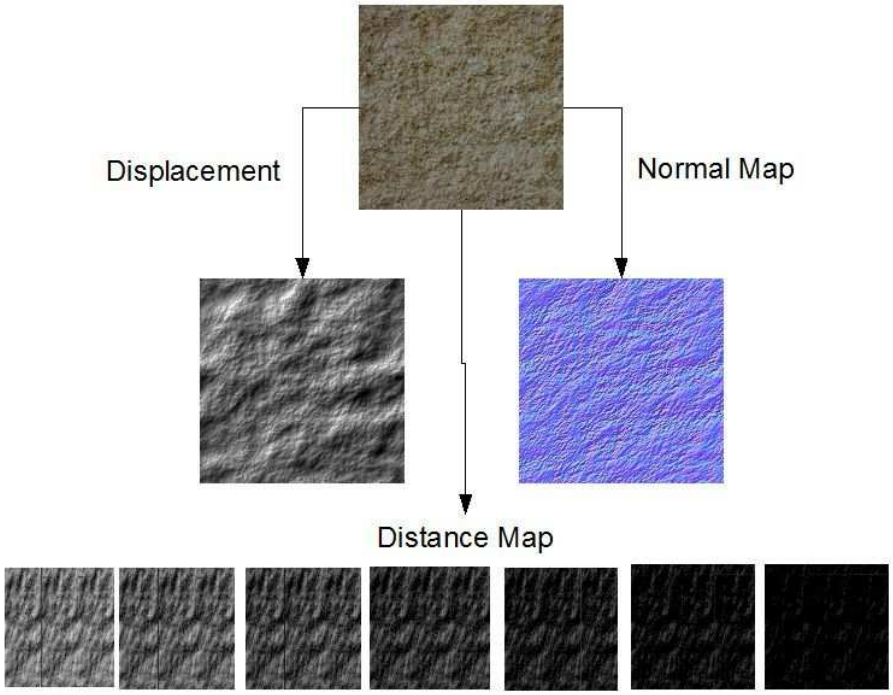


Figure 4.15.: Soil maps generation.

# Chapter 5.

# Experimental Results

In the previous chapters we described a combination of shaders to use for a construction machinery simulator to achieve better visualization results when rendering weather conditions and soil. The shaders, except the soil shader, are adapted from existing methods but improved for better performance. The soil shader builds on top of a technique [Don05] to reach a high frequency of granularity and applies a color gradient to simulate different horizons. In this chapter visual quality is compared to real photos to have a measurement of the photo realism. Shaders related to weather conditions are highly optimized as many simple assumptions simplifying could be made, e.g. the atmosphere shader do not have to be dependent to an altitude, or the clouds do not have to be volumetric and so on.

| **Shader** | **Instructions** | **Cycles** | **Pixel throughput (MP/s)** | **Frames/s** |
|---|---|---|---|---|
| Soil | 45 | 29.00 | 220.69 | 84 |
| Sky | 21 | 15.00 | 426.67 | 163 |
| SimpleGround | 14 | 12.00 | 533.33 | 203 |
| SimpleTexGround | 16 | 13.00 | 429.31 | 164 |
| SimpleTexTexGround | 35 | 22.00 | 290.91 | 111 |
| SimpleTexBGround | 22 | 16.00 | 400.00 | 153 |
| SimpleTexTexBGround | 48 | 28.75 | 228.57 | 87 |
| Clouds | 30 | 15.00 | 426.67 | 163 |
| Rain | 3 | 2.00 | 3200.00 | 1221 |
| RainDropsOnwindows | 15 | 6.00 | 1070.00 | 408 |

Table 5.1.: Performance assessment using the NVShaderperf tool from Nvidia [NVI05a]. The NV40 chip and driver version 77.72 is used as profile. The frames per second (Frames/s) are the overall frames per second and given under certain assumptions: screen resolution 1280x1024, current effect fills the whole screen, no CPU limitation due to bad Batching [Wlo03], vertex shader cuts the performance in half.

To evaluate the assumption that shaders improve the immersion we performed some perceptional experiments, where the test persons had to make a subjective decision on

the level of realism the shaders provide. We ran the tests in an immersive and in a non-immersive setup. It also was the goal to obtain a metric on the importance of every shader to be able to switch a shader off, in the case, where the hardware does not meet the performance requirements (e.g., no support for vertex/fragment shader). In order to provide an overview on the resulting shaders, Table 5.1 shows the performance of each shader on the target hardware, which was the NV40 chip of Nvidia (Geforce 6800 GT PCI-Express) on a Pentium 4 CPU and 2 GB DDR-RAM. The first column represents the fragment shader that has been tested. In the second column is the number of assembler instructions, which are generated using the Nvidia driver version 77.72 out of the GLSL shader. The third column represents the number of cycles the shader takes on the NV40 GPU to perform the instructions. The fourth column describes the pixel throughput in mega pixel per second. Finally, the last column represents the theoretical frames per second the fragment shader can reach based on the pixel throughput, if it is assumed that the screen resolution is 1280x1024 that the current effect fills the whole screen, and that no CPU limitation through bad *Batching* [Wlo03] is present. Unfortunately, the NVShaderperf is not able to process GLSL vertex shaders. Therefor, we assume that the vertex shaders we developed cut the overall performance in half in the worst case, though the shaders are developed to be fragment shader centric due to high pixel pipeline performance and effective SIMD architecture. The most simple fragment shader, with only one instruction to set the fragment color, compiles to one cycle and has a pixel throughput of 6400 mega-pixel per second (MP/s).

In this section we first discuss the Results of the soil shader and compare it to photos to see if photo realism can be reached using the described method. Then, we discuss the shaders responsible for rendering weather conditions and also compare them to real photos.

## 5.1. Soil

With the method described in section 3.1, it is possible to render a flat surface as a granular surface without an overhead of geometric complexity. Moreover, the color of the surface changes as soon as the depth of the surface changes. Two possible solutions were presented in this work to set the depth value for a vertex or region, either using a texture or a per-vertex attribute that can be saved e.g. in the texture coordinate attribute. Even though we recommend the per vertex attribute, we implemented in this work the height map look-up as the per-vertex attribute shader would be highly application specific, and at the time we write this work the VAR-Trainer visualization module is not ready for use.

In Figure 5.1 the virtual soil is compared to a real picture of soil. The change in horizon/color through the usage of a color gradient is highly efficient and creates good interpolated results. Unfortunately, it is not possible to use different textures for different horizons as real horizons not only change their color but also their texture. However, a change in the

texture would mean to change the surface granularity and this would in turn require a dynamic changing distance map that has to be calculated over and over. But the calculation is time consuming[1], and thus it would not be practicable for real-time rendering.

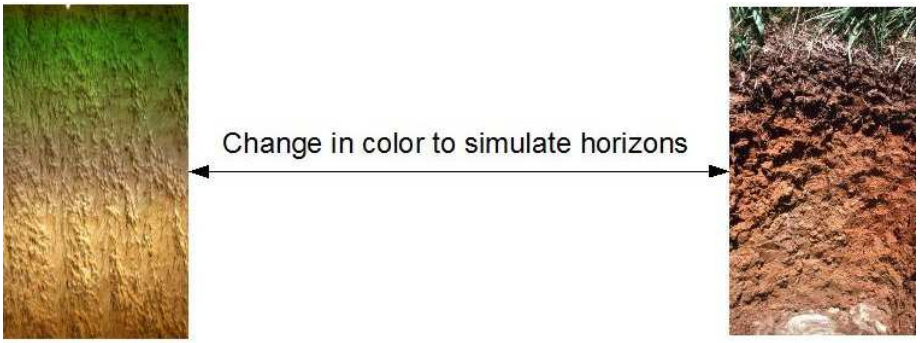


Figure 5.1.: A 1-dimensional texture is used to lookup the color of the soil. This results in different horizons.

The complexity of the mesh in the soil shader can have a very low resolution. An example is shown in Figure 5.2, where the mesh could also have been a simple quad or two triangles. The advantage of such a fragment centric shader is that it only computes the visible fragments. Also, fragment processors have more performance, which makes the decision to move to a per fragment computation even more feasible. Thus, a progressive mesh providing details only in regions of the mesh that are necessary would be the best solution for the soil shader.

The *NVShaderPerf* outputs information about the shader. Three useful outputs are: number of instructions, which defines the number of assembler instructions that has been compiled with the current OpenGL driver and current GPU. The number of cycles defines the number of cycles the instructions need to be executed on the GPU. Pixel throughput defines the amount of pixels that can be rendered using the shader. The output for the soil fragment shader is:

```
# 45 instructions, Cycles: 29.00, Pixel throughput: 220.69 MP/s
```

This is the most complex fragment shader implemented in this work, but it still has a high pixel throughput and reaches approximately 84 fps, though it computes the soil and the atmospheric scattering for accurate lighting.

[1]About 10 sec. on an AMD 3000+, where the texture is $512x512$ and the depth 16.

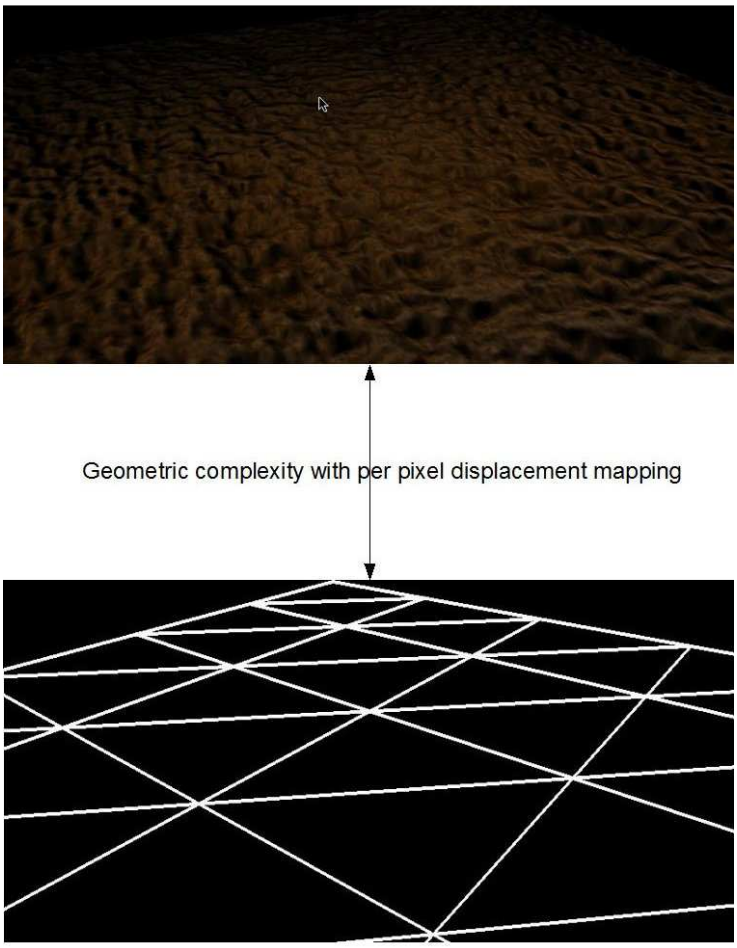


Figure 5.2.: The complexity of high frequency in the granularity of soil is simulated in the fragment processor and thus does not require complex polygon models.

## 5.2. Sunlight and Atmospheric Scattering

A correct lighting of the scene is one of the most important factors for realizing photo realism. Outdoor scenes have the advantage that the only light source is the sun, but as the sun travels through the atmosphere certain wavelengths are scattered and thus the sky is blue or distance objects loose contrast. Additionally, outdoor scenes have a high indirect lighting component, which is not treated in this work but can be simulated using an ambient factor that is added to the final color of every fragment.

The adapted method presented in section 3.2 provides a fast way to render the sky and atmospheric scattering. Its advantage is that many parameters are calculated only once on the CPU and are then used to apply on all objects in the scene. The disadvantage of such a method is that parameters has to be scripted to create an accurate effect of the

Figure 5.3.: Atmospheric perspective rendered with the shaders described in this work.

scattering, and even then it is not the physically correct color due to all simplifications that have been made. In [HP03], for instance, the sky and sunlight changes accurately due to a change of one parameter *east, west*, whereas in the shaders described in this work, a combination of parameters define such effects. However, the realized shader meets the requirements of the VAR-Trainer project, as the scenes in a training session should not change their environmental effects. Thus, an artist is able to define a scene and set the parameters to reach a desired effect. The artist might also connect the parameters together, using a scripting interface.

Figure 5.4.: Atmospheric perspective in a photo.

The resulting atmospheric effect is reached by applying the simpleground.glsl shader to all scene objects[2]. Doing this ensures almost accurate atmospheric perspective and exponential

[2]There are also modified versions of this shader realized to provide rendering of textured and bump-mapped

fog calculation. Figure 5.3 shows a result of a typical outdoor scene. Notice how distance mountains loose contrast and get blueish. The rendered version is a good approximation of real atmospheric scattering observed in Figure 5.4.

The *NVShaderPerf* output for the sky fragment shader is:

```
# 21 instructions, Cycles: 15.00, Pixel throughput 426.67 MP/s
```

This is theoretically more than 160 fps. The number of cycles is mainly because of the exponentiation in the fragment shader that provides the controlling of an exposure value for the virtual camera. This can and should be ported in the future into a HDR (High Dynamic Range) shader to have a much more realistic lighting of the scene [O'N05]. Unfortunately, OpenSG does not provide an easy way to render to textures at the time this work is written, which is essential for HDR rendering. The other exponentiation is in the calculation of the Mie phase term and cannot be avoided. Figure 5.5 shows how the overall scene impression can change by changing some atmosphere parameters. (Compare with Figure 5.3).

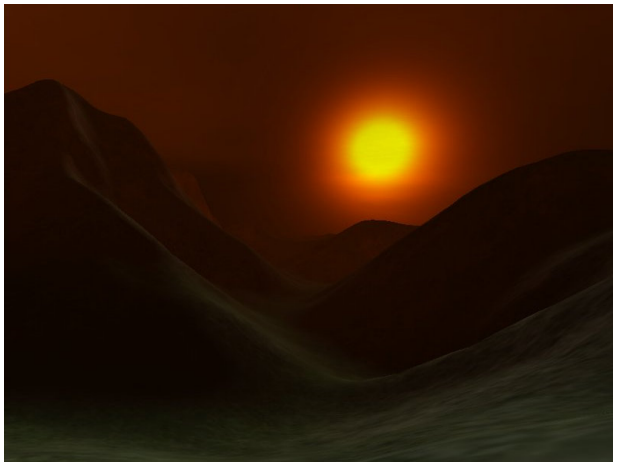

Figure 5.5.: Changing parameters of the atmospheric effect can result in dramatic scene impression. (Compare with Figure 5.3).

The *NVShaderPerf* output for the simpleGround fragment shader, which just renders the material color under the constraint of atmospheric effects is:

```
# 14 instructions, Cycles: 12.00, Pixel throughput 533.33 MP/s
```

With 14 instructions the shader still needs 12 cycles $\approx$ 203 fps, due to the same exponentiation explained above. Applying a texture to the shader (simpleTexGround) increases the number of cycles to 13 $\approx$ 164 fps:

objects that are affected by the atmospheric effects.

```
# 16 instructions, Cycles: 13.00, Pixel throughput 492.31 MP/s
```

In simpleTexTexGround the number of instructions increases because two textures are mixed together dependent on the envelope of the ground. This ensures smooth transitions from one texture into another at the cost of 22 cycles ≈ 111 fps:

```
# 35 instructions, Cycles: 22.00, Pixel throughput 290.91 MP/s
```

Then, two more modifications exist, which apply a bump map. SimpleTexBGround uses one texture and a bump map texture and needs 16 cycles ≈ 153 fps:

```
# 22 instructions, Cycles: 16.00, Pixel throughput 400.00 MP/s
```

SimpleTexTexBGround uses the mixed texture variant depending on the envelope of the mesh, and an additional bump map texture with the cost of 28.75 cycles ≈ 87 fps:

```
# 48 instructions, Cycles: 28.75, Pixel throughput 228.57 MP/s
```

Figure 5.6 shows the difference between the shaders.

## 5.3. Clouds

The results of the cloud shader is not very realistic as the shading of the clouds does not react on the position of the sun. However, it fulfills the needs for a construction site scene, since the worker does not focus on the sky for a long time, when she/he has to concentrate on her/his tasks. In Figure 5.7 you can see that the shading on the right is not accurate to real photos of clouds on the left. The movement of the clouds is credible and natural as it is based on [Dub05].

The cloud shader is with 15 cycles and approximated output of 163 fps not very expensive:

```
# 30 instructions, Cycles: 15.00, Pixel throughput 426.67 MP/s
```

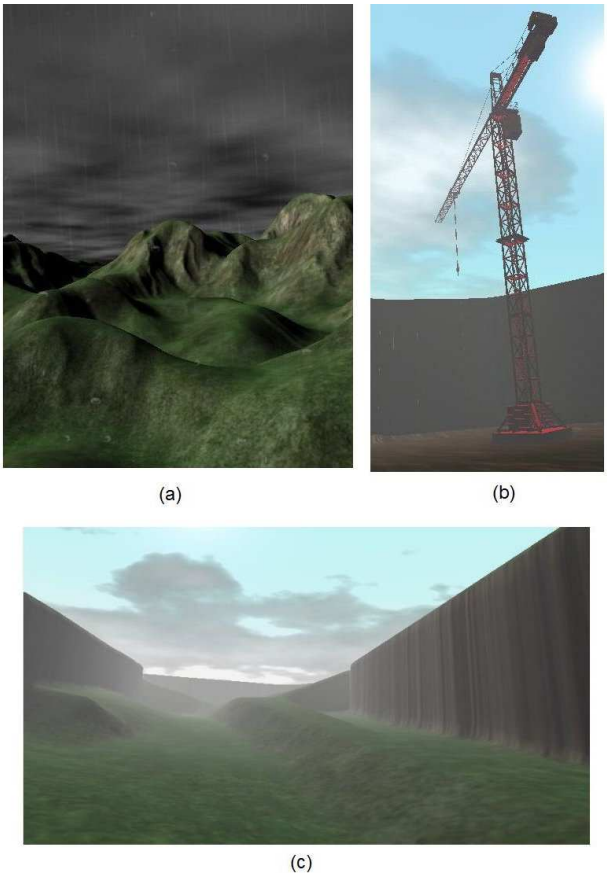


Figure 5.6.: In (a) the terrain uses *simpleTexTexBGround* shader to mix two textures and apply a bump-map to it. The crane in (b) uses the *simpleground* shader as the material has only a color value. The terrain in (c) uses *simpleTexTexGround* to mix two textures depending on the envelope of the terrain.

## 5.4. Rain

The result of the rain shader is credible, if a certain distance (about 3 meters) of the viewer from the ground is provided, allowing for missing splashes and puddles on the ground.

Rain is the cheapest fragment shader (theoretically more than 1k fps) as it only has to apply an animated texture:

```
# 3 instructions, Cycles: 2.00 Pixel throughput 3.20 GP/s
```

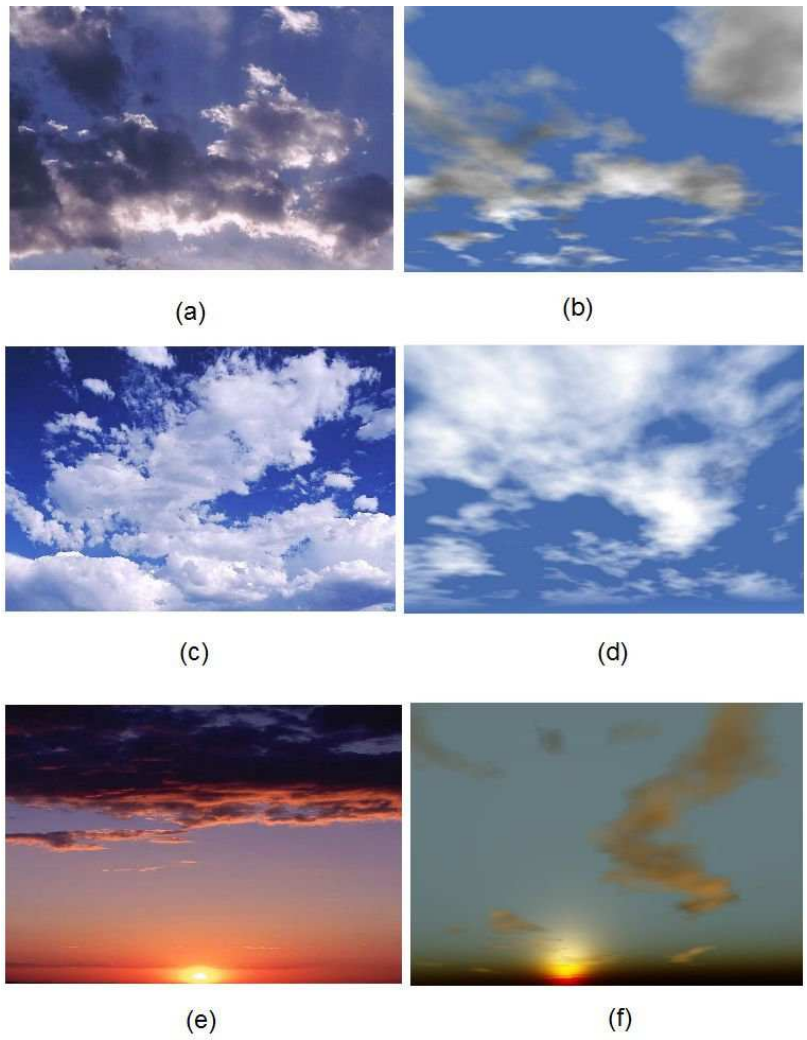


Figure 5.7.: Real clouds on the left (a, c, e) and simulated clouds on the right (b, d, f).

## 5.5. Raindrops on Window

The raindrops create more realism for a rainy scene seen through some transparent glass (Figure 5.9).

Raindrops is also a cheap fragment shader (theoretically more than 400 fps) as it only has to add an animated and masked texture together and apply it on a surface. The number of instructions increased over normal rain because of the smooth interpolation between masked and unmasked areas:

```
# 15 instructions, Cycles: 6.00 Pixel throughput 1.07 GP/s
```

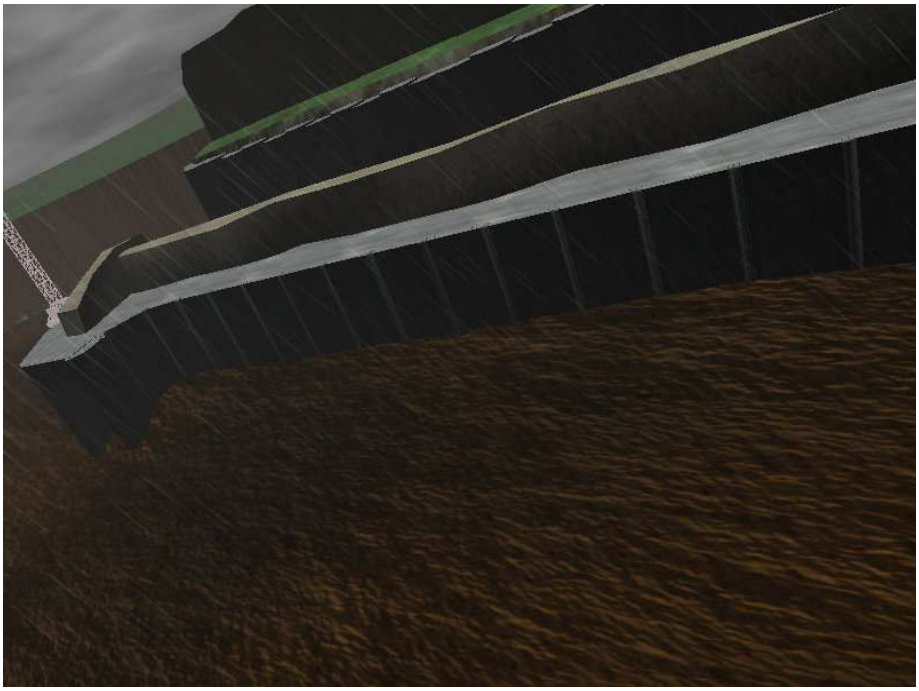

Figure 5.8.: Rain simulated through texture animation.

## 5.6. Perceptional Evaluation

This section is about the evaluation of the shaders developed within this work. We also give an importance metric about the shaders to allow a perceptional based decision on less powerful machines. Based on the metric, shaders that do not have much importance on realism can be turned off, and simple textures can be used instead.

Stokes references in [SFWG04] many papers describing perceptual metrics that can be used to establish stopping criteria for high quality rendering systems and using perceptual metrics to optimally manage resource allocation for efficient visualization. Those metrics are gathered either through perceptual experiments where people make subjective decisions on visualizations or through the difference of an image to a *gold standard* image [PF03]. The use of such a metric is to render highest possible quality images under given constraints. Particularly, Ahumada [Ahu93] and Ramasubramania [Ram00] provide good reviews of computational image quality metrics developed by computer graphics and vision science fields.

To be more clear about the "realism", Ferwerda defined "Three Varieties of Realism in Computer Graphics" [Fer94] :

- **physical realism** in which the image provides the same **visual stimulation** as the scene

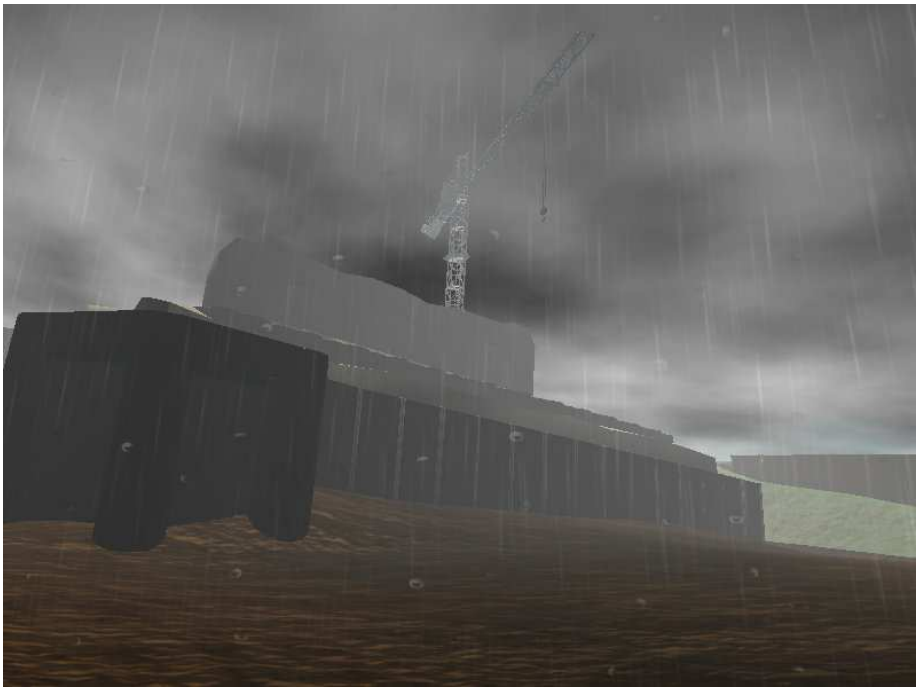

Figure 5.9.: Raindrops on Window.

- **photo realism** in which the image produces the same **visual response** as the scene and
- **functional realism** in which the image provides the same **visual information** as the scene

Photo realism is mostly a manner of measuring the quality of the photometry and thus the eye's response to light energy. This measurement is difficult on existing displays as they can not reproduce the vast ranges of light energy found in different scenes. Thus, the experiments we make in this work will result in a metric, based on the subjective and spontaneous opinion about rendered images.

### 5.6.1. Experimental Setup

Two experimental setups were tested within this project: An immersive stereoscopic setup which consists of a head mounted display (HMD), and a non-immersive setup with a 17" monitor. As the HMD had a built-in tracking feature, the user was able to rotate the virtual camera by rotating his/her head. The user was able to navigate through the scene. During the experiments, we asked questions about subjective perception of a certain effect, where the users had to evaluate the photo realism component of a certain effect based on their personal experience in a range from $[0, 9]$ where 0 means that "it cannot even be

identified" and 9 means "it looks that real as if we can touch it". Another question we asked them was about the importance (in the range of $[0, 1]$) of the components sky, clouds, soil and rain, if they imagine they sit in a construction machine and be on a construction site. Finally, the user had to close the eyes. In the meantime, we switched between shader mode and simple texture (non-shader) mode to see if the user recognizes the change in the scene. We used this procedure for the sky, clouds and soil effect to evaluate their importance in a dynamic scene. We ran the experiments with 21 test persons from which half of them were computer graphics scientists and the others were computer gamers. Thus, the test persons knew more or less about simulation and computer graphics, which made the conditions harder. Every test person did only either the immersive or the non-immersive test. With this we ensured that no test person might have expectations about a HMD or a simple monitor.

### 5.6.2. Experimental Results

The results of the first series of tests with the HMD are shown in Figure 5.10. The clouds are the most realistic shader in the immersive experiment series, though they are not accurately shaded. In the non-immersive experiment series, the sky is the most realistic shader. The soil shader is in both experiment series the worst shader, though it has still higher than 5 points.

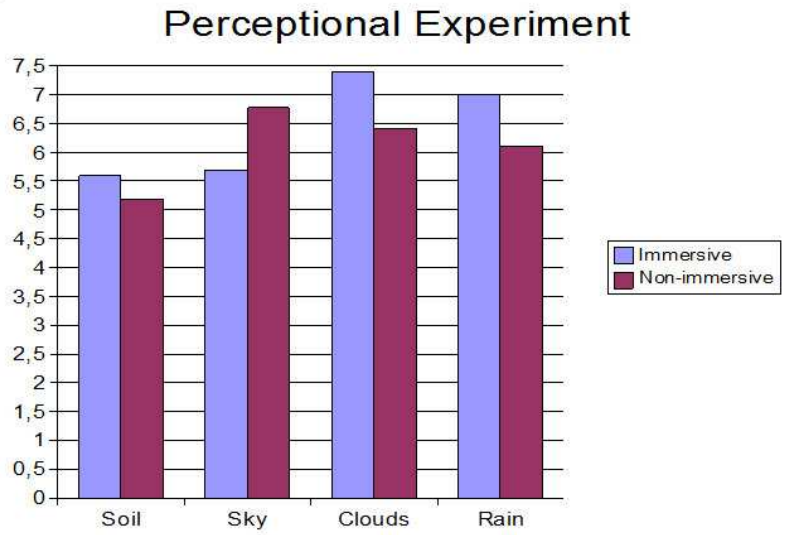


Figure 5.10.: Results of the first series of tests show the ranking of the photo realism of shaders in the range $[0, 9]$ where 0 means bad photo realism, and 9 means high photo realism.

However, this was the results of rating the realism of the shaders, but not the importance of the shaders. The results of the importance of every shader on a construction site were:

Soil (0.9), Sky (0.2), Clouds(0.4), Rain (0.7). Figure 5.11 shows how the metric changes the order of the effects.

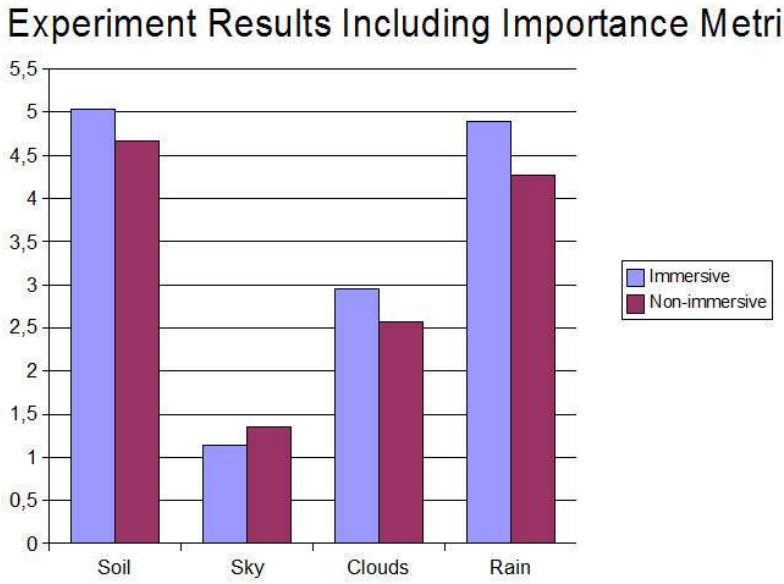


Figure 5.11.: Results of the ranking after applying the importance metric to it.

In the second part of the experiments 37% of the test persons did not recognize the change from shader rendered sky and clouds to simple texture mapped sky and clouds (Figure 5.13). Moreover, 48% of the test persons realized that something has changed in the sky, but they could not say for sure what it was. "Maybe the color, or the wind direction[...]", "Different time of day[...]", "[...]less clouds.", were a few answers. Only 15% recognized that the sky has totaly changed and the clouds are now not moving. On the other side 100% of the test persons recognized the change of the soil shader in a texture (same texture used to create the soil shader maps Figure 5.14). Figure 5.12 shows the final metric after the results of the second series of experiments have been applied to it.

This means that the ranking of the shaders according to the metric is in the following order:

1. **Soil and Simple*Ground shader** is the most important shader. Although the soil shader is not the most realistic shader described in this work, it still provides a higher level of realism. The Simple*Ground shaders are responsible for the atmospheric perspective and fog and thus it is clear that loosing this kind of effect decreases the quantity of realism.[3]

[3]Though exponential fog can be simulated using the standard fog in the fixed pipeline, it never can simulate the atmospheric perspective.

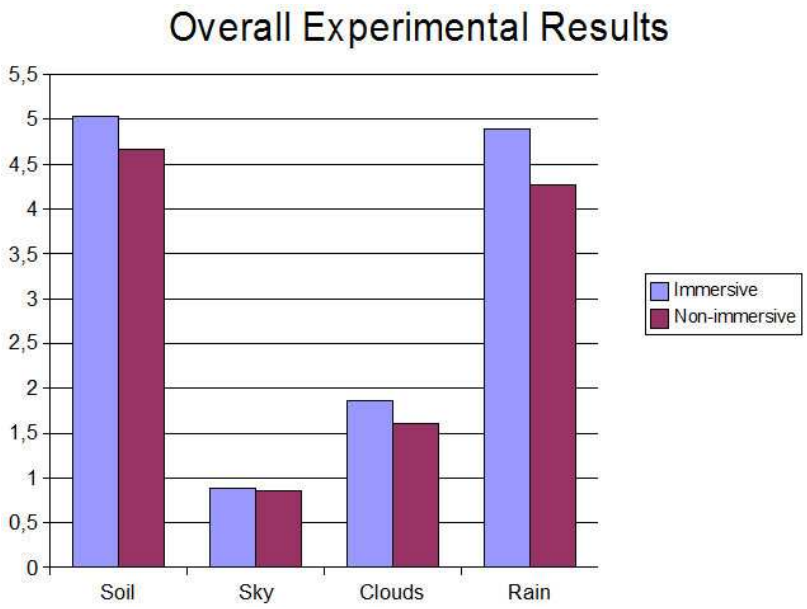


Figure 5.12.: Overall results after applying the results of the second series of experiments to the metric.

2. **Rain shader** can be adapted to some level completely in the fixed pipeline, but would require computations to be made per-frame on the CPU, which we wanted to avoid. It is clear that a rain effect is highly important to simulate a rainy day. Thus this is the equally important with the soil shader, if a rainy day has to be simulated.

3. **Clouds shader** is important for a credible cloud movement and a feeling of time. It creates a more dynamic than static rendered scene. But as the worker does not focus on them, they could also be static and change for instance when the worker focus on the ground.

4. **Sky shader** is important, if the time of the day changes rapidly or the simulation runs over hours. But it is also not very important for the purpose of VAR-Trainer as the worker focuses on the ground.

The results of the experiment showed that shaders bring more realism into a simulated construction machinery environment, but they also showed that some visual effects are highly application dependent. For a mostly static environment where the viewer does not often look up into the sky, a static sky-dome or sky-box is sufficient. There exist great tools providing rendering of such environmental image maps, e.g. Terragen [Pla05]. However, you might not get accurate results as, for example, the atmospheric perspective depends on color of the sky. In this case it can be difficult to generate a correct sky color with accurate color transitions and atmospheric results.

Figure 5.13.: Switch between textured (left) and shaded (right) sky and clouds were not recognized by 47% of the test persons.

Figure 5.14.: Switch between textured (left) and shaded (right) soil was clearly recognized by 100% of the test persons.

## 5.7. Summary

In this chapter we first discussed the results of the shaders in a technical manner, where we used the Nvidia *NVShaderPerf* tool [NVI05a] to test the performance of the shaders on the target hardware. Unfortunately, only the fragment shaders could be measured as the GLSL vertex shaders are not supported in *NVshaderPerf*. However, we showed that most shaders use less than 20 cycles per-fragment, and fulfill the requirements of a real time application. Moreover, the test application reached over 50 frames per second (fps) with all shaders visible, and thus leave enough resources for other requirements. We also discussed the results in a subjective perceptional manner, and compared the solutions of the rendering with real photos. This showed that the level of photo realism can not be reached fully but some effects can be simulated in real-time and have a credible look

compared to the reality.

We also built a metric based on perceptional experiments. With this metric it is possible to decide if an effect is important for realism, or if it can be left off when the GPU is bounded and not capable to run the application with high frame rates. The results of the experiment also provided a metric for subjective perception of realism for the shaders presented in this work and how they improve the simple texture mapping technique. The ranking of the shaders is:

1. Soil and Simple*Ground effects
2. Rain
3. Clouds
4. Sky

# Chapter 6.

# Conclusions & Future Work

In this chapter we review this work, and give short statements of the inferences encountered during the realization of this work. Based on the results of the experiments in the previous chapter we answer the main question in this work: is there an improved perception in a construction machinery simulator with shaders? Finally, we also include a section about possible future research based on our results.

Improving the look of surfaces using textures was the first stage toward realism in real time visualization [TMR99a]. As the graphic chips improved more and more their performance, and gave the ability of programmable vertex and fragment shaders, even more realism could and can be obtained. Thus, improving the perception in a construction machinery simulator using the shader technology is definitely practicable and is worth the work for reaching a higher level of realism. Actually, this was the main objective of this work. However, the experiments showed that in some cases the effort of shader utilization is not compelling and is highly application specific. In fact, developing an environment for a specific simulator needs at first a design process, where the most seen materials and effects should be summarized and categorized to focus the effort on them. Then, the decision has to be made if the material properties can be achieved using simple techniques like: phong shading or texture mapping [TMR99a]. If not, shaders should be used to achieve the desired effect. In VAR-Trainer [VT05], a scene should be able to simulate/visualize realistic weather conditions and soil as those are the most seen environmental conditions and material in a construction machinery site. We showed that most existing simulators do not use techniques to obtain the level of photo realism that is reachable when the techniques shown in this work are used for the environment of an construction machinery simulator. Most of the simulators only provide functional realism even if the needed hardware exists.

In chapter 3 the approaches for soil and weather condition rendering were described, where we combined and improved existing methods to reach the desired effects. Soil is an adapted method of [Don05] for rendering high frequency, granular surfaces. The color of soil varies with its horizon and thus from which depth it is. This information can either be stored in a height map or in one of the texture coordinate attributes every vertex can have. This

information is normalized and used to look-up in a pre-designed color gradient texture to simulate smooth changes in the color.

The atmospheric effect realized is an adaption of [O'N05], which is capable to render the sky, sun and atmospheric effects like fog, haze and atmospheric perspective. Again, we improved the method for the needs of VAR-Trainer, and added more possibilities, like position of the sun, intensity of haze at the horizon, etc. to influence the result.

The clouds are based on the idea of [Dub05], but also improved for the specific needs of a construction machine simulator. The clouds are 2.5 dimensional and built using noise textures of different octaves. By adding weighted and scaled octaves together, subtracting low values from the result and exponentiating all together, the output results in a credible cloud texture. In order to improve the outcome we introduced a new way of shading the clouds with simple average operations with other clouds textures.

Furthermore, we realized an adaption of the method from [Wan05], which renders rain by mapping a texture on a double cone transforming the texture coordinates with a matrix. In addition, we implemented a method to render rain drops on virtual windows by masking a rain drops texture with a noise texture.

In order to evaluate the results of this work and the initial assumptions we conducted a series of experiments. The experiments consisted of two setups: immersive and non immersive. Thus we not only compared shader vs. non shader (texture mapping) but also immersive shader vs. immersive non shader and vise versa. Based on the evaluation, we built a metric to decide which shader is responsible for the highest credibility, and which shader can be switched off on a less performant GPU.

Finally, we evaluated the performance of the shaders, and the improved perception of shaders versus textures The result is that most shaders use less than 20 cycles per-fragment, and fulfill the requirements of a real time application. Moreover, the test application reached over 50 fps with all shaders visible.

We also evaluated the results in a subjective perceptional manner and compared the solutions of the rendering with real photos. The result is that the level of photo realism can not be reached fully, but some effects can be simulated in real-time and have a credible look compared to the reality. The perceptional evaluation showed that in some cases a shader does not absolutely achieve a better result than a simple texture mapping. The experiments also showed that there is a small difference between immersive and non-immersive visualization.

Moreover, the results of the experiments allowed to build a perceptional metric of the importance of the implemented shaders. This metric can be used to decide which shader could be replaced by simple texture mapping, e.g. because of high GPU load. The ranking of the shaders based on the metric is:

1. Soil and Atmospheric Effects

2. Rain
3. Clouds
4. Sky

The main problem the experiments caused was the question: how do we measure photo realism?. The solution we then used to measure photo realism was a subjective perceptional experiment where the test person should decide whether the rendered output is as real as the reality they have experienced or not [SFWG04]. Some shaders used in this work are adapted from other existing shaders, but optimized for the VAR-Trainer project. However, there are still some improvements that could be made on the shaders.

**The sky shader:**

- It is not highly accurate due to the combined Mie and Rayleigh Phase Function ( see subsection 3.2.1 or [HP03]) in the adopted Henyey-Greenstein Phase Function. This topic is also addressed by O'Neil [O'N05].
- Heuristically we found it sometimes hard to set the correct parameters for a specific look of the sky. A scripted solution of the sky with less parameters (e.g. *east* and *west*) would be a more easy way for the dynamic animation of the sky and quick changes in a scene editor.
- The implementation of high dynamic range (HDR) would make the results more realistic.
- The definition of haze should be variable like in [Nie03].
- Fog in a dusk or dawn effect could be realized implementing an accumulation of the hardware fog depending on the position of the sun.

**The cloud shader:**

- It fakes the shading of the clouds with good results but does not reach the level of realism of a real picture. A better result could be achieved using the method presented by Dube [Dub05]. However, his method requires shader model 3.0 (SM 3.0) and high computational power on the fragment processor. The inner path tracing can be made faster using a preprocessed distance map, like in the soil shader and thus converting the path tracing into a sphere tracing.

**The rain shader:**

- Rain splashes on the ground and on all objects would be a great improvement.
- Rain drops on windows should refract the background and run over the window.
- Both are realized in the ATI [ATI05] *Toyshop*.

**The soil shader:**

- Changing the granularity is a matter of changing the whole underlying texture and thus not practicable when the granularity should go smooth from one size to another. The only way to do this would be to split the mesh in granularity regions and apply desired textures to every region. However, this would generate a seam at the edges. If necessary this can be avoided at higher computational cost by rendering the mesh twice (or as often as granularities are specified) but setting a blend factor at the regions with different granularities. This allows a smooth blending from one texture into another.

What is definely missing in this work for more photo realism are soft shadows for all scene objects and cloud shadows. For now it is possible to decrease or increase the overall light perceived from the scene with the *exposure* parameter. However, this should be connected to the *cloud cover* parameter to provide realistic reaction of sunlight with clouds.

# Bibliography


[Ahu93] Jr. Albert J. Ahumada. Computational image quality metrics: A review. Technical Report 24, Society for Information Display International Symposium Digest of Technical Papers, 1993.

[Ake93] Kurt Akeley. Reality engine graphics. In *SIGGRAPH '93: Proceedings of the 20th annual conference on Computer graphics and interactive techniques*, pages 109–116, New York, NY, USA, 1993. ACM Press.

[ATI05] ATI. www.ati.com, last visited: October. 2005.

[BLH02] Brook Bakay, Paul Lalonde, and Wolfgang Heidrich. Real time animated grass. Technical report, EUROGRAPHICS, 2002.

[Bli78] James F. Blinn. Simulation of wrinkled surfaces. In *Computer Graphics (SIGGRAPH '78 Proceedings)*, volume 12, pages 286–292, August 1978.

[Bro05] Martin Brownlow. *Widgets: Rendering Fast and Persistent Foliage*, volume 5 of *Game Programming Gems*, chapter 5.3, pages 515–526. Charles River Media, 2005.

[Cla05] Ryan Clark. Displacement generator. www.ryanclark.net, last visited October. 2005.

[Coo84] Robert L. Cook. Shade trees. In *SIGGRAPH '84: Proceedings of the 11th annual conference on Computer graphics and interactive techniques*, pages 223–231, New York, NY, USA, 1984. ACM Press.

[Dan80] Per-Erik Danielsson. Euclidean distance mapping. *Computer Graphics and Image Processing*, 14:227–248, 1980.

[Dir05] DirectX. Microsoft. http://www.microsoft.com/windows/directx/default.aspx, last visited October. 2005.

[Don05] William Donnelly. *Per-Pixel Displacement Mapping with Distance Functions*, volume 2 of *GPU Gems*, chapter 8, pages 123–136. Addison-Wesley, March 2005.

[Dub05] Jean-Francois Dube. *Realistic Cloud rendering on Modern GPUs*, volume 5 of *Game Programming Gems*, chapter 5.1, pages 499–505. Charles River Media, 2005.

[Fer94] James Ferwerda. Three varieties of realism in computer graphics. citeseer.ist.psu.edu/699574.html, 1994.

[Geh05] Michael Gehling. Darstellung und animation eines himmels mit wolken und himmelskörpern in einer 3d-echtzeitumgebung. Master's thesis, Fachhochschule Braunschweig / Wolfenbüttel, 2005.

[Har96] J. C. Hart. Sphere tracing: A geometric method for the antialiased ray tracing of implicit surfaces. *The Visual Computer*, 12(10):527–545, 1996.

[HBSL03] Mark J. Harris, William V. Baxter, Thorsten Scheuermann, and Anselmo Lastra. Simulation of cloud dynamics on graphics hardware. In *HWWS '03: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, pages 92–101, Aire-la-Ville, Switzerland, Switzerland, 2003. Eurographics Association.

[HL01] Mark J Harris and Anselmo Lastra. Real-time cloud rendering. Technical report, University of North Carolina at Chapel Hill, Chapel Hill, NC, USA, 2001.

[HP03] Naty Hoffman and Arcot J. Preetham, editors. *Real-time light-atmosphere interactions for outdoor scenes*, Rockland, MA, USA, 2003. Charles River Media, Inc.

[INR05] INRETS. Driving simulators links. http://www.inrets.fr, last visited: October 2005.

[IR02] John Isidoro and Guennadi Riguer. *Texture Perturbation Effects*, volume 2 of *ShaderX*, chapter 3, pages 212–220. Wordware Publishing, 2002.

[KF05] Emmet Kilgariff and Randima Fernando. *The Geforce 6 Series GPU Architecture*, volume 2 of *GPU Gems*, chapter 30, pages 471–491. Addison-Wesley, 2005.

[KSW04] Peter Kipfer, Mark Segal, and Rüdiger Westermann. Uberflow: a gpu-based particle engine. In *HWWS '04: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, pages 115–122, New York, NY, USA, 2004. ACM Press.

[Lae05] Jesse Laeuchli. *Volumetric Clouds*, volume 3 of *ShaderX*, chapter 8.4, pages 611–616. Charles River Media, 2005.

[LAN05] LANDER. http://www.landersimulation.com, last visited: October. 2005.

[LKO05] Aaron Lefohn, Joe Kniss, and John Owens. *Implementing Efficient Parallel Data Structures on GPUs*, volume 2 of *GPU Gems*, chapter 33, pages 521–544. Addison-Wesley, 2005.

[Max88] Nelson L. Max. Horizon mapping: shadows for bump-mapped surfaces. *The Visual Computer*, 4(2):109–117, 1988.

[Mic05] Microsoft. Encarta encyclopedia, 2005.

[Nie03] Ralf Stokholm Nielsen. Real time rendering of atmospheric scattering effects for flight simulators. Master's thesis, Technical University of Denmark, 2003.

[Nor05] University Simulator Northeastern. http://www1.coe.neu.edu/ mourant/ve-lab.html, last visited: October 2005.

[NSTN93] Tomoyuki Nishita, Takao Sirai, Katsumi Tadamura, and Eihachiro Nakamae. Display of the earth taking into account atmospheric scattering. In *SIGGRAPH '93: Proceedings of the 20th annual conference on Computer graphics and interactive techniques*, pages 175–182, New York, NY, USA, 1993. ACM Press.

[NVI05a] NVIDIA. http://developer.nvidia.com, last visited: October. 2005.

[NVI05b] NVIDIA. Ogres and fairies: Secrets of the nvidia demo team. http://developer.nvidia.com/docs/, last visited: October. 2005.

[Okt05] Oktal. Scanner 2. http://www.scaner2.com, last visited: October. 2005.

[O'N05] Sean O'Neil. *Accurate Atmospheric Scattering*, volume 2 of *GPU Gems*, chapter 16, pages 253–268. Addison-Wesley, 2005.

[OPE05a] OPENGL. Opengl. www.opengl.org, last visited October. 2005.

[Ope05b] OpenSceneGraph. http://www.openscenegraph.org/, last visited: October. 2005.

[Ope05c] OpenSG. Opensg forum. http://www.opensg.org, last visited October. 2005.

[Owe05] John Owens. *Streamin Architectures and Technology Trends*, volume 2 of *GPU Gems*, chapter 29, pages 457–470. Addison-Wesley, 2005.

[PC01] Frank Perbet and Maric-Paule Cani. Animating prairies in real-time. In *SI3D '01: Proceedings of the 2001 symposium on Interactive 3D graphics*, pages 103–110, New York, NY, USA, 2001. ACM Press.

[Pel04] Kurt Pelzer. *Rendering Countless Blades of Waving Grass*, volume 1 of *GPU Gems*, chapter 8. Addison-Wesley, 2004.

[PF03] REYNALD DUMONTand FABIO PELLACINI and JAMES A. FERWERDA. Perceptually-driven decision theory for interactive realistic rendering. Technical report, Cornell University, 2003.

[Phy05] CMLABS Simulating Physics. Vortex. http://www.cm-labs.com, last visited: October. 2005.

[PIX05] PIXAR. Renderman. https://renderman.pixar.com/, last visited October. 2005.

[Pla05] Planetside. Terragen. www.planetside.co.uk/terragen, last visited October. 2005.

[POC05] F. Policarpo, Manuel M. Oliveira, and J. L. D. Comba. Real-time relief mapping on arbitrary polygonal surfaces. In *SI3D '05: Proceedings of the 2005 symposium on Interactive 3D graphics and games*, pages 155–162, New York, NY, USA, 2005. ACM Press.

[PSS99] A. J. Preetham, Peter Shirley, and Brian Smits. A practical analytic model for daylight. In *SIGGRAPH '99: Proceedings of the 26th annual conference on Computer graphics and interactive techniques*, pages 91–100, New York, NY, USA, 1999. ACM Press/Addison-Wesley Publishing Co.

[Ram00] Mahesh Ramasubramanian. A perceptually based physical error metric for realistic image synthesis. Master's thesis, Cornell University, 2000.

[Ree83] W. T. Reeves. Particle systems - a technique for modeling a class of fuzzy objects. *ACM Trans. Graph.*, 2(2):91–108, 1983.

[Ros04] Randi J. Rost. *OpenGL Shading Language*. Addison Wesley, 2004.

[SAR05] SARTURIS. Technische universitaet darmstadt. http://www.sarturis.de/sarturis.html, last visited: October. 2005.

[SC00] Peter-Pike Sloan and Michael F. Cohen. *Interactive Horizon Mapping*. Eurographics Rendering Workshop, June 2000.

[SFWG04] William A. Stokes, James A. Ferwerda, Bruce Walter, and Donald P. Greenberg. Perceptual illumination components: a new approach to efficient, high quality global illumination rendering. *ACM Trans. Graph.*, 23(3):742–749, 2004.

[Sim05a] Civilian Driving Simulator. E-com. http://www.e-comsystems.cz/civilian.htm, last visited: October. 2005.

[SIM05b] SIMUSYS. University of zaragoza. http://laimuz.unizar.es/simusys/, last visited: October 2005.

[STS05] STSoftware. Car driving simulation software. http://www.stsoftware.nl, last visited: October 2005.

[SW99] Sonia Starik and Michael Werman. Simulation of rain in videos, 1999.

[Tec05] Immersive Technologies. Virtual environments. http://www.immersivetechnologies.com, last visited: October. 2005.

[TMR99a] David Blythe Tom Mc Reynolds. Lighting and shading techniques for interactive applications. *SIGGRAPH Course Notes*, pages 40–77, August 1999.

[TMR99b] David Blythe Tom Mc Reynolds. Lighting and shading techniques for interactive applications. *SIGGRAPH Course Notes*, pages 72–77, August 1999.

[TMR99c] David Blythe Tom Mc Reynolds. Lighting and shading techniques for interactive applications. *SIGGRAPH Course Notes*, page 103, August 1999.

[TMR99d] David Blythe Tom Mc Reynolds. Lighting and shading techniques for interactive applications. *SIGGRAPH Course Notes*, page 59, August 1999.

[TMR99e] David Blythe Tom Mc Reynolds. Lighting and shading techniques for interactive applications. *SIGGRAPH Course Notes*, pages 95–110, August 1999.

[Tom01] et al. Tomomichi Kaneko, editor. *Detailed Shape Representation with Parallax Mapping*. ICAT, 2001.

[TRU05] TRUCKSIM. http://www.trucksim.co.uk/, last visited: October. 2005.

[VST05] VSTEP. Driving simulator. http://www.vstep.nl, last visited: October. 2005.

[VT05] VAR-Trainer. http://www.ikerlan.es/vartrainer/default.htm, last visited: November. 2005.

[VTI05] VTI. Driving simulators. http://www.vti.se, last visited: October 2005.

[Wan03] Niniane Wang. Realistic and fast cloud rendering in computer games. In *GRAPH '03: Proceedings of the SIGGRAPH 2003 conference on Sketches & applications*, pages 1–1, New York, NY, USA, 2003. ACM Press.

[Wan05] Niniane Wang. *Let It Snow, Let It Snow, Let It Snow (and Rain)*, volume 5 of *Game Programming Gems*, chapter 5.2, pages 507–513. Charles River Media, 2005.

[Wel04] Terry Welsh. Parallax mapping with offset limiting: a per–pixel approximation of uneven surfaces. Technical report, Infiscape Corporation, 2004.

[Wik05] Wikipedia. http://wikipedia.org, last visited: October. 2005.

[Wlo03] Mathias Wloka. Batch, batch , batch: What does it really mean? Technical report, Nvidia, 2003.

[WWT+03] Lifeng Wang, Xi Wang, Xin Tong, Stephen Lin, Shimin Hu, Baining Guo, and Heung-Yeung Shum. View-dependent displacement mapping. *ACM Trans. Graph.*, 22(3):334–339, 2003.

[WWZ+05] Changbo Wang, Zhangye Wang, Qi Zhou, Chengfang Song, Yu Guan, and Qunsheng Pen. Dynamic modeling and rendering of grass wagging in wind. *COMPUTER ANIMATION AND VIRTUAL WORLDS*, 16:377–389, 2005.

[XXS+04] Wang X., Tong X., Lin S., Hu S., Guo B., and Shum H.-Y. Generalized displacement maps. *Eurographics Symposium on Rendering*, pages 227–233, 2004.

# Appendix A.

# OpenSG Source Code Snippets

## A.1. Basic OpenSG Scene

Listing A.1: Basic OpenSG application [Ope05c]

```
//Some needed include files
#include <OpenSG/OSGConfig.h>
#include <OpenSG/OSGGLUT.h>
#include <OpenSG/OSGSimpleGeometry.h>
#include <OpenSG/OSGGLUTWindow.h>
#include <OpenSG/OSGSimpleSceneManager.h>

//In most cases it is useful to add this line,
//otherwise every OpenSG command must be preceeded by an extra OSG::
OSG_USING_NAMESPACE
//The SimpleSceneManager to manage simple configurations
SimpleSceneManager *mgr;
//forward declaration
int setupGLUT( int *argc, char *argv[] );
NodePtr createScene();

int main(int argc, char **argv)
{
    // Init the OpenSG subsystem
    osgInit(argc,argv);
    // We create a GLUT Window
    int winid = setupGLUT(&argc, argv);
    GLUTWindowPtr gwin= GLUTWindow::create();
    gwin->setId(winid);
    gwin->init();
    // This is the whole scene
    NodePtr scene = createScene();
    //Create a scene manager, which is a helper class
    mgr = new SimpleSceneManager;
    mgr->setWindow(gwin);
    mgr->setRoot(scene);
    mgr->showAll();
    //Give Control to the GLUT Main Loop
    glutMainLoop();
    return 0;
}
//create the scene, e.g. a torus
NodePtr createScene()
{
```

```
        return makeTorus(.5, 2, 16, 16);
}
// react to size changes
void reshape(int w, int h)
{
    mgr->resize(w, h);
    glutPostRedisplay();
}
// just redraw our scene if this GLUT callback is invoked
void display(void)
{
                // internally this call traverses the graph and
                // renders all renderable nodes
    mgr->redraw();
}
//The GLUT subsystem is set up here. This is very similar
//to other GLUT applications.
int setupGLUT(int *argc, char *argv[])
{
    glutInit(argc, argv);
    glutInitDisplayMode(GLUT_RGB | GLUT_DEPTH | GLUT_DOUBLE);
    int winid = glutCreateWindow("OpenSG First Application");
    // register the GLUT callback functions
    glutDisplayFunc(display);
    glutReshapeFunc(reshape);
    return winid;
}
```

## A.2. OpenSG Scene Set-Up for GLSL Utilization

Listing A.2: Simple scene with GLSL Shader in OpenSG [Ope05c]

```
createScene()
{
  //load an image file as texture
  ImagePtr textureImage = Image::create();
  if(!grassImage->read("texture.jpg"))
  {
      fprintf(stderr, "Couldn't read texture\n");
      assert(0);
  }
  //prepare texture and set OpenGL Min-Mag parameters
  //(see OpenGL Red-Book for details on parameters)
  TextureChunkPtr tex_texture = TextureChunk::create();
  beginEditCP(tex_texture);
      tex_texture->setImage(textureImage);
      tex_texture->setMinFilter(GL_LINEAR_MIPMAP_LINEAR);
      tex_texture->setMagFilter(GL_LINEAR);
      tex_texture->setWrapS(GL_REPEAT);
      tex_texture->setWrapT(GL_REPEAT);
      tex_texture->setEnvMode(GL_MODULATE);
  endEditCP(tex_grass);

  //create the ambient color
  Vec4f ambientCol = Vec4f(0.2f, 0.4f, 0.3f, 1.0f);

  // create the shader material
  ChunkMaterialPtr cmat = ChunkMaterial::create();
```

```
  SHLChunkPtr _shl = SHLChunk::create();
  beginEditCP(_shl);
      //load the GLSL shader files
      if(!_shl->readVertexProgram("simple-vertex-shader.glsl"))
          fprintf(stderr, "Couldn't read vertex program\n");
      if(!_shl->readFragmentProgram("simple-fragment-shader.glsl"))
          fprintf(stderr, "Couldn't read fragment program\n");
      //set the uniform parameter (ambientColor) for the shader
      _shl->setUniformParameter("ambientColor", ambientCol);
      //set the uniform parameter (sampler myTexture)
      // the zero means that it is the first texture Id
      //handled by the material
      _shl->setUniformParameter("myTexture", 0);
  endEditCP(_shl);
  //set the material to use shader and texture
  beginEditCP(cmat);
      cmat->addChunk(_shl);
      //gives Id=0 for use in the shader
      cmat->addChunk(tex_texture);
  endEditCP(cmat);
  //create the scene (a torus)
  NodePtr scene = makeTorus(.5, 2, 16, 16);
  //get the Geometry of the torus
  GeometryPtr torusGeo = GeometryPtr::dcast(scene->getCore());
  //set the material for the torus
  beginEditCP(torusGeo, Geometry::MaterialFieldMask);
      torusGeo->setMaterial(cmat);
  endEditCP(torusGeo, Geometry::MaterialFieldMask);
  //return the scene ( a textured torus)
  return scene;
}
```

# Appendix B.

# UML Diagrams

## B.1. Class Diagrams

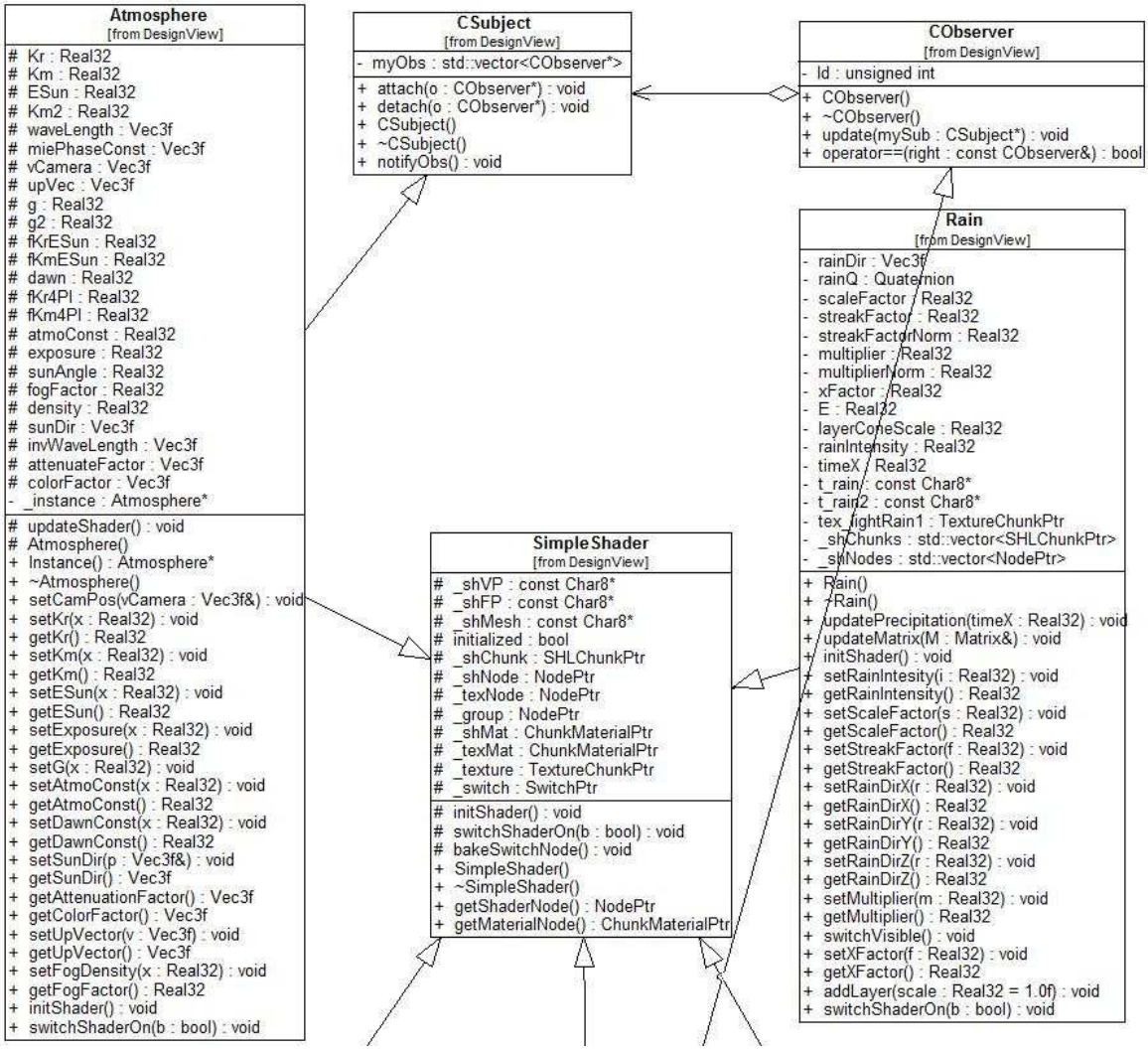


Figure B.1.: UML Class Diagram of the shader classes showing observer pattern, the SimpleShader class from which all other shaders are derived from, Atmosphere and Rain class.

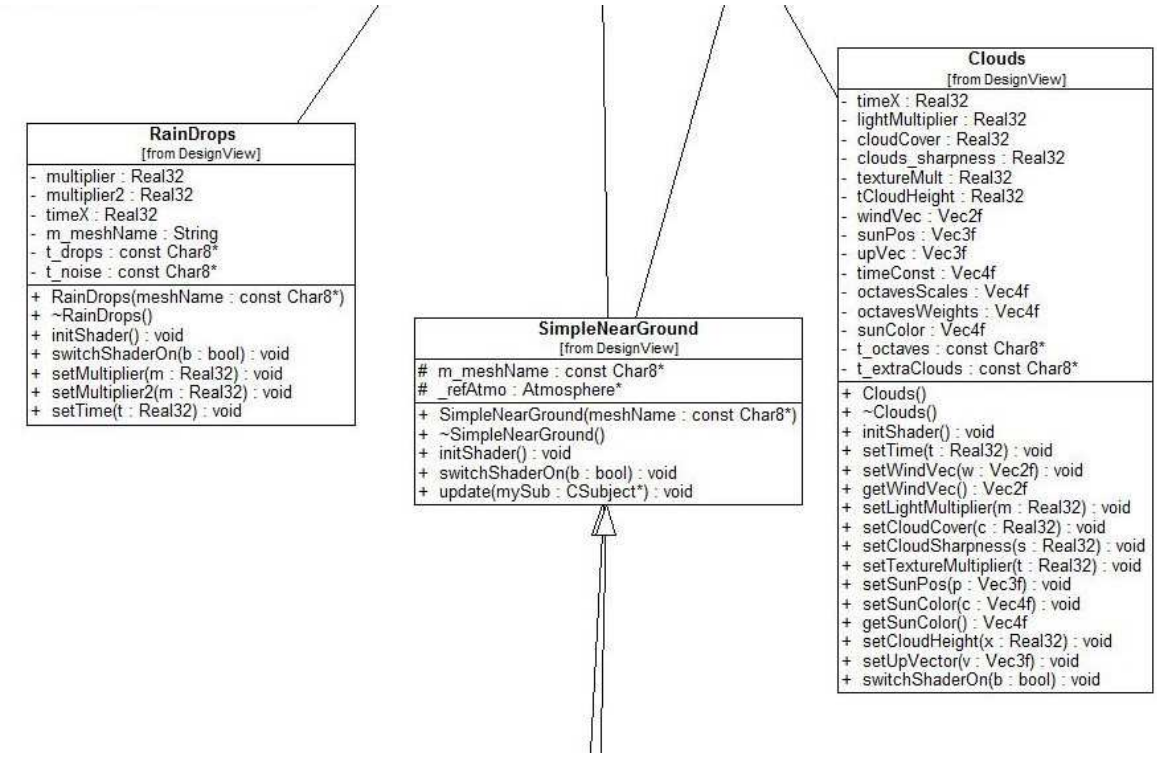


Figure B.2.: UML Class Diagram of the shader classes showing SimpleNearGround class which controls the atmospheric effects for near-ground objects, RainDrops and Clouds class. They are derived from the SimpleShader Class Figure B.1.

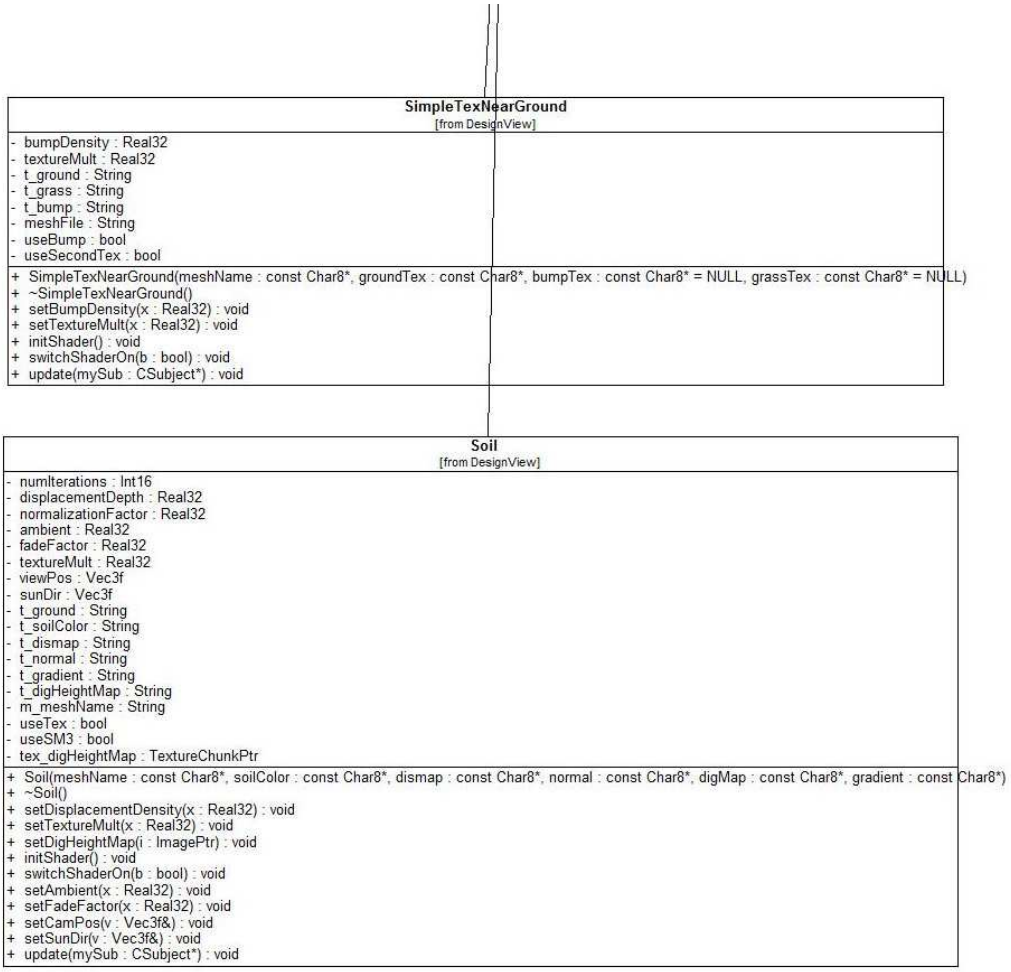


Figure B.3.: UML Class Diagram of the shader classes showing the advanced version of near-ground object classes and the soil class. They are derived from the SimpleShader Class Figure B.1.

# Appendix C.

# GLSL Shaders

## C.1. Soil

Listing C.1: Soil vertex shader in GLSL

```
/**
*   vertex shader for rendering soil
*   the color of the soil is calculated by a color gradient and
*   a heighmap which defines where and from which deep (greyscale)
*   the soil is
*/

/// position of the viewer /camera
uniform vec3 viewPos;
/// deep of the displacement
uniform float displacementDepth;
/// the normalized sun Direction
uniform vec3 sunDir;
/// needed for correct coloring
uniform vec3 attenuationFactor;
uniform vec3 colorFactor;
uniform float atmoConst;
uniform vec3 upVec;

varying float fogC;
//varying vec3 normal;
/// tangent view and sun direction
varying vec3 tanEyeVec;
varying vec3 tanLightVec;

void main( void )
{

   /// Project position into screen space
   /// and pass through texture coordinate
   gl_Position = ftransform();
   gl_TexCoord[0]     = gl_MultiTexCoord0;

   /// part of the coloring with the sky
   /// see atmosphere for more details
   float angle = dot(upVec, gl_Normal);
   angle = (angle+1.0)*0.5;
   vec3 v3Attenuate = exp( attenuationFactor*exp2(angle*atmoConst));
   gl_FrontSecondaryColor.rgb = v3Attenuate;
```

```
    gl_FrontColor.rgb = v3Attenuate * (colorFactor);

    // fog
    vec4 V = gl_ModelViewMatrix * gl_Vertex;
    fogC = abs(V.z);
    //normal = gl_Normal;

    // eye vector .
    // Adjust the slope in tangent space based on bump depth
    vec3 eyeVec = viewPos - gl_Vertex.xyz;

    //bump
    //transform by orthonormal basis
    // OSG packs tangent and binormal into texcoord after
    // calling calcVertexTangents()
    mat3 tangentMat = mat3(gl_MultiTexCoord1.xyz,
                           gl_MultiTexCoord2.xyz,
                           gl_Normal);

    // Transform the eye vector into tangent space.
    // and apply displacement depth
    tanEyeVec = (eyeVec) * tangentMat;
    tanEyeVec.z *= (-1.0/displacementDepth);
    // Transform the sun vector into tangent space.
    tanLightVec = sunDir*tangentMat;

}
```

Listing C.2: Soil fragment shader in GLSL

```
/**
* applys a per pixel displacement map and color with texture
*/
// how deep gets our tracer
#define NUM_ITERATIONS 8

// the texture for the soil
uniform sampler2D soilSampler;
// precalculated 3D texture distance map
uniform sampler3D disMapSampler;
// normal Texture
uniform sampler2D normalSampler;
// gradient which holds color ramp
uniform sampler2D gradientSampler;
// the heighmap holding the information about where and how deep the soil is
uniform sampler2D digHeightMapSampler;
// this is the depth of 3dTexture divided by the width of the 3dTexture
// 8/256 in my case
uniform float normalizationFactor;
// lightens up the soil if it gets too dark with the diffuse component
uniform float ambient;
// texture Multiplier
uniform float textureMult;

//parameters for atmospheric effects
uniform float exposure;
uniform float fogFactor;
uniform vec3 sunDir;

varying float fogC;
//varying vec3 normal;

varying vec3 tanEyeVec;
```

```
varying vec3 tanLightVec;

void main( void )
{
   //the main texture coordinates
   vec3 texCoord = vec3(gl_TexCoord[0].xy*textureMult, 1.0);
   //the heightmap
   vec2 colorDig = texture2D(digHeightMapSampler, gl_TexCoord[0].xy).rr;
   //the normalized offset for the texture
   vec3 offset = normalize(tanEyeVec);
   offset *= normalizationFactor;

   float distance2Trace;
   //the main sphere tracing
   for (int i = 0; i < NUM_ITERATIONS; i++) {
      distance2Trace = texture3D(disMapSampler, texCoord).r;
      texCoord += offset * distance2Trace;
   }

   // Compute derivatives of unperturbed texcoords.
   // This is because the offset texcoords will have discontinuities
   // which lead to incorrect filtering.
   //vec2 dx = dFdx(TexCoord.xy);
   //vec2 dy = dFdy(TexCoord.xy);
   // the transformed texture coordinates for the bump mapping
   vec3 tanNormal = texture2D(normalSampler, texCoord.xy).xyz * 2.0 - 1.0;
   //the transformed light vector
   vec3 tanLightVecN = normalize(tanLightVec);
   // the diffuse component for current fragment
   float diffuse = clamp(dot(tanLightVecN, tanNormal), 0.0, 1.0);

   // the base texture color
   vec4 baseColor = texture2D(soilSampler, texCoord.xy);
   //the color gradient from the depth colordig.greylevel
   vec4 gradientColor = texture2D(gradientSampler, colorDig.rr);

   baseColor *= gradientColor;
   /// atmospheric effects
   float fog = exp2(-fogFactor * fogC);
   vec4 color =  gl_Color + baseColor * gl_SecondaryColor;
   baseColor *=(diffuse+ambient);
   // should get into HDR
   color = 1.0 - exp(-exposure * color);
   gl_FragColor = mix(color, baseColor, fog);

}
```