Universität Potsdam
Naturwissenschaftliche Fakultät
Hasso-Plattner Institut für Software System Technik
Prof. Dr. rer. nat. habil. Jürgen Döllner

Diplomarbeit
zur Erlangung des Grades
Diplom-Informatiker
an der Universität Potsdam

Analysis and Exploration of Virtual 3D-Citymodels using 3D Information Lenses

Eingereicht von:
Matthias Trapp
Heinrich von Kleist Str. 15
14482 Potsdam

Potsdam, den 15. Januar 2007

Analysis and Exploration of Virtual 3D-Citymodels using 3D Information Lenses

Abstract

This thesis addresses real-time rendering techniques for 3D Information-Lenses, which are based on the focus & context metaphor. It analyzes, conceives, implements and reviews its applicability to objects and structures of virtual 3D city models. In contradiction to digital terrain models, the application of focus & context visualization to virtual 3D city models is barely researched. However, the purposeful visualization of contextual data of is extreme importance for the interactive exploration and analysis of this field. Programmable hardware enables the implementation of new lens-techniques, which allows the augmentation of the perceptive and cognitive quality of the visualization, compared to classical perspective projections. A set of 3D information-lenses is integrated into a 3D scene graph system:

- Occlusion Lenses modify the appearance of virtual 3D city-model objects in order to resolve their occlusion and consequently facilitate the navigation.
- Best-View Lenses display city-model objects in a priority-based manner and mediate their meta-information. Thus, they support exploration and navigation of virtual 3D-city-models.
- Color and deformation lenses modify the appearance and geometry of 3D city-models to facilitate their perception.

The present techniques for 3D information lenses and the application to virtual 3D city models clarify their potential for interactive visualization and form a base for further development.

Analyse und Exploration virtueller 3D-Stadtmodelle durch 3D-Informationslinsen

Zusammenfassung

Diese Diplomarbeit behandelt echtzeitfähige Renderingverfahren für 3D-Informationslinsen, die auf der Fokus-&-Kontext Metapher basieren. Im folgenden werden ihre Anwendbarkeit auf Objekte und Strukturen von virtuellen 3D-Stadtmodellen analysiert, konzipiert, implementiert und bewertet. Die Focus-&-Kontext Visualisierung für virtuelle 3D-Stadtmodelle ist im Gegensatz zum Anwendungsbereich der 3D-Geländemodelle kaum untersucht. Hier jedoch ist eine gezielte Visualisierung von kontextbezogenen Daten zu Objekten von großer Bedeutung für die interaktive Exploration und Analyse. Programmierbare Computerhardware erlaubt die Umsetzung neuer Linsen-Techniken, welche die Steigerung der perzeptorischen und kognitiven Qualität der Visualisierung im Vergleich zu klassischen perspektivischen Projektionen zum Ziel hat. Für eine Auswahl von 3D-Informationslinsen wird die Integration in ein 3D-Szenengraph-System durchgeführt:

- Verdeckungslinsen modifizieren die Gestaltung von virtuellen 3D-Stadtmodell- Objekten, um deren Verdeckungen aufzulösen und somit die Navigation zu erleichtern.
- Best-View Linsen zeigen Stadtmodell-Objekte in einer prioritätsdefinierten Weise und vermitteln Meta-Informationen virtueller 3D-Stadtmodelle. Sie unterstützen dadurch deren Exploration und Navigation.
- Farb- und Deformationslinsen modifizieren die Gestaltung und die Geometrie von 3D-Stadtmodell-Bereichen, um deren Wahrnehmung zu steigern.

Die in dieser Arbeit präsentierten Techniken für 3D Informationslinsen und die Anwendung auf virtuelle 3D Stadt-Modelle verdeutlichen deren Potenzial in der interaktiven Visualisierung und bilden eine Basis für Weiterentwicklungen.

Acknowledgments

Herewith, I thank the work group of Prof. Dr. Juergen Doellner, especially himself, M.Sc. Haik Lorenz, Dr. Marc Nienhaus and Oleg Dedkow for their support, expertise and their frankly attitude.
I am very grateful to my family: my mother, my father and my wonderful sister, who supported me all over the time and gave me the great opportunity to study according to my wishes. They always stood beside me with good advice. I like to thank also all my friends who helped me through the hard time and filled my life and heart.
Especially, I like to thank my long standing girlfriend Sabine Pommerening, knowing that no phrases could describe what she means to me. She always believes in me and keeps me grounded. Thank you very much.

Matthias Trapp

Trademarks, Patents and Copyrights

Contents

List of Figures

List of Tables

List of Listings

List of Abbreviations

A

API ..... Application Programming Interface
ARB ..... Architectural Review Board
AABB ..... Axis-Aligned Bounding Box

B

BVL ..... Best View Lens

C

CA ..... Context Anchor
CL ..... Context-Line
COP ..... Center of Projection
CCS ..... Camera Coordinate System
CPU ..... Central Processing Unit
CSG ..... Constructive Solid Geometry

D

DBVL ..... Dynamic Best View Lens
DOF ..... Depth of Field
DOI ..... Degree of Interest

F

FA ..... Focus Anchor
FBO ..... Frame Buffer Object
FCP ..... Far Clipping Plane
FCV ..... Focus and Context Visualization
FHHT ..... Fragment Handler Hook Table
FHO ..... Fragment Handler Object
FHP ..... Fragment Handler Prototype
FOV ..... Field of View
FSH ..... Fragment Shader Handler

G

GLSL ..... OpenGL Shading Language
GPU ..... Graphic Processing Unit

L

LBV ..... Lens Based Visualization
LDX ..... LandXplorer
LF ..... Look-From
LOD ..... Level-Of-Detail
LT ..... Look-To
LU ..... Look-Up

M

MCOP ..... Multiple Center of Projection
MPR ..... Multi Pass Rendering
MRT ..... Multiple Render Targets
MS ..... Multi Sampling
MVL ..... Map View Lens

N

NCP ..... Near Clipping Plane
NDC ..... Normalized Device Coordinates
NPOTS ..... Non-Power-of-Two-Sized
NPR ..... Non Photorealistic Rendering

O

OO ..... Object Orientation
OpenGL ..... Open Graphics Library
OpenSG ..... Open SceneGraph

P

PDT ..... Pliable Display Technology
POI ..... Point of Interest
POTS ..... Power-of-Two-Sized
PRS ..... Post Render Style

R

RP ..... Refinement Pattern
ROI ..... Region Of Interest
RPM ..... Refinement Pattern Map
RSC ..... Render Style Configuration
RTT ..... Render To Texture

S

SBVL ..... Static Best View Lens
SCOP ..... Single Centre of Projection
SDOF ..... Semantic Depth of Field
SG ..... Scene Graph
SH ..... Shader Handler
SMS ..... Shader Management System
SRS ..... Scene Render Style

U

US ..... Uber-Shader

V

Chapter 1

Introduction

They are ill discoverers that think there is no land,
when they can see nothing but sea.
-Sir Francis Bacon

1.1 Motivation

Today all privileged regions of the world that have access to modern information technology suffer from a fundamental problem. The quantum of encoded information grows at a tremendous rate while the ability of unproblematic access to this data decreases at the same time. Also geospatial information represented by geo-data, such as virtual 3D city model data,is affected by this phenomenon. A 3D city model usually is a three-dimensional representation of an existing city or an urban environment.
Due to the rapid development of computer hardware and the progress in (semi-) automatic data acquisition it is now possible to create large-scale 3D city models at reasonable costs. This development has led to a numerous applications, e.g. in urban planning, telecommunications and ecology as well as in tourism, and entertainment.
In times of services like *Google Earth*, *WorldWind*, and *geotainment products* like *Munich 3D*, *Berlin 3D*, and *Virtual Helsinki* fast and coherent access to geospatial information becomes more and more important, i.e. to find a specific information without the transgression of a critical amount of time. This implies solving a search problem: from an uncertain key information to a certain particular one. The optimal case would be if knowledge is available about what specific information is needed and where it can be found. If no such kind of a mapping exists, we are forced to explore and navigate through the data space. In case of virtual 3D city models users face a three dimensional space. Unfortunately, users tend to get lost in many 3D systems requiring them to navigate [113].
Information visualization addresses the problem of how to effectively present information visually. Visualization techniques include selective hiding of data, layering data, and taking advantage of psychological principles of layout, such as proximity, alignment, and shared visual properties (e.g. color).
Focus and context visualization (FCV) is a principle of information visualization. It displays the most important data at the focal point at full size and detail, and display the area around the focal point (the context) to help make sense of how the important information relates to the entire data structure. Regions far from the focal point may be displayed smaller (as in fisheye views) or selectively omitted. Displaying information in

a context that makes it easier for users to understand is the central task in information visualization. Information visualization is an attempt to display structural relationships and context that would be more difficult to detect by individual retrieval requests [91]. Today we are in demand of visualizations that support fast decisions. Providing *overview and detail* is only one possible solution. In case of *virtual 3D city models*, the application of FCV is generally not explored, but this technology possesses a wide range of target applications (inter alia):

- Displaying roads or other surface networks. The user should be able to obtain a detailed view onto these objects without occlusions or navigation overhead.
- Highlighting or depicting points/object of interest or important route finding information such as cross roads etc. which are far away from the viewers location.
- Enabling visualization of spatial significance for market reports or similar purposes.
- Depicting user- or referenced data like floor occupancy (living space vs. office space) or other annotations [39, 41].
- Facilitating selective LOD representations. The geometry in the focus area with can be rendered with a higher level of detail or with a lower LOD (simplifier lens). A possible application could be the exploration of a city model based on *Smart Buildings* [43].

A computergraphical lens is a metaphor of FCV
Clipping is to spatially partition geometric primitives, according to their containment within some region. Clipping can be used to: Distinguish whether geometric primitives are inside or outside of a viewing frustum or picking frustum Detecting intersections between primitives

1.2 Problem Statement

Nowadays it is possible to render a large amount of spatial data in real-time under the assumption of having optimized LOD data structures and hardware accelerated rendering methods. The principle of detail and context visualization conflicts with some of these methods. The nature of virtual 3D city models requires some important restrictions:

1. Spatial relations within a model are fixed. Methods for reflecting data relationships as spatial relationships such as in tree visualization are not applicable.
2. It cannot be assumed that hierarchical information like building adjacencies or other statistical criteria are available a priori.
3. One has to act on the assumption that a large amount of geometrical data has to be processed.

The aim is to develop scene graph tools and techniques for 3D focus and context visualization/ navigation, 3D object highlighting as well as 3D focus and context separation methods. These methods should work without any semantic information about the objects in the scene. It is known that at least two main restrictions affect the visualization of

3D city models: the limitation of screen space and processing power. The first restriction directly addresses the problem of *screen real estate*: the amount of space available on a display for an application to provide output. Typically, the effective usage of screen real estate is one of the most difficult design challenges because of the desire to have as much data and as many controls visible on the screen as possible to minimize the need for hidden commands or scrolling. At the same time, excessive information may be poorly organized or confusing. Because of that, effective screen layouts must be developed with appropriate use of free space.
Usually, FCV techniques are able to maximize the use of screen real estate and can present a large amount of data within a small space. They allow the examination of a local area in detail within context of the whole data set. But how to integrate additional information in a rendering of a 3D city model scene that uses standard perspective projection? The following list should give an overview over some possibilities for improving the analysis and exploration of virtual 3D city models:

- Resolving building occlusion using geometrical distortions or visual abstractions of shape and texture. Visual abstractions are able to shift the cognitive load to the application. Abstract information increases the ability of the users to assimilate and retrieve information. This is useful for verifying or falsifying a hypothesis by analyzing the 3D information space.

- Giving overview and insight for areas which are located far away from the viewer by using multiple simultaneous views or separated global views. This supports the exploration of a city model as well as the investigation of the 3D information space without a hypothesis.

- Easing orientation, navigation, and preattentive perception by applying different render techniques or visual abstractions like non-photo-realistic rendering.

- Adding thematic information to the scene. Application-defined data attached to buildings is essential for all applications operating on 3D city models.

1.3 Fundamentals & Notations

It follows a list of short descriptions of fundamental terms used in this thesis.

Virtual 3D City-Model A virtual 3D city-model represents a specialized geovirtual environment and consists of an underlying 3D terrain model, 3D buildings, and 3D vegetation. Additionally, street and green spaces can be defined. 3D city models provide basic functionality for exploration, analysis, presentation and editing of spatial information.

3D Information Lens A 3D information lens unifies the aspects of focus and context visualization/navigation as well as thematic/semantic lenses in the domain of virtual 3D city models.

Focus In this thesis the term focus can be understood as a location of immediate interest. This location is usually placed in the coordinate system of the virtual 3D city model. Its dimension is described by the focus area.

Context The specific circumstances (e.g. spatial data) of the focus will denoted as context. It also describes the coherence of situation and topic that relates to the focus. Similar to the focus the dimension of the complementary context will be described by the context area.

Focus Area The focus area describes the location and spatial dimensions of a region of interest. Thus, it describes all geometry which is inside this volume or attached to it. A 3D information lens could possess of more than one focus area.

Context Area The context area describes the complementary space of all foci areas. In difference to multiple focus areas there is just one context area. All geometry outside any focus area is part of the context area.

Focus Rendering The result of the rendering of the geometry inside the focus area is denoted as focus rendering, which can be the product of a complex render technique or can be empty.

Context Rendering Analog to focus rendering the context rendering denotes the rendering of the geometry of the context area.

Due to variations in notation for vector and matrix calculations, a short overview of the notation used throughout this thesis follows. Vectors are denoted with capital, components with small letters, e.g. $A = (x, y, z)$. The length of a vector is written as $|A|$, the normalized form is expressed by $\|A\|$, a $\cdot$ represents scalar multiplication and / scalar division. The specific vector $O = (0, 0, 0) \in \mathbb{R}^3$ represents the origin of a coordinate system. Finally $A \bullet B$ describes the dot-product and $A \times B$ the cross-product of the vectors A and B. Matrices are denoted with non-italic uppercase bold letters: $\mathbf{M}$. Functions are designated with Greek letters. Table 1.3.1 shows an overview of coordinate systems in use.

Abbreviation	Explanation
$\mathcal{WCS} \subseteq \mathbb{R}^3$	World Space Coordinate System
$\mathcal{CCS} \subseteq \mathbb{R}^3$	Camera or Eye Space Coordinate System
$\mathcal{SCS} = [0, w] \times [0, h] \subset \mathbb{N}^2$	Screen Space Coordinate System
$\mathcal{NDC} = [-1, 1]^2 \subset \mathbb{R}^2$	Normalized Device Coordinates

Table 1.3.1: *Coordinate systems used in this thesis.*

1.4 Structure & Typographic Conventions

Structure: The remainder of this thesis is structured as follows:
Chapter 2 briefly reviews related work in the fields of focus an context visualization, introduces necessary hardware-accelerated rendering techniques and concepts that utilize the programmable rendering pipeline.
Chapter 3 outlines the concepts and principles of 3D information lenses and possible applications for each lens type. It introduces volumetric depth sprites as well as the volumetric depth test. The chapters covers occlusion, best-view, color, and deformation lenses. The concept of generic uber-shaders will be developed and specified.
Chapter 4 shortly describes important implementation issues of the concepts mentioned above. It covers basic design decisions and software architectural aspects of this thesis.
Chapter 5 analyzes and discusses the performance and limitations of the presented approaches. The chapter concludes with potential future research directions. It covers potential technical improvements as well as an outline of continuative features of 3D information lenses.
Chapter 6 gives conclusions by summarizing and reviewing the presented approaches related to their applications.

Typographic Conventions: This thesis includes different typesettings. Words that relate to an implementation key word will be set in `typewriter`. Proper nouns will be set *emphazised*. All class diagrams describing issues of software architecture are composed in UML 2.0 [32].

Chapter 2

Related Work

Software *lens-based visualization* (LBV) providing capabilities for in-place presentation of details in a global context. Interactive LBV can be applied to explore continuous geospatial representations, as well as non-geospatial visualizations such as network diagrams [14]. This section focuses on 3D lens rendering approaches and as well as focus & context visualization approaches that could be utilized for 3D information lenses.

2.1 3D Lens-Based Visualization Techniques

We can distinguish between two types of 3D-lenses: flat 3D lenses and lenses with a volumetric shape. This section presents a brief introduction to the basic concepts of 3D lenses and does not address any applications to volume rendering.
The *magic lens* metaphor and *Toolglasses*™ have been introduced by Bier et al. [19]. They describe widgets as interface tools that can appear between an application and a traditional cursor. Visual filters bind to the widgets, known as magic lenses, can modify the visual appearance of application objects, enhance data of interest or suppress information in the region of interest, which is determined by the shape of the lens. A sophisticating overview of 3D magic lenses and magic lights is given in [97]. This work applicates this metaphor to immersive building services.

Analytical Approaches An application to 3D environments and *volumetric lenses* was first published by Cignoni et al. [82]. They introduced the *MagicSphere* metaphor as an insight tool for 3D data visualization which is restricted to a spherical shape. The analytical approach classifies the geometry upon its relation to the lens shape (inside, outside, on the border). The rendering is done within two passes, each for every classification. In each pass different visual appearances can be applied. The border geometry is rendered in both of them. The MagicSphere metaphor generates visual artifacts near its border.
A different analytical approach of a similar concept has been used by Idelix Software Inc. [40, 14]. The pliable display technology 3D (PDT3D) avoids object occlusions in 3D virtual environments by analyzing camera and lens parameters and applying corresponding geometric transformations to occluding objects. Thus, it is possible to select a region of interest to which the system provides an occlusion-free view. The major disadvantage of this concept is the modification of the scene structure lying outside the region of interest through geometrical transformations, which leads to a loss of contextual information.

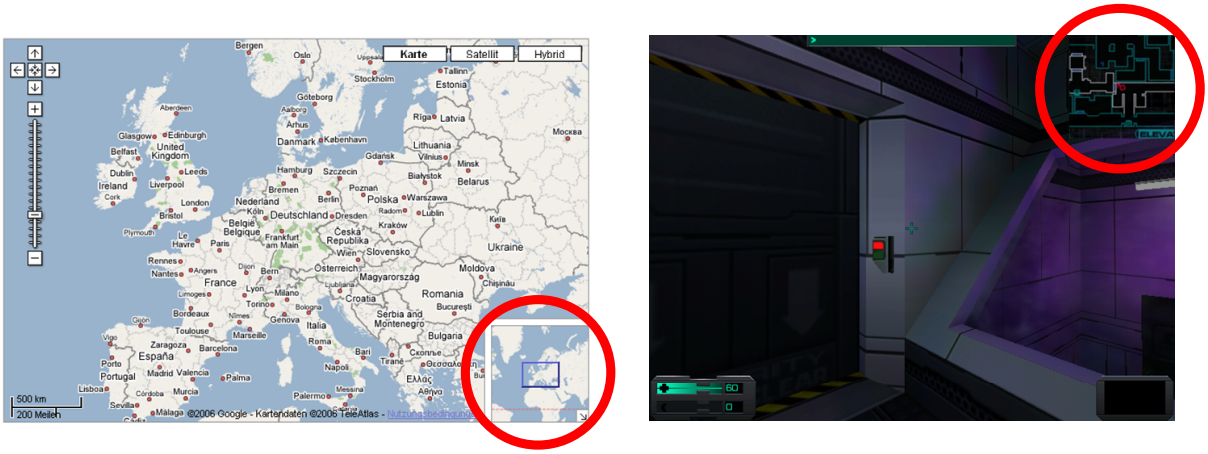


Figure 2.2.1: *Examples detail and overview context maps. Left: Google Maps©, Right: System Shock 2© game.*

Image-Based Approaches A more general extension of the magic lens metaphor to 3D virtual environments has been presented by Viega et al. [51]. They introduced an algorithm for the visualization of volumetric lenses as well as flat lenses in a 3D environment. The implementation is done by using infinite clipping planes for the volume faces. This approach is computationally expensive for complex lens shapes.
Ropinski [109] presented an algorithm for real-time rendering of volumetric magic lenses that have an arbitrary convex shape, which is fully hardware-accelerated. It supports the combination of different visualization appearances in one scene. The approach uses multipass rendering and shadow-mapping to separate focus from context data.

2.2 Focus & Context Visualization

Focus & Context Visualization (FCV) in virtual 3D environments has been well researched during the past years [26, 102, 13, 78, 38, 37]. There is a multitude of approaches for virtual 3D terrain lenses: view dependent non-linear visualization techniques such as *pliable surface technology* (PDT) [40, 68, 66, 67, 113, 70, 83, 79, 56]. These approaches distort the underlying mesh vertices so that the impression of magnification occurs. One can find also texture based approaches like *cartographic lenses* [30] and *thematic texture lenses* [111, 45, 36, 46]. Many researchers have addressed the screen real-estate problem. One solution, the so-called *detail-in-context* technique, integrates detail with contextual information. Figure 2.2.1 shows two examples. This section is restricted to techniques which are applicable to 3D virtual environments.

Non-Distortion Techniques The *Through-The-Lens* metaphor [99] presents a set of tools that enable simultaneous exploration of a virtual world from two different viewpoints. One is used to display the surrounding environment and represents the user, the other is interactively adjusted to a POI. The resulting image is displayed in a dedicated window. Textual and 2D image landmark representations lack the depth and context needed for humans to reliably recognize 3D landmarks. *Worldlets* [18] describe a 3D thumbnail landmark affordance. It represents 3D fragment of a virtual world and enables first-

person, multi-viewpoint representations of potential destinations.
Semantic Depth of Field Rendering (SDOF) utilizes a well-known method from photography and cinematography (depth-of-field effect) for information visualization, which is to blur different parts of the depicted scene in dependence of their relevance. Independent of their spatial locations, objects of interest are depicted sharply in SDOF, whereas the context of the visualization is blurred [90, 89]. Evaluations of this technique prove that the SDOF concept is pre-attentive and that it supports directly the perception of sharp target items when the context is blurred. SDOF can significantly support users in focusing on relevant data and guide their attention [112].

Occlusion Techniques The depiction of occluded structures is a common problem in computer graphics. This difficulty is known under the terms *virtual X-ray vision*, *Cut-Away*, *Break-Away*, as well as *ghost views*. The goal of this set of techniques is to show objects that are present in the scene but occluded from view. A taxonomy of occlusion techniques and a comprehensive problem analysis is provided by Elmqvist et al.[80].
X-Ray Vision is mainly researched in the field of augmented reality. A set of interactive virtual X-Ray vision tools for depicting occluded infrastructure is presented in [94]. The tools directly augment the users' view of the environment, allowing them to explore the scene in direct, first person view. Different depiction styles for enhancing the depth relationships of objects are researched in [71]. In the field of virtual reality, a correct 3D perspective cut-away lens technique is introduced in [9]. The user can define a cutout shape and sweep it over the occluding geometry of an arbitrary 3D graphics scenes. This approach uses CSG methods to cut into the obstructing geometry. Occlusion lenses can also be found in volume rendering. By navigating through a dense volume dataset the view of the camera will always be occluded. To avoid this problem Ropinski et.al [105, 108] propose an occlusion lens which renders those parts of the volume dataset transparently that occludes the region of interest.

2.3 Programmable Graphic Hardware

The latest improvements of the rendering pipeline and the turning away of its equivalence in hardware increase the degree of general processing with graphic accelerators. Figure 2.3.1 shows a comparison of the standard rendering pipeline (A) and the modern DX10 [95] influenced rendering pipeline (B). Besides a new memory model that allows the ubiquity resource access in every programmable stage of the pipeline, *geometry shaders* and *stream output* [72] are the main alterations. Geometry shaders allow geometry amplification due to emission of new primitives of a specified output type. The stream output allows data to be directly passed through either the vertex or geometry shader which then in turn passes the information straight to the frame buffer memory. This facilitates the intra/inter-frame re-use of geometry.

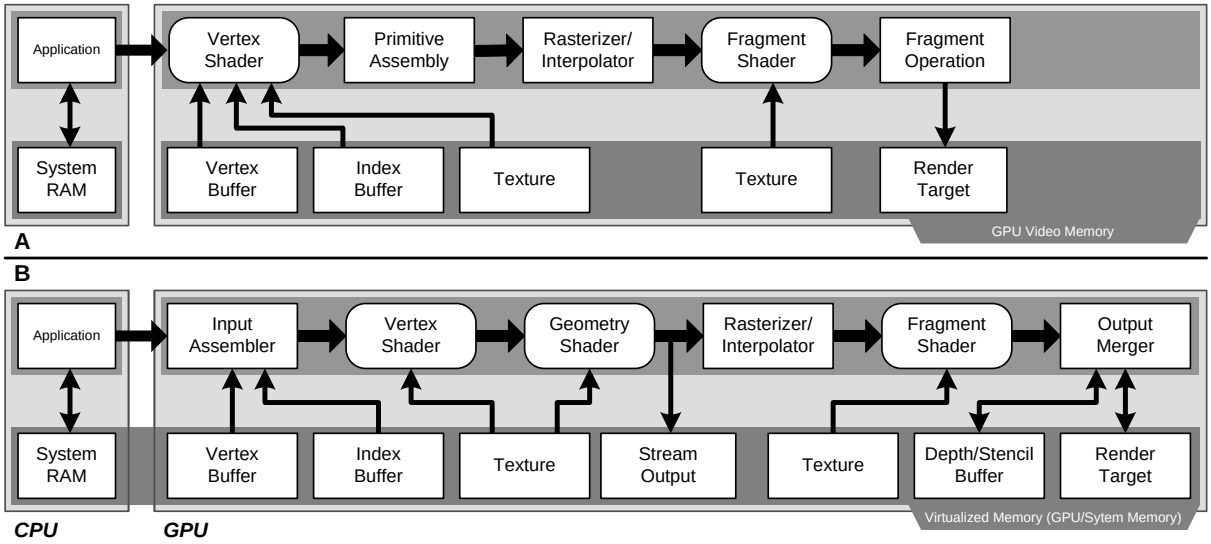


Figure 2.3.1: *Comparison of rendering pipelines. A: standard rendering pipeline, B: DX10 rendering pipeline*

OpenGL Shading Language The *OpenGL Shading Language* (GLSL or GLslang) is a C-like high-level shading language specifically designed for the OpenGL Architecture by the OpenGL ARB. It can be used to gain direct control of specific features of the graphics pipeline. Shader programs consist of shaders that implement leastwise either one vertex and/or one fragment shader. Shader programs are then made part of the current rendering state of the rendering context of OpenGL [69]. Consequently, only one program can be active at one point of the time. The listing 2.3 shows a common example for a vertex shader.

```
varying vec3 normal;

void main(void)
{
  gl_Position = ftransform();
  normal = normalize(gl_NormalMatrix * gl_Normal);
  gl_TexCoord[0] = gl_MultiTexCoord0;

  return;
}
```

Listing 2.3: *Vertexshader Example*

The shader calculates the vertex position `gl_Position` for the rasterizer/interpolator. Therefore it uses the *built-in function* `ftransform` which represents the fixed-function vertex transformation. It also computes the vertex normal in eye-space coordinates using a *derived matrix state*. Finally, the shader transfers the first multi-texture coordinate by using a *built-in varying variable*. A varying variable represents an interface between vertex and fragment shader. Listing 2.3 shows the corresponding fragment shader.

```
uniform sampler2D sampler0;
varying vec3 normal;

void main(void)
{
  // 2 render targets
  gl_FragData[0] = texture2D(sampler0, gl_TexCoord[0].st);
  gl_FragData[1] = vec4(normal, 1.0);

  return;
}
```

Listing 2.3: *Fragmentshader Example*

The shader demonstrates GLSL ability to render into multiple targets in a single pass (see section 2.3). It samples from a 2D texture using the *built-in texture function* `texture2D` with the *texture handle* `sampler0` and the texture coordinate interpolated by the rasterizer/interpolator for the current fragment as arguments. The *output variable array* `gl_FragData[]` enables the fragment shader to address multiple render targets.

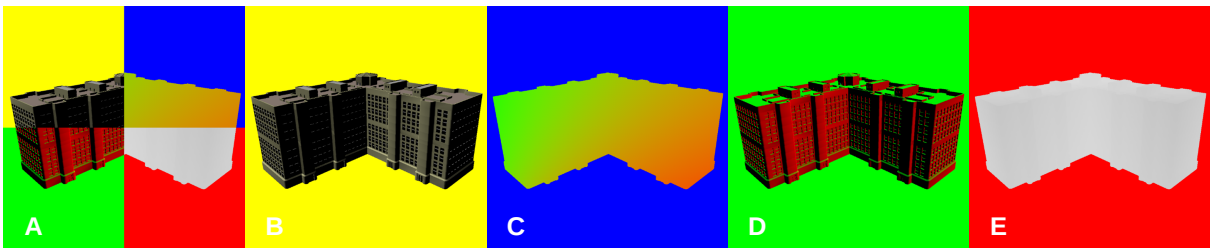


Figure 2.3.2: *Example of using multiple rendertargets for color (B), normalized world-view-coordinates (C), normal (D) and depth (E).*

Render-To-Texture *Render-To-Texture* (RTT) is a efficient method to use pixel data that have been rendered to a texture. The RTT method allows to write pixel data directly into a buffer that could be a texture. The alternative *copy-to-texture*(CTT) method performs worst and is out of date. Depending on the API, render-to-texture can be implemented in various ways. Since this thesis is based on the OpenGL API, it uses framebuffer object (FBO) extension in combination with high-precision 16/32bit floating point textures [72].
Ping-Pong Rendering [15] is a technique that is used with RTT to avoid reading and writing the same buffer simultaneously, instead of swapping between a pair of buffers. Such a technique is often required in general purpose computations

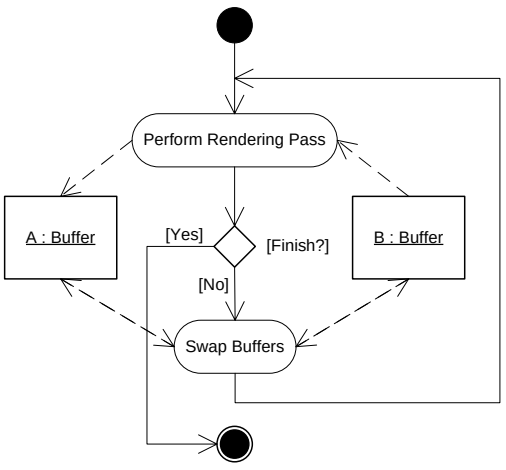


Figure 2.3.3: *Activity diagram of the ping-pong rendering technique.*

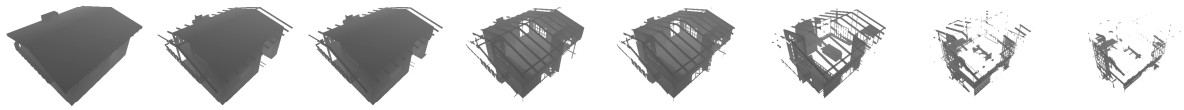


Figure 2.3.5: *Example of depth peeling algorithm for $n = 8$. The depth values increase from left to right.*

on the GPU: iterative algorithms write data in one pass and then read back this data to generate the results of the next pass. Figure 2.3.3 depicts this process. Alternating, the buffers A and B are bound for reading and writing respectively.
The *Multiple Render Target* (MRT) [72] technology allows the fragment shader to save per-pixel data in multiple buffers within one rendering pass. Typical information stored in these kinds of buffers include position, normal, color, and material. This allows advanced post-processing techniques like *deferred shading* or other effects. Figure 2.3.2 shows an example of this technique. The background colors were chosen to punctuate the differences.

Depth Peeling Depth peeling is the underlying multipass fragment-level technique that allows order independent transparency, i.e. it eliminates the need for depth sort or traditional preprocessing on CPU and is suitable for per-pixel lighting. It is an image space algorithm on GPU that emulates dual depth buffer tests. Standard depth testing gives us the nearest fragment without imposing any ordering restrictions. However, it does not give us any straightforward way to render the n^{th} nearest surface. Depth peeling solves this problem. This technique uses n passes over a scene to obtain n layers of unique depth [8] and the particular color maps (see figure 2.3.4). These maps will be alpha-blended in back to front order. Depth Peeling can be used in combination with edge enhancement or blueprint rendering [69]. For more precision, it could be used in combination with *linearised depth buffers* [22, 21] (see figure 2.3.5). The necessity of several rendering passes represents a serious drawback of this approach. To achieve high visual quality as well as an acceptable performance it is required to know the sufficient number of passes n. It is possible to approximate depth-peeling for efficient transparency by bounding the number of rendering passes using a blending heuristic [27].

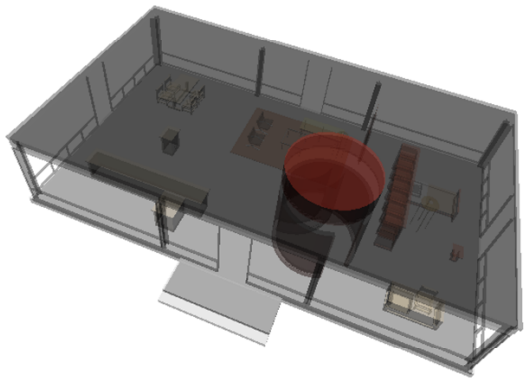


Figure 2.3.4: *Example of the depth peeling technique with $n = 6$ layers.*

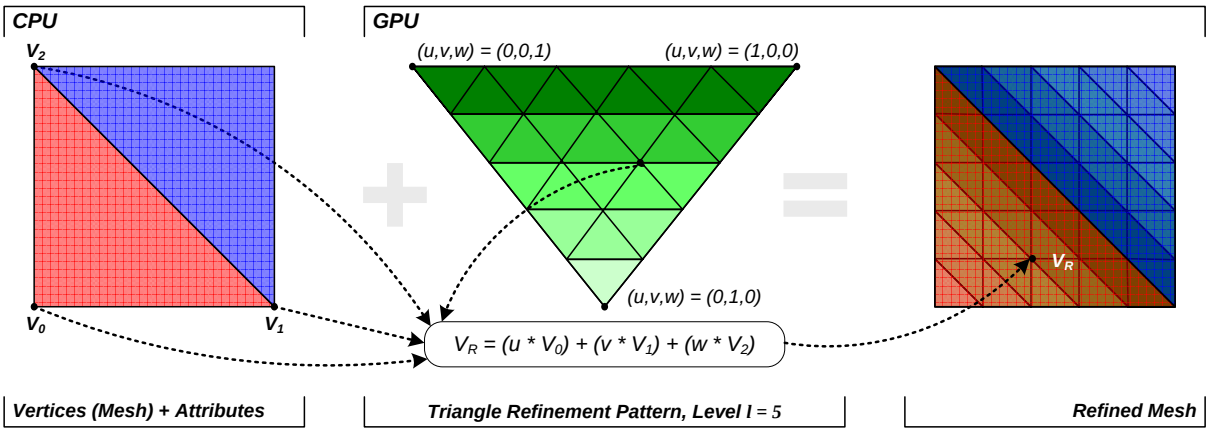


Figure 2.3.6: *Concept of generic mesh refinement on GPU.*

Generic Mesh Refinement It can be found different methods to improve the visual quality by keeping a low geometry complexity. Texture-, bump- and displacement-mapping are only some examples. To distort geometry, we require methods which allow us to keep visual quality high and ensure an amount of vertex information for distortion when needed. One ubiquitous technique to generate complex geometric models is to start from a coarse model and apply refinement techniques to get the enriched model. The refinement techniques that have been proposed can be divided in two main families: *displacement mapping* that is usually employed to add some geometric details to a coarse model, and *subdivision surfaces* that are used to generate smooth surfaces from a small number of polygons. To enable mesh distortion in combination with *out-of-core rendering*, the *generic mesh refinement* approach by [106] is used. It is flexible, easy to implement, and can be applied on a large variety of refinement techniques. The main idea is to define a generic refinement pattern (RP) that will be used to virtually create additional inner vertices for a given polygon. These vertices are then transformed by using linear interpolation (see figure 2.3.6 for details).

Chapter 3

Concept of 3D Information Lenses

Art and science have their meeting point in method.
-Edward Robert Bulwer-Lytton

Information lenses for virtual 3D city models unify the aspects of focus and context visualization and navigation as well as thematic or semantic lenses in this area of application. This thesis tries to sketch a framework for this purpose which is applicable for real-time rendering. Hereby focus and context data is strictly separated, i.e. there is no *transition area* between focus and context. Consequently, the techniques presented in this thesis cannot deal with a continuous *degree of interest* (DOI). To achieve lens functionality in real-time a 3D lens has to perform the following main tasks:

- **Separate** the geometry in the focus area (focus geometry) from the geometry of the context (context geometry). The focus geometry can possess semantic/thematic information/properties such as demographic data or other application dependent meta data.
- **Render** the focus, context, and lens geometry in a proper way by using different visualization techniques and their combinations [33, 44, 64].
- **Integrate** the above renderings by a composition using different integration modi.

Generic Properties A 3D lens possesses a set of common attributes [25] like *name*, a numerical *ID*, and a *color* which grant the possibility to distinguish it on different levels of an application. Each lens has a *position* in world space coordinates $P \in \mathcal{WSC}$. Usually the position coincides with a *point of interest* (POI) that is represented by the lens. For interaction purposes a lens could adopt to one of four *interaction states*: *Normal, Roll-Over, Selected, and Disabled*. This thesis tries to take into account that for each type of lens a multiple number of instances should be available. The maximum number of lenses of each type depends on its implementation.

Specific Lenses The generic lens is extended by four kind of lenses introduced by this thesis:

- **Occlusion lenses** resolve intra-object and inter-object occlusions. This lens type introduces non-uniform transparency distribution and the x-ray shading technique for city structures.
- **Best-view lenses** allow the image-based static and dynamic annotation of city structures by using detail and overview techniques in combination with context-lines.
- **Color lenses** support pixel-precise encoding of spatial information by exploiting the difference of rendering styles.
- **Deformation lens** is an experimental technique to facilitate the accentuation of buildings by global deformations.

3.1 Volumetric Depth-Sprites

For the focus and context separation methods in the next section it is necessary to find a flexible representation of the 3D lens shape. To allow accurate, scalable, and fast focus and context separation/integration methods it is useful to represent the lens shapes as high-precision textures. *Vertex texture fetch* (VTF) [72] offers the possibility to access raster data in the transformation and lighting (T&L) stage of the rendering pipeline. Thus, one can determine for a given vertex of the scene geometry and a *volumetric depth sprite* (VDS) of the lens shape, if it is inside the focus or not. For efficient encoding of the shape information into a raster representation and to overcome the limitation of usual depth-sprites I introduce the concept of a *volumetric depth-sprites* (VDS) data structure. This concept is essential for the most algorithms presented in this work, especially for image-based focus and context separation and integration methods.

3.1.1 Definition

Conceptually, volumetric depth-sprites are bilateral depth sprites *without* color information. It is acceptable to disregard the color information because the main aspect of this approach lies in the representation of the shapes volume information. Depth-Sprites or Z-Sprites have their origin in image based rendering [24]. A depth sprite is a billboarded quad with a grayscale texture for offsetting the depth buffer so that flat sprites appear to have shape and volume. Because of that, the sprites can intersect each other or other geometry.

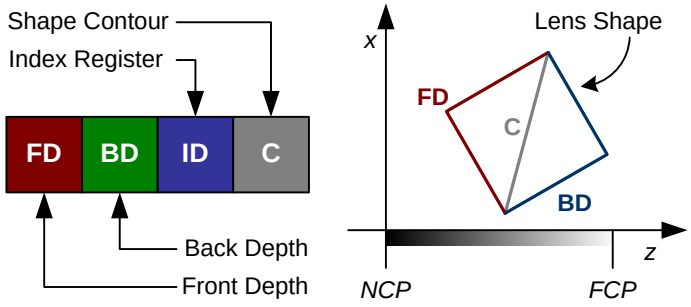


Figure 3.1.1: *Details of the volumetric sprite concept. A: value encoding, B: coherence in an orthographic projection.*

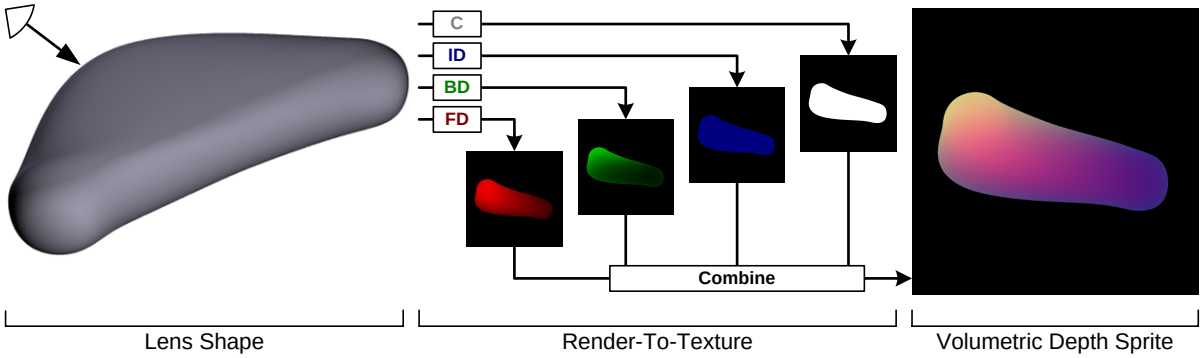


Figure 3.1.2: *The creation concept and the constituents of a volumetric depth sprite for a complex shape.*

Usually, the reference depth is the front-depth of an object. A volumetric depth sprite stores both, the front and the back depth of the lens object. The concept of VDS is only applicable to convex geometry. The values are stored by reinterpreting the usual RGBA[1] layers of an image. Figure 3.1.1 shows the used mappings and the particular coherence for a cubic lens shape. The quality of a VDS depends on the resolution and format of the texture (see section 3.1.3 for details).

3.1.2 Creation Process

A VDS can be created with minor effort during preprocessing of a scene or on demand. The difference lies within the particular projection setup. The creation process can be done with a two-stage *render-to-texture* (RTT) technique in combination with a shader program (compare to figure **??**):

1. Setup the standard depth test (*less*), clear the depth buffer with value 1, set every color channel in the render target to zero, and render front depth of the input geometry (lens shape) to a texture. In this pass the encoding of contour and lens ID is also done.

2. Change the depth test to *greater*, clear depth buffer and every color channel in the render target to zero, render the back depth of the geometry, and integrate the results with the results of the previous pass.

For preprocessing issues, additionally the *near* (NCP) and far (FCP) parameter of the projection are required to scale the depth values while integrating the VDS into a scene with different near and far clipping planes. The sprite front- or back-depth value d_S between the creation clipping setting $near_S, far_S$ and the integration setting $near_I, far_I$, assuming $near_I \leq near_S < far_S \leq far_I$, can be achieved by interval scaling:

$$d_I = \frac{(near_S \cdot (1 - d_S) + far_S \cdot d_S) - near_I}{far_I} \tag{3.1}$$

The encoding of the object identity $id \in I = 0, \ldots, n$, where n denotes the maximum number identities, can be calculated as follows:

$$\gamma : I \longrightarrow C, \qquad id \longmapsto 2^{id}/2^{n+1}, \qquad C = \{2^x/2^{n+1} | \forall x \in I\} \tag{3.2}$$

[1]The additive color system mixes a color by using red, green and blue components.

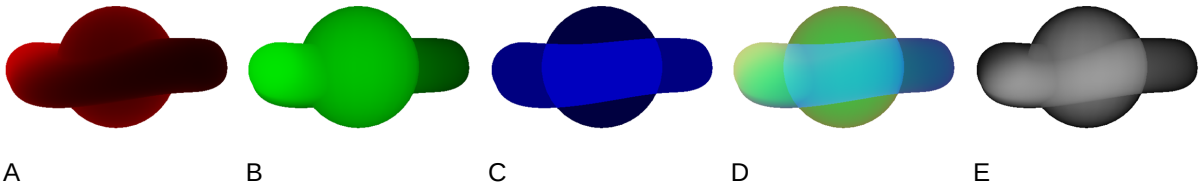


Figure 3.1.3: *Aggregation of two volumetric depth sprites. A: Front depth, B: Back depth, C: Object identity, D: All layers, E: Visualization of the VDS volume.*

With $n+1$ bits being the maximum resolution of a color channel. This is necessary due to the lack of bit-wise operations in the shading language [50]. It is possible to combine two volumetric depth sprites (figure 3.1.3). The result is a union of both depth sprites. Object identities as well as the contours will be added together. An inverse mapping of γ exists is described in section 4).

3.1.3 Depth-Buffer Precision Issues

By speaking of representation of depth values, a sufficient buffer precision is essential. There are four different depth-buffer types available at the moment (compare to figure 3.1.4):

- **Z-Buffer and Complementary Z-Buffer**: Classical Z-buffering is non-linear and allocates more bits for surfaces that are close to the eye-point and less bits farther away [17] (figure 3.1.4.A and B). The normalized mapping functions λ for Z-buffering are defined as follows:

$$\lambda_Z(d) = \frac{f}{f-n} \cdot \left(1 - \frac{n}{d}\right) \qquad \lambda_{\overline{Z}}(d) = 1 - \lambda_Z(d) = \frac{n}{f-n} \cdot \left(\frac{f}{d} - 1\right) \tag{3.3}$$

  Whereas $d \in [n, f] \subseteq \mathbb{R}$ is the z component of a vertex $V \in \mathcal{CCS}$. Furthermore, n denotes the distance from the eye-point to the *near clipping plane* and f the distance to the *far clipping plane*. The greater the ratio $r = f/n$, the less effective the Z-buffer is at distinguishing between surfaces that are close to each other. This quantization of the depth buffer results in stair-artifacts in the distance.

- **W-Buffer and Complementary 1/W-Buffer**: A W-buffer and its complementary 1/W-buffer (see figure 3.1.4.C and D) is a perspective-correct quasi-linear depth buffer. It is more accurate and generally produces better results in the mid-range. The normalized mapping functions λ for W-buffer are defined as:

$$\lambda_W(d) = \frac{d}{f} \qquad \lambda_{\overline{W}}(d) = \frac{n}{d} \tag{3.4}$$

  A W-buffer delivers a bad resolution if $r = f/n \approx 1$ thus has an incomplete storage range but comes at a low additional calculation cost.

Under the assumption of large distances $f > 100$ and high-precision floating point texture (16 or 32 bit) the 1/W-buffer and the complementary Z-Buffer are candidates for an optimal storage format for VDS [21, 22]. Under the additional assumption that most of the lenses will be placed in the mid-range of the scene, 1/W-buffers should be preferred.

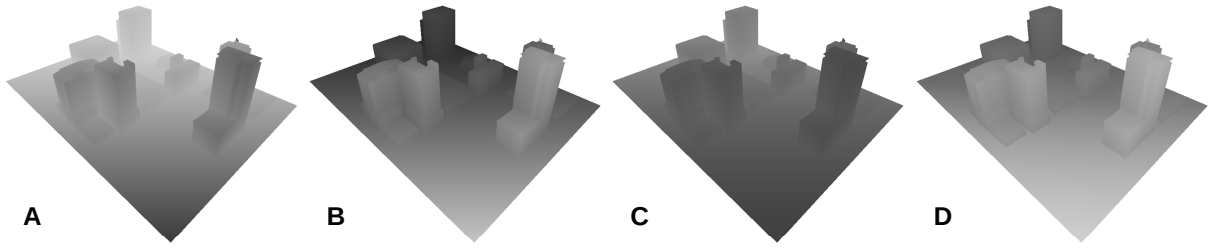


Figure 3.1.4: *Comparison of different depth-buffer approaches. A: Non-linear Z-buffer, B: Inverted non-linear Z-buffer, C: Linear W-buffer, D: Inverted linear 1/W buffer.*

3.2 Decomposition of Focus & Context Areas

To achieve focus and context visualization in combination with graphic hardware acceleration and scene-graph oriented graphic APIs it is necessary to distinguish the particular influence of the focus from the context. This differentiation can be achieved on different levels. Mainly, there are three possible approaches to determine whether a particular geometry falls into focus or context:

1. The *object-based approach* operates whilst evaluating the scene graph, *before* the geometry of shapes is sent to the rendering pipeline.
2. The *vertex-based approach* decides for each vertex $V \in \mathcal{WCS}$ if it is placed inside a lens or not.
3. The third approach performs this test for each fragment $F \in \mathcal{SCS}$ in image space and is from now on denoted as *image-based approach*.

The latter two approaches are unproblematic if programmable hardware is available. All the presented techniques have advantages and drawbacks which will be discussed in the next sections. These approaches can be combined as well. Table 3.2.1 shows the approaches that are applied by each lens type.

3D Information Lens	Object based	Vertex based	Image based
Color lens	o	o	×
Deformation lens	×	×	o
Occlusion lens	×	o	×

Table 3.2.1: *3D Information lenses classified after their decomposition approaches.*

3.2.1 Object-Based Approach

This is a coarse decomposition approach. It operates on per-object basis, i.e. on geometric shapes and their bounding volumes [24]. *Object-based decomposition* takes place during the traversal of the scene graph. Depending on the result of the decomposition the attributes of the scene-graph can be changed, copied, or extended in order to achieve lens functionality. Certain lens methods can require to alter geometry and need to save

original data [97]. This implies that an explicit access to the geometry or its bounding volume is essential for the accurateness of this test. Since this decomposition approach utilizes the CPU it is the most flexible of the three introduced methods. The inter-object occlusion test (see section 3.3.1) is an example for this class of methods.

3.2.2 Vertex-Based Approach

Vertex-based focus and context decomposition represents the next possible refinement level. This approach can applied in world and eye space coordinates and is important for the implementation of deformation lenses. For each vertex $V \in \mathcal{WCS}$ can be decided if it belong the focus or the context.

Vertex-Based Approach with Analytic Shapes

A simple and intuitive method to tackle the decomposition problem are analytical methods. They are suitable for analytic shapes like cylinders, spheres or cubes. It represents a limited approach that can be only applied to a small selection of shapes and is mentioned for the completeness. Especially for further applications it could be useful to provide more degrees of freedom. To overcome this limitations the next section presents a technique that enables the access of raster data for parametrization.

Vertex-Based Approach with Textures

Consider a VDS that represents a convex shape or just a contour of this shape. VTF allows the access to the texture data of the VDS. The following section describes an approach to calculate texture coordinates for a given vertex. It is has parallels to the *projective texturing* method described in [23]. However, the presented solution allow more control and overcomes problems like reverse projection. The algorithm can be implemented in a vertex shader. After gaining access to the VDS or an other arbitrary texture which represent the lens shape, we can determine how the vertex is affiliated to a lens shape. This vertex-based approach is necessary for the concept of deformation lenses that will introduced in section 3.6. Considering a plane $\mathcal{P} = (F, A, B)$ and the scaling vectors $w, h \in \mathbb{R}_{\setminus 0}$. The plane is defined by it normal vector $N = A \times B$, the base vector $F \in \mathcal{WCS}$ and the normalized direction vectors $A, B \in [0, 1]^3$. The function

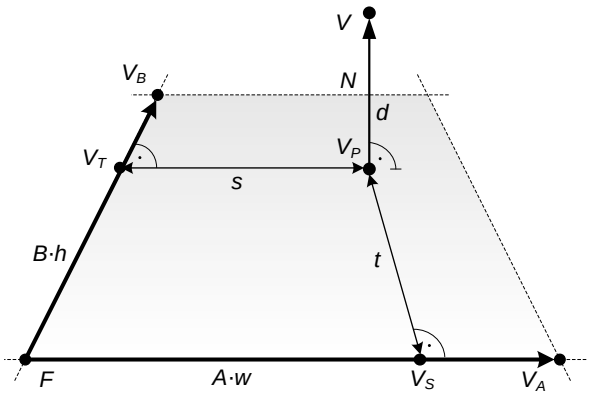


Figure 3.2.1: *Texture coordinate calculation.*

$$\kappa : \mathcal{WCS} \times \mathcal{P} \longrightarrow [0, 1]^2 \tag{3.5}$$

automatically generates texture coordinates texture coordinates $s, t \in [0, 1]$ for a given vertex $V \in \mathcal{WCS}$. It is defined by:

$$s = \frac{|F - V_S|}{|A \cdot w|} \qquad t = \frac{|F - V_T|}{|B \cdot h|} \tag{3.6}$$
$$V_P = \rho_{Plane}(V, F, N) \quad V_S = \rho_{Line}(V_P, F, V_A) \quad V_T = \rho_{Line}(V_P, F, V_B)$$

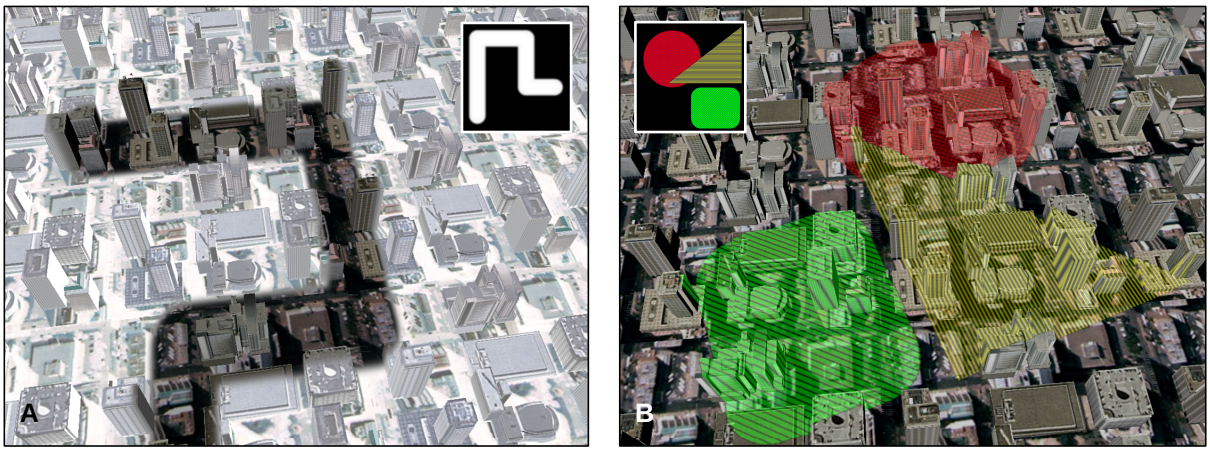


Figure 3.2.2: *Applications of vertex-based approach for color lenses.*

This function use two kind of projections: ρ_{Plane} projects a vertex onto a plane while ρ_{Line} determine the lot of the vertex onto a given line. These functions are defined as follows:

$$\rho_{Plane}(P, O, N) = P - ((P - O) \bullet N) \cdot N \tag{3.7}$$

$$\rho_{Line}(P, A, B) = A + (P - A) \cdot ((P - A) \bullet \|B - A\|) \tag{3.8}$$

To determine if V lay in the correct half-space of $\mathcal{P}$ one can apply the following boolean test:

$$\vartheta(V_P, A, B, V_S, V_T) = \begin{cases} 1 & if \quad (A \times N) \bullet (V_S - V_P) < 0 \wedge (B \times N) \bullet (V_T - V_P) < 0 \\ 0 & otherwise \end{cases} \tag{3.9}$$

Using the above equations one can determine if a vertex V is associated with the focus or the context. This method is very flexible and allows the representation of arbitrary 2D lens shapes (see figure 5.3.3). So far this approach is limited to two dimension. By calculating the distance $d = V_P - V$ we have access to a third dimension that allows a *volumetric depth test* in the world coordinate system. The next section describes and demonstrate the application of this method in image space.

3.2.3 Image-Based Approach

The image-based approach is essential for lens algorithms that operate in image space. It delivers pixel-precise results [24] and is implemented using shader programs. Analogue to the vertex-based decomposition we can distinguish between two approaches of different flexibility. For each fragment $F \in \mathcal{SCS}$ and a given VDS, which describes the dimension of a lens (the focus area), a *volumetric* or *two-sided* depth test can determine whether F is in the focus area or not. The classic depth test performs a boolean operation $\circ \in \{<, >, \leq, \geq, =, \neq, Never, Always\} : \mathbb{D} \times \mathbb{D} \longrightarrow \mathbb{B} = \{0, 1\}$ upon $d_I \circ d_C$ where d_I is the incoming depth of a fragment and d_C the current depth in the depth buffer. The depth values are normalized in $\mathbb{D} = [0, 1] \subseteq \mathbb{R}$. A volumetric depth test:

$$\delta_F : \mathbb{D} \times \mathbb{D} \times \mathbb{D} \longrightarrow \mathbb{B} \tag{3.10}$$

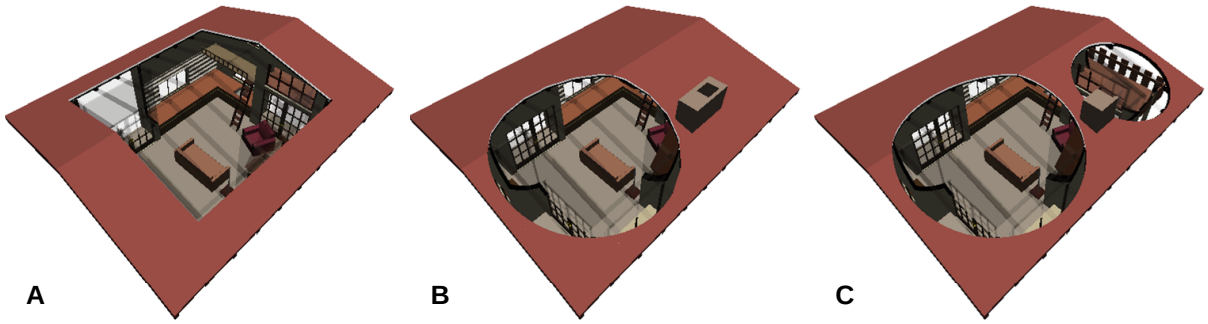


Figure 3.2.3: *Application of the image-based focus and context separation and integration on the example of an intra-object occlusion lens using a volumetric depth sprite of a cube (A), a sphere (B) and multiple spheres with different radii (C).*

can perform the following modi $F \in \{Inside, Outside, Equal, Never, Always\}$ on an incoming depth d_I and the front- and back-depth $d_F, d_B \in \mathbb{D}$ of a VDS respectively:

$$\delta_{Inside}(d_I, d_F, d_B) = \begin{cases} 1, if(d_F < d_I) \wedge (d_I < d_B) \\ 0, otherwise \end{cases} \tag{3.11}$$

$$\delta_{Outside}(d_I, d_F, d_B) = \begin{cases} 1, if(d_F > d_I) \vee (d_I > d_B) \\ 0, otherwise \end{cases} \tag{3.12}$$

$$\delta_{Equal}(d_I, d_F, d_B) = \begin{cases} 1, if(d_F = d_I) \vee (d_I = d_B) \\ 0, otherwise \end{cases} \tag{3.13}$$

Figure 3.2.3 shows some examples of the volumetric depth test using single and multiple shapes.

3.3 Occlusion Lenses

As the name suggests, the task of an occlusion lens in virtual 3D city models is the compensation of a certain kind of occlusions. In this application the viewer encounters two kinds of categories: *intra-object occlusions* and *inter-object occlusions*.

Intra-Object Occlusion describes a seldom case of occlusion in 3D city models and is of secondary importance in this section. Considering a LOD-4 building [43, 31] which interior is occluded by its surrounding walls. Intra-object occlusion occurs if the viewer wants to see into a building or the view is blocked inside a building by interior walls. Thus, intra-object occlusion denotes the partial occlusion of object parts by each other [58]. This occurs mainly within *point-based user activity*. Figure 3.3.1 shows an approach to deal with intra-object occlusions by using *selective order-independent transparency rendering* via depth peeling [8] (see section 2.3) in combination with a non-uniform alpha value distribution along the peeled color layers. Thereby, the alpha value of the color maps decreases toward the COP. Given a number of color layers $n \in \mathbb{N}$. The alpha value $a \in [0, 1]$ for a color layer $L_i \in \{0, ...n\}$ can be calculated using a smooth step-function:

$$\begin{aligned} a &= t^2 \cdot (3–2 \cdot t) \\ t &= \min(\max(i/n, 0), 1) \end{aligned} \tag{3.14}$$

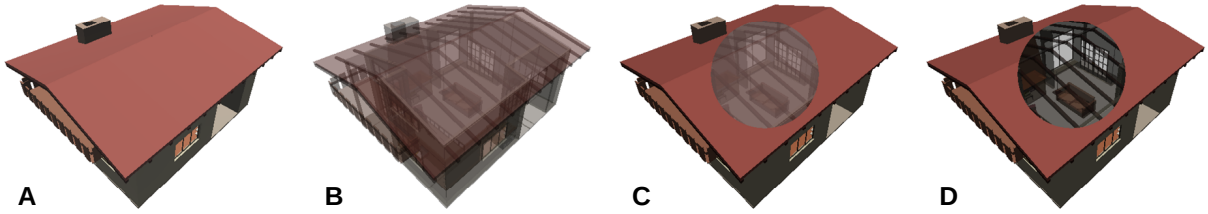


Figure 3.3.1: *A comparison of screen-aligned intra-object occlusion lenses. A: Opaque rendering of the Maybeck Studio, B: Transparent rendering with layers of uniform alpha values, C: Selective transparency with layers of uniform alpha values, D: Selective transparency with layers of non-uniform alpha values.*

Here, n is the farthest layer from the camera. The advantage of a non-uniform alpha distribution can be perceived by comparing the sub-figures 3.3.1.C and 3.3.1.D.

Inter-Object Occlusion or scene occlusion denotes the occlusion between two structures. The most common case would be the inter-object occlusion between a number of buildings. This targets mostly *local-based* user activity. An object that is hidden behind another one will be denoted as *occludee*. An object that hides an occludee is denoted as *occluder*. In this case an occluder prevents the user from gaining access to visual, spatial or structural information of the occludee. The benefit of resolving inter-object occlusion lies in a decrease of navigation overhead for the user. This can be achieved by reinterpretation or transformation of the occluder's building information, like structure or their appearance in the vicinity of the viewers location. The reinterpretation in form of special rendering techniques allows a surplus with the preservation of occluder information (like shape or texture) in combination with the information of the occludee. This could also be applied to the editing process of a city model. However, the concept of an occlusion lens is quite simple. It uses the object-based separation approach for a set of city structures like buildings or vegetation objects to divide it into two disjoint sets based on their characteristic properties regarding the users COP. The *occluder area set* contains all structures that are classified as occluders by failing an *occlusion detection test* (see section 3.3.1). All other structures are part of the complementary *occludee area set*. There are three different rendering techniques for structures of the occluder area set:

- Discard any geometry of the occluding object.
- Apply visual abstraction methods to the occluder object.
- Flatten the geometry of the occluder and preserve special features of the object (e.g. façade texture information).

The following section will present three occlusion tests for city structures under the assumption that an *axis-aligned bounding box* (AABB) is available that describes the volume of the structures.

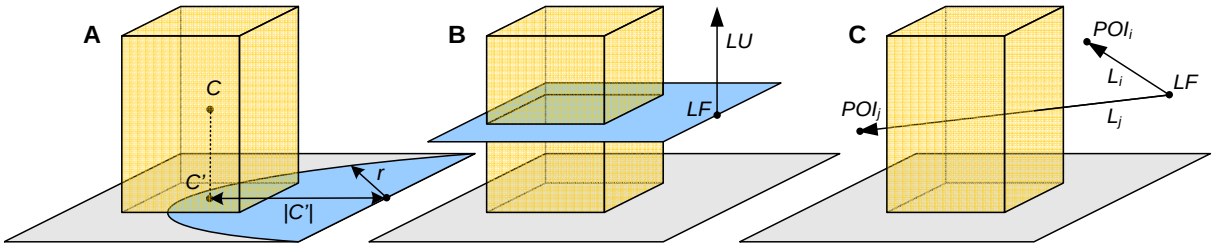


Figure 3.3.2: *Comparison of occlusion detection tests for occlusion lenses. A: Spherical occlusion test, B: Viewers Height occlusion test, C: View Axis occlusion test.*

3.3.1 Occlusion Detection Tests

In the context described above, *occlusion detection* is the categorization of each building or other structures as an occluder or occludee. If such a structure is determined as an occluder, one of the three techniques can be applied to it. Considering the user's orientation $L = (LF, LT, LU) \in \mathbb{R}^3 \times \mathbb{R}^3 \times \mathbb{R}^3$, whereas LF represents the COP, LT the view direction vector, and LU the up-vector of the camera. Together with the buildings $AABB = (LLF, URB)$, an *occlusion detection test* or function delivers a boolean value *true* if the structure is categorized as an occluder:

$$\omega_F(AABB, L) \longmapsto \mathbb{B} \tag{3.15}$$

For a particular function instance F a differentiation between the following three occlusion detection approaches is possible. The values 1 and 0 will be further referenced as $true$ and $false$.

Spherical Occlusion Detection Given a radius $r \in \mathbb{R}$ that determines the area of occlusion around the COP and the centre point $C \in \mathbb{R}^3$ of the AABB we can define a *spherical occlusion detection test* as follows:

$$\omega_{Spherical}(ABBB, L) = \begin{cases} 1, & if \quad |\mathbf{MV} \cdot C| < r \\ 0, & otherwise \end{cases} \tag{3.16}$$

Hereby the vector C is transformed into the camera coordinate system by multiplication with the current model view transformation matrix $\mathbf{MV}$. It is also possible to use another reference point instead of C or even a set of reference points to test against in order to deliver more accurate results. The above equation also express a cylindrical occlusion test by projecting C onto a plane with origin LF and normal vector LU so that $C' = \rho_{Plane}(C, LF, LU)$. (compare to figure 3.3.2.A).

Viewers-Height Occlusion Detection This approach extends the *spherical occlusion detection test*. Given the users orientation L and the AABB, a building can be categorized as occluder if the AABB intersects the plane with origin LF and the normalized vector LU. Compare to figure 3.3.2.B.

$$\omega_{Plane}(ABBB, L) = \omega_{Spherical}(ABBB, L) \wedge intersect_{Plane}(AABB, L) \tag{3.17}$$

This test can also be accomplished in image-space.

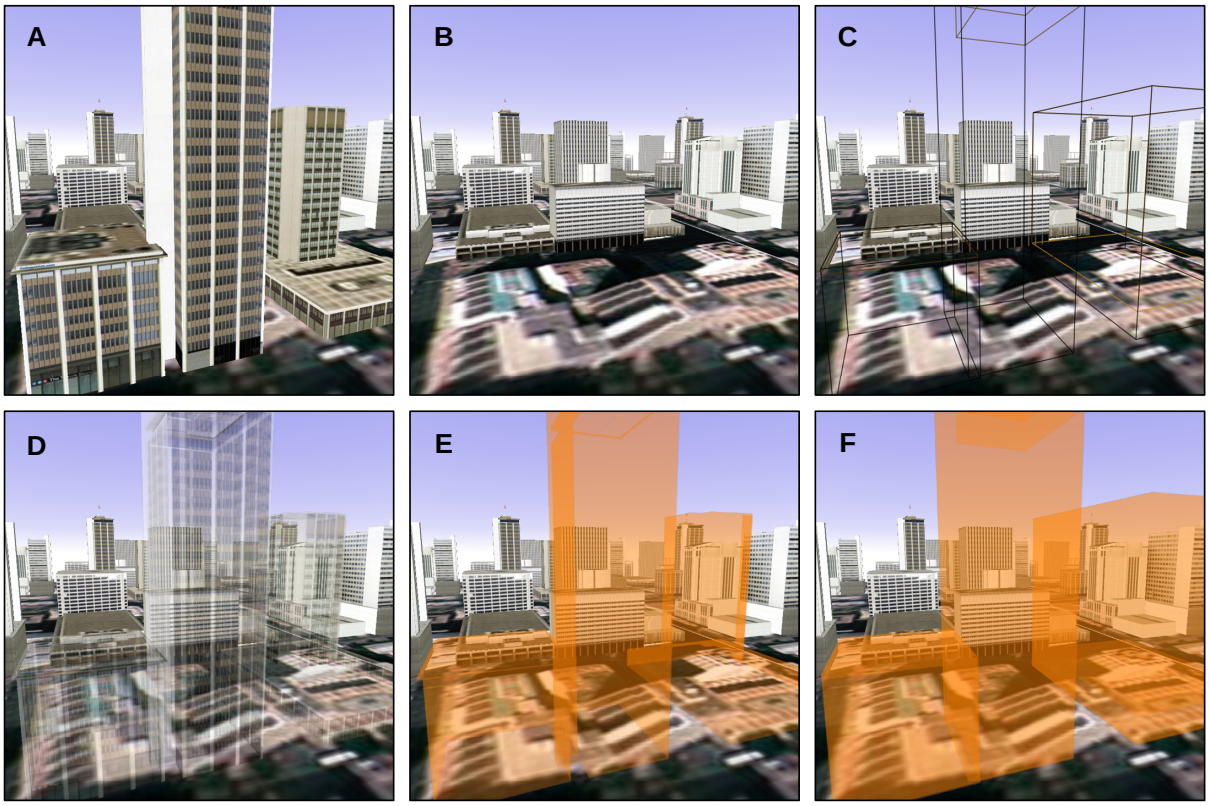


Figure 3.3.3: *Examples for visual abstraction of buildings. A: Occluder buildings, B: Discarded occluder geometry, C: Rendering of the occluders bounding box, D: Transparent rendering of the occluder with preserved texture information, E: Transparent rendering of the occluder with discarded texture information, F: Transparent rendering of bounding box.*

View-Axis Occlusion Detection Given a set of scene POIs $S = \{POI_0, \ldots, POI_n\}$ (see figure 3.3.2.C), and the user's orientation L, the *view-axis occlusion test* will deliver a positive result if the AABB intersects at least one view axis A_i, with $A_i = ((0,0,0), \mathbf{MV} \cdot POI_i)$. Formally spoken:

$$\omega_{Axis}(AABB, L) = \begin{cases} 1, & if \sum_{i=0}^{n} intersect_{Line}(AABB, A_i) > 0 \\ 0, & otherwise \end{cases} \tag{3.18}$$

A fast AABB intersection test $intersect_{Line}$ is described in [52].

3.3.2 Rendering of Occlusion Lenses

Depending on the method of resolving the occlusions the rendering of occlusion lenses can be done using multi-pass rendering:

1. **Context Rendering:** The first pass renders the terrain geometry of the city model and all structures classified as occludees.

2. **Focus Rendering:** The second and all successive passes render only occluder geometry by applying rendering techniques for visual abstractions.

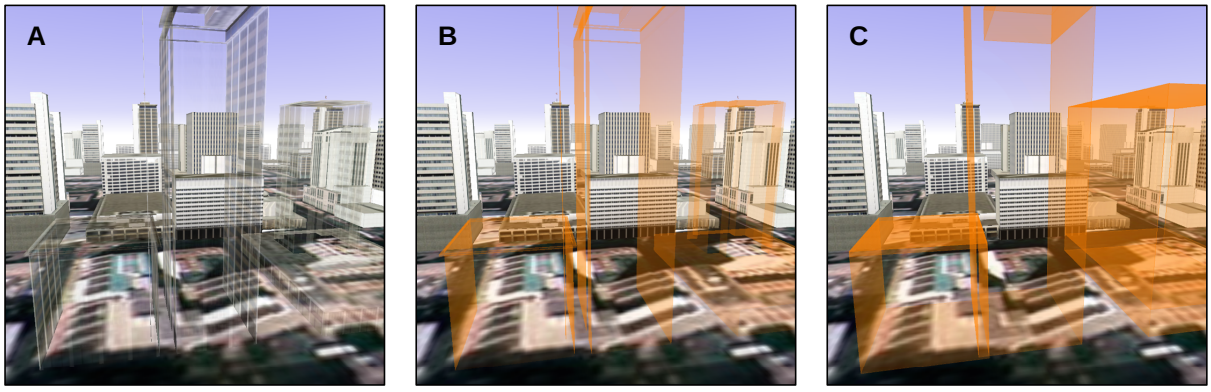


Figure 3.3.4: *Example of an inter-object occlusion lens utilizing an x-ray shader: with preserved texture information (A), with color abstraction (B), and with shape and color abstraction.*

The next two sections focus on rendering techniques that are able to convey certain aspects or information of the occluder.

3.3.3 Visual Abstraction

In the context of 3D city models one can roughly distinguish between two kinds of visual abstractions: the abstraction of a building's shape or its façade information, like shading, texturing [53, 33] or edges [69]. For navigation and orientation issues it could be necessary to maintain a generalized shape of an occluder building. Besides a generalized hull [100, 101] the building's axis-aligned bounding box (AABB) is such a generalized shape. Order-independent transparency as described in section 2.3 serves to simplify the buildings color information. Figure 3.3.3 shows several examples of occlusion lenses with different levels of shape appearance and abstraction. The visual abstractions possess some disadvantages. To omit all occluder information can irritate the user. Sub-figures C and D suffer from a low contrast between the bounding box or the transparent shapes and the scene. But decreasing the transparency leads to a reduced perception of the occludees (sub-figures E and F). The essence of resolving occlusions is presented by a so called *X-Ray* or *Ghost Shader*. All surfaces that are parallel to the view plane become transparent. This reduces the number of overlapping transparent shapes and increases the perception of the occludees. The approach is reasonable by considering the generally cubical form of the buildings. For a given vertex $V \in \mathcal{WCS}$ and its corresponding normal $N \in \mathcal{CCS}$ the opacity term $o \in [0, 1]$ is calculated as follows:

$$o = 1 - \Big(\big| \| - N \| \bullet \| - (\mathbf{MV} \cdot V - O) \| \big| \Big)^e \tag{3.19}$$

The edge fall-off parameter is determined by $e \in [0, 1]$. The opacity term o can be mapped directly onto alpha value of the vertex V. To achieve more user control, the vertex color and opacity can be sampled from 1D textures depending on o. Figure 3.3.4 demonstrates the results.

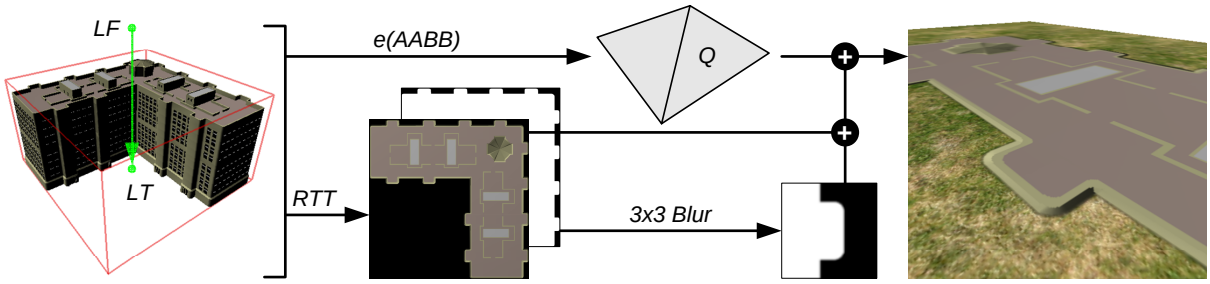


Figure 3.3.5: *Concept of flatten lens.*

3.3.4 Flatten Geometry

Another possibility to overcome object occlusion is flattening the object which occludes a POI. This approach can be useful for recognizing the original position and the dimension of the base area of the occluder.

Ad Hoc Solution An object can be flattened just by setting the particular vertex component $V = (x, y, z, 1) \in \mathbb{R}^4$ to an specified value $v \in \mathbb{R}$ which represents the base of the object. Hereby, v can be taken from buildings AABB. Consequently, flatting can be done using the following transformation:

$$V' = V \cdot \mathbf{M}_S \cdot \mathbf{M}_T(v), \quad \mathbf{M}_T(v) = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & v \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}, \quad \mathbf{M}_S = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{3.20}$$

The quality of this approach is sufficient if objects in the far or middle distance are supposed to be flattened. Since the solution delivers dissatisfying results near the COP this thesis propose a high-quality approach which is described in the following.

Advanced Solution Figure 3.3.5 shows the creation process. An approach that delivers better results can be achieved by using a *render to texture* (RTT) technique (see section 2.3) for the focus rendering. The object has to be flatten against the ground XZ-plane by using an orthographic projection. Assuming an axis-aligned bounding box $AABB = (LLF, URB) \in \mathbb{R}^3 \times \mathbb{R}^3$ one could calculate the additional vectors that are denoted by a combination of F for front, B for back, L for left, R for right, U for upper, and L for lower. For offscreen-rendering, the orthographic projection with a viewing volume is used, defined by: $top = LRB_x, bottom = LLF_z, left = LLF_x, right = LRB_x, near = d$, and $far = LLF_y$ is used. This ensures the correct size and proportions for the texture. d is the near distance which can be defined by the user. The orientation of L is defined as look from top with:

$$L = ((LLF + LRB)/2 + (0, |URB_y - LLF_y| + d, 0), (LLF + LRB)/2, (0, 0, 1)) \tag{3.21}$$

In order to achieve accurate lighting, we have to consider the following issue: The OpenGL specification [50] states that a light position is converted automatically to eye space coordinates, i.e. camera coordinates. Let $\mathbf{M}_{SP}, \mathbf{M}_{SO}$ be the projection and orientation

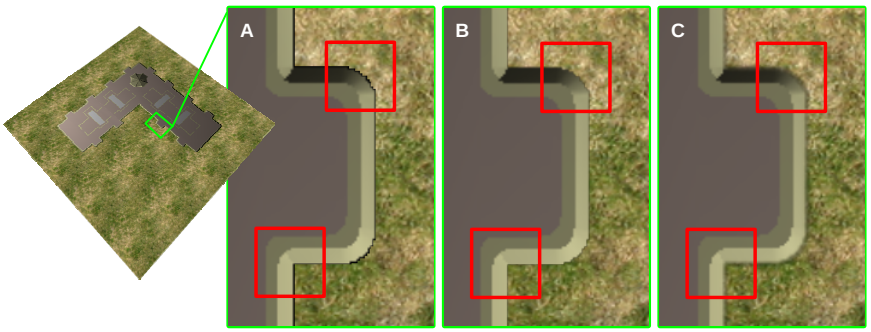


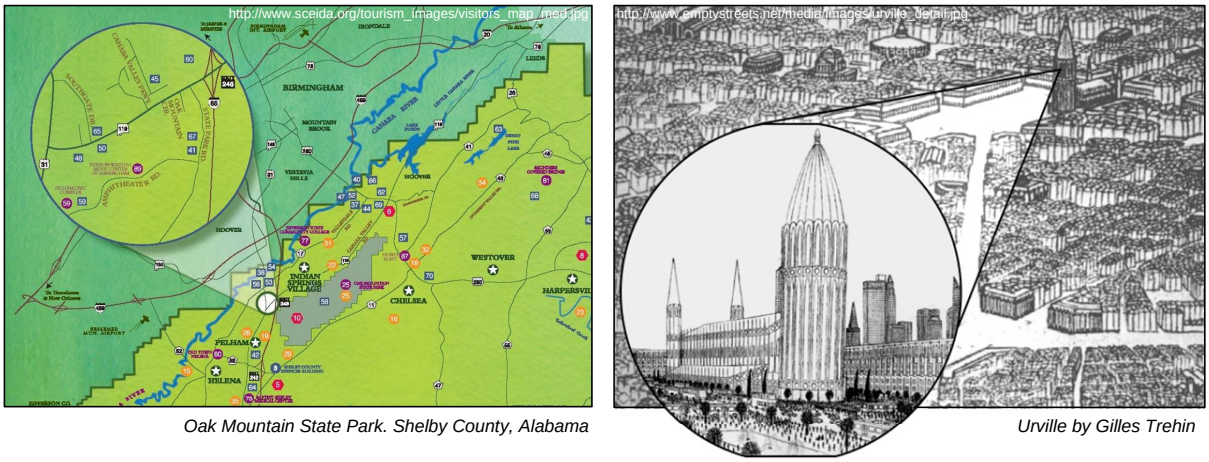
Figure 3.3.6: *Comparison of different integration methods for a flat-lens texture using alpha test (A), simple alpha blending (B) and alpha blending with adaptive blur of the alpha mask and the texture edges (C).*

transformation matrices of the scene camera and $\mathbf{M}_{LP}, \mathbf{M}_{LO}$ be the matrices in respect to the local camera. There are at least two possibilities to fix this problem. One could use a vertex shader to correct the lighting respectively to $\mathbf{M}_{SO}$ or apply *deferred shading* by employing a fragment shader in combination with *multiple-render-targets* (MTR). However, the solution can be done without the usage of shaders. We have to adapt the orientation matrix for the local camera. The correct lighting transformation settings $\mathbf{M}'_{LP}, \mathbf{M}'_{LO}$ can be calculated as follows:

$$\mathbf{M}'_{LP} = \mathbf{M}_{LP} \cdot \mathbf{M}_{SO}^{-1}, \qquad \mathbf{M}'_{LO} = \mathbf{M}_{SO} \cdot \mathbf{M}_{LO} \tag{3.22}$$

These transformation corrections must be made every time the orientation of the scene camera changes. To take the light attenuation into account, the flatten object has to be translated. The translation vector T can determined by $T = (0, -(|LLF_y| + |URB_y|, 0)$. Instead of rendering the building geometry, the technique renders a textured quad Q which was extracted from the AABB. To avoid *Z-fighting*[2], the depth test should be disabled. An alternative integration method can be done by *projective texturing* which requires an additional rendering pass and also the storage of the texture. The integration of the flatten texture can be enhanced by blurring it alpha mask before texturing the quad Q. Figure 3.3.6 depict the comparison between alpha testing (A), simple alpha blending with sharp alpha mask (B) and alpha blending with blurred alpha mask (C).

[2]Z-fighting is a phenomenon that occurs when two coplanar primitives have similar values in the Z-buffer.



Oak Mountain State Park. Shelby County, Alabama

Urville by Gilles Trehin

Figure 3.4.1: *Custom-made best-view lenses and context-lines that integrate the focus or the context depictions into a scene.*

3.4 Best-View Lens

This class of lenses is designed to aid the exploration of 3D city models by depicting distant locations in virtual worlds. The concept of a *3D best-view lens* (BVL) evolves from the principle of *2D context maps* and the *Through-the-Lens* metaphor [99]. Context maps represent an information space at a larger or smaller scale by providing overview or detail information. They allow dynamic interactive positioning of the local detail without compromising spatial relationships [113] severely. This facilitates combined 2D/3D interfaces that extends annotations of 3D scene objects by linking and referencing complex 2D views with callout lines [96]. This *overview and detail* approach has been found useful in previous studies [54] but in general, context maps suffer from two essential problems:

- **Occlusion problem**: A lens (e.g. magnification lens) occludes parts of its context if it is placed over the area of interest.
- **Continuity problem**: If the focus or context areas are dislocated the association between them is difficult for the user.

We can encounter both problems with best-view lenses too. Another related visualization approach is *multiple views* or *multiple viewport* metaphor [10, 2]. A single scene is rendered from different camera positions into multiple viewports. Basically, a BVL is an abstraction of this technology. It overcomes the spatial separation of multiple viewports by integrating them into the scene rendering. That can be achieved by placing focus and context overlays on separated parts of the viewport together with the depiction of their associations to the POI in the scene. This image-based approach is used frequently. Two examples are shown in figure 3.4.1 and 2.2.1. They demonstrate different aspects of usage. Figure 2.2.1 provides context information while figure 3.4.1 shows a focus view of a scene object in the distance.

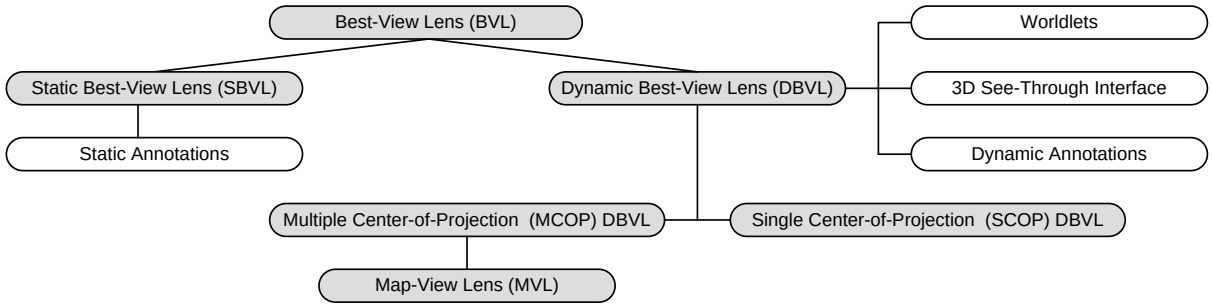


Figure 3.4.2: *Taxonomy of best-view lenses covered by this thesis.*

We can find different applications in virtual 3D city models for a best-view lens:

- It allows the depiction of city-model objects and their special aspects in a priority based manner.
- It supports focus and context navigation by integrating *landmarks*, photographs of POI, and other raster data into the scene rendering.
- It eases the creation and integration of *dynamic scene-bookmarks*.
- It facilitates the tracing of moving objects in the virtual scene.

3.4.1 Lens Models

Figure 3.4.2 provides an overview of the best-view lenses introduced in this section. It reveals that *worldlets* [18], *static* as well as *dynamic landmarks* and *map-views* are special instances of a BVL. They can be implemented with a single framework. Its concept consists mainly of the following two parts:

- **Overlay:** An image that is placed over the scene rendering. It contains either the focus or the context rendering.
- **Context-Line:** It represents the association between the overlay and a point in the scene.

A *static* BVL associates a given depiction or photo with a geo-referenced object, while a *dynamic* BVL creates this depiction by rendering the city model with a local camera. The parameters of the local camera, particularly its orientation and projection, can be altered per frame. A further specialization into *multiple-center-of-projection* (MCOP) and *single-center-of-projection* (SCOP) best-view lenses, is based on the orientation constraints of the local scene camera.

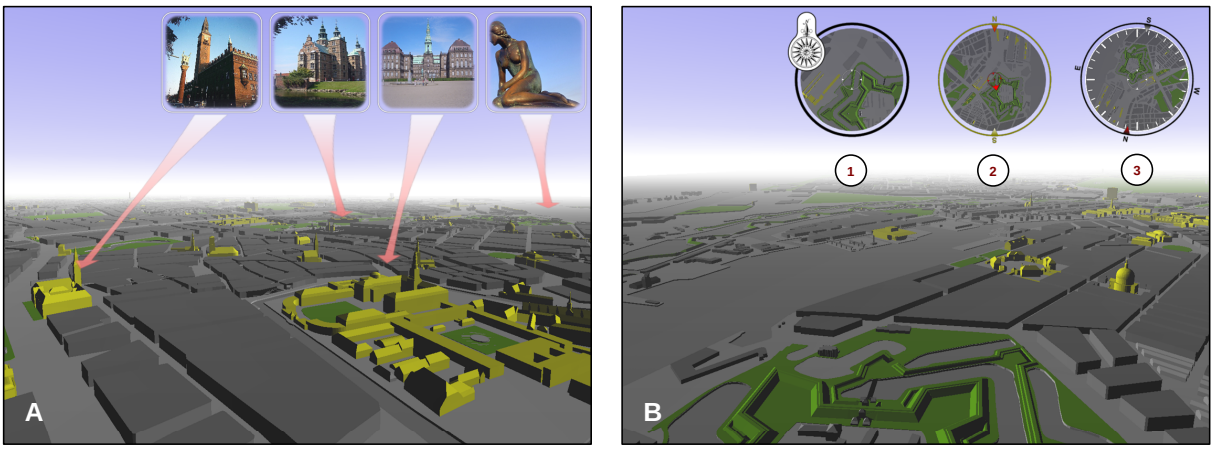


Figure 3.4.3: *Examples of a static best-view and a map-view lens.*

Static Best-View Lens

A *static best-view lens* (SBVL) or *annotation lens* integrates predefined raster data with the viewport and allows an association with a given number of locations. Figure 3.4.3.A demonstrates an application. In contrast to the classical landmark and POI rendering, the static depiction is placed in an overlay on the viewport and not in the scene directly. This is reasonable if the distance between the viewer and the POI is large. The overlay can also occlude the scene partially. In this case, one can use a tilted 3D *call-out* which could resolve some occlusions of this local context [84]. The call-out principle is a visual device for associating annotations with an image, program listing, or similar figure and has its origin in typesetting. Each location is identified with a mark, and the annotation is identified with the same mark. This is somewhat analogous to the notion of footnotes in print. An advantage of this lens type is the representation of an arbitrary number of relations between overlay and scene.

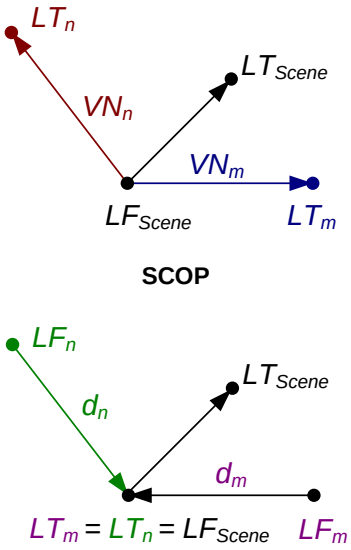


Figure 3.4.4: *Comparison between a SCOP and MCOP best-view lenses.*

Dynamic Best-View Lens

A *dynamic best-view lens* (DBVL) is represented by a local scene camera that delivers the focus or context rendering for the overlay representation. The DBVL can be dependent or independent of the viewers position. Using dynamic BVL instead of static ones has some major advantages. Focus and context exhibit a homogeneous appearance. This increases the probability of recognizing the depicted scene objects. The animation of the local camera enables tracing a number of moving objects in the scene without forcing the observer to move. Besides free positioning of camera in the scene, its location can also be calculated by

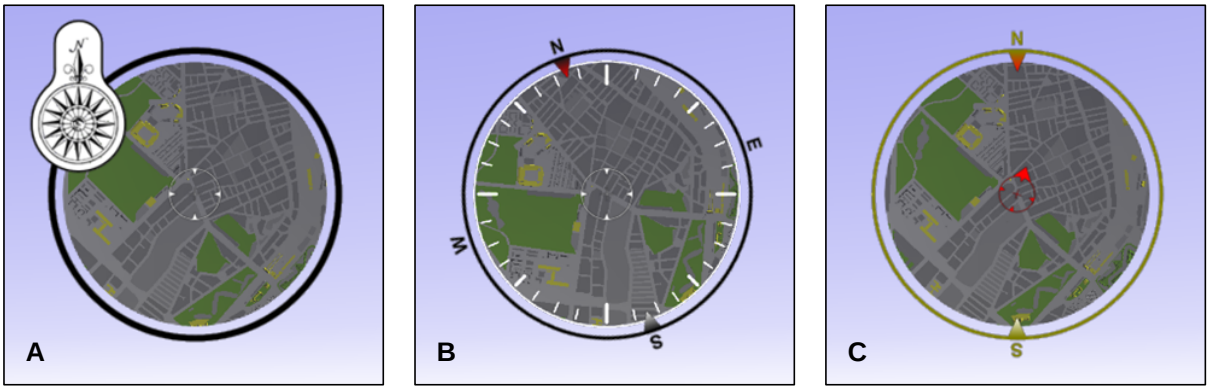


Figure 3.4.5: *Rendering of map-view lenses. A: North-up map-view lens, B: Head-up map-view lens, C: North-up map-view lens with indication of the user's view direction.*

using a given bounding box. This requires the scene object to have a *best-view normal* (BVN) that indicates a preferred view on the object. For a projection with a horizontal field-of-view (FOV) of 90 degrees and a bounding sphere $\mathcal{S} = (M, r)$ of the object, the orientation L_{BVN} of the local scene camera can be calculated as follows:

$$L_{BVN} = (LF_{BVN}, M, LU_{Scene}) \quad \text{with} \quad LF_{BVN} = M + BVN \cdot \sqrt{2 \cdot r^2} \tag{3.23}$$

$M \in \mathbb{R}^3$ denotes the center and $r \in \mathbb{R}$ the radius of the sphere. For a given viewer orientation $L = (LF_{Scene}, LT_{Scene}, LU_{Scene})$, a differentiation of DBVL into two categories is possible. The *multiple-center-of-projection* (MCOP) DBVL is always orientated toward LF_{Scene}. A *single-center-of-projection* (SCOP) DBVL obtains this vector as COP, but possesses different view directions. A rear view mirror is an example of such a DBVL. Figure 3.4.4 clarifies the distinction of both sub-classifications by using two local camera setups $L_{LC_n} = (LF_n, LT_n, LU_n)$ and $L_{LC_m} = (LF_m, LT_m, LU_m)$. Given the view direction normal $VN \in \mathbb{R}^3$ and a distance $d \in \mathbb{R}$, the orientation of the respective local scene camera $L_{MCOP} = (LF_{MCOP}, LT_{MCOP}, LU_{MCOP})$ can be calculated by

$$L_{MCOP} = (LF_{Scene} + (VN \cdot d), LF_{Scene}, LT_{Scene} - LF_{Scene}) \tag{3.24}$$

The absolute orientation $L_{SCOP} = (LF_{SCOP}, LT_{SCOP}, LU_{SCOP})$ of the SCOP DBVL can be determined by

$$L_{SCOP} = (LF_{Scene}, LF_{Scene} + VD, LU_{Scene}) \tag{3.25}$$

The view direction of this orientation is expressed by the normal vector $VD \in \mathbb{R}^3$.

Map-View Lens

A *map-view lens* (MVL) represents the specialization of a MCOP best-view lens. This method provides context information by generating an overview of the current viewer's surrounding. Figure 2.2.1 shows two applications of map-view lenses. The principle orientation setup of the local scene camera is calculated using equation (3.24). We can distinguish between three different dynamic map-view lenses in general [11] (compare to figure 3.4.5).

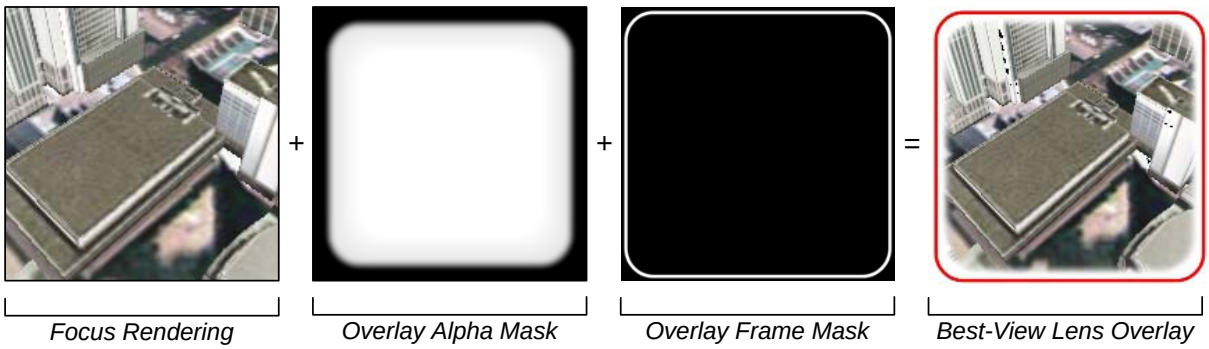


Figure 3.4.6: *Components of an overlay. The focus rendering or the static annotation texture will be alpha-blended with the overlay alpha-map and the frame-mask.*

- **North-Up Mode:** The map view is always aligned north. In this modus the user cannot determine its heading from the map view. The up-vector of the local scene camera is corrected according to a scene north vector $N_S \in \mathbb{R}^3$ so that: $LU_{SC} = \|N_S\|$. This viewer position is centered in the overlay (see figure 3.4.5.A).
- **Track-Up or Head-Up Mode:** The map view is aligned with the current view direction of the user. Figure 3.4.5.B shows a compass that allows orientation within the context.
- **North-Up with User-View Mode:** It operates by using the same principle as the North-Up mode, but possess an symbol that visualizes the users heading direction (compare to figure 3.4.5.C).

3.4.2 Overlays

Overlays allow th displacement of the focus rendering from its original position in the scene. Simultaneously, they also solve the occlusion problem partially. An overlay is mainly represented by its dimensions on the viewport and its transparency. The dimensions will be defined by a layout. The essential task of the layout is to manage the available screen space in an efficient way. The overlay style can be configured by using 2D texture maps for the alpha mask [24] and the frame mask (see figure 3.4.6). The texture representation of overlay alpha and frame mask afford a high of design possibilies and facilitates the application of irregular lens shapes.

3.4.3 Context-Lines

"One challenge in navigating through any large data space is maintaining a sense of relationship between what you are looking at and where it is with respect to the rest of the data" [6]. A context-line (CL) visualizes the association between the overlay and the corresponding point in the scene. This often can be seen in touristic visualizations of a city or a particular area (see figure 3.4.1 left). Subjectively, users prefer to have a linked overview, but they are not necessarily faster or more effective using it [55]. A context line is mainly defined by a focus anchor FA and a context anchor point CA (compare to figure 3.4.7). The focus anchor point is the projected lens position. The usage of context

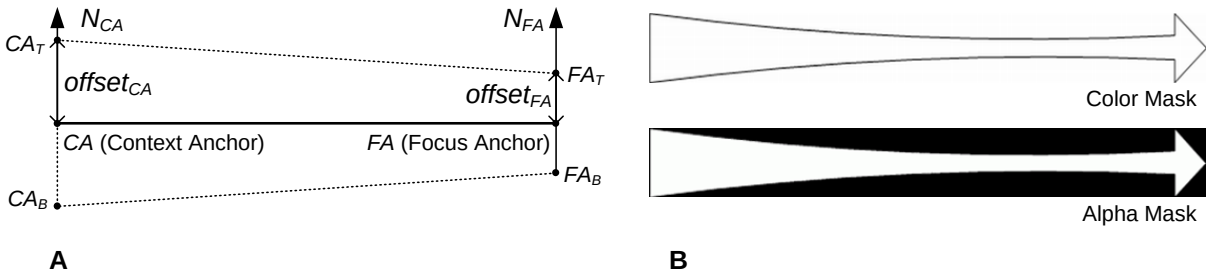


Figure 3.4.7: *Concept of a straight context-line. A: the quad is represented by two vectors CA and FA with their offsets. B: Example skin for the context-line.*

lines leads to two problems. On the one hand, two context lines can overlap or interfere each other. The resulting confusion of the user can be avoided partially by sorting the context lines according to their alignment. On the other hand, the context lines occlude important areas of the scene. By acting on the following assumption, the problem can be solved: The look-at point is the current center of interest within the scene. With a given radius $r \in \mathbb{N}$ around the screen center $C = (w/2, h/2) \in \mathcal{SCS}$, and a drop-off parameter $e \in \mathbb{N}, 0 \leq e \leq r$, we can fade the alpha value $a \in [0, 1]$ of an occluding context line fragment $F \in \mathcal{SCS}$ (see figure 3.4.8) according to

$$a = mix(0, 1, smoothstep(e, r, d)) \quad \text{with} \quad d = |C - F| \tag{3.26}$$

The description of problems and their solutions above evolve into a generic solution in form of globally applied constraints that can alter the visibility or appearance of a context-line.

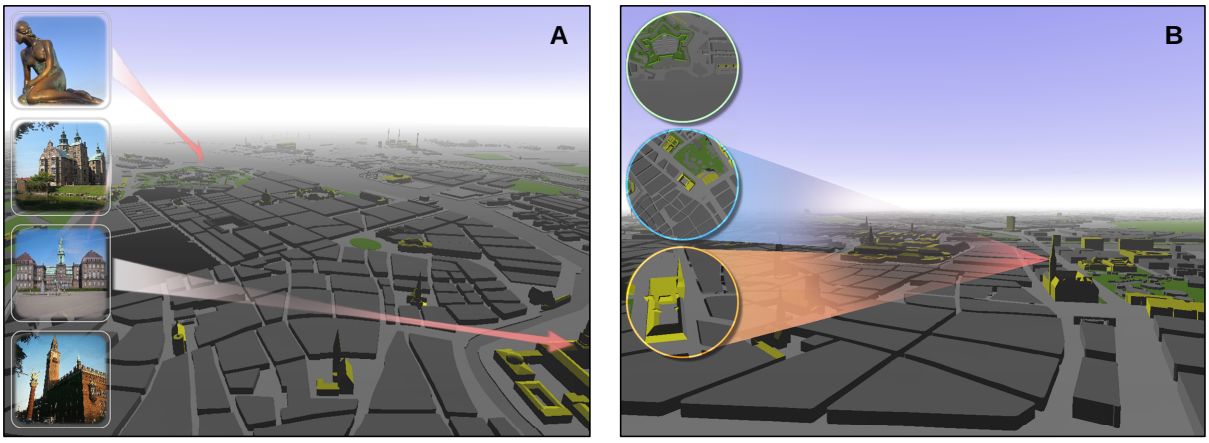


Figure 3.4.8: *Examples of visibility constraints for context-lines. A: Demonstration of CL ordering and hiding. B: Alpha fading towards the center of interest.*

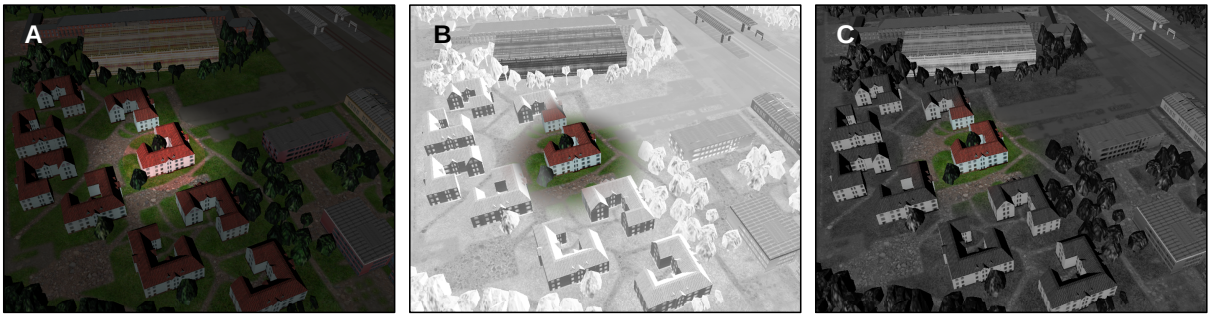


Figure 3.5.2: *Examples of different post render styles which supporting preattentive perception. A: Context cueing, the context is darkened while the focus remain unchanged. B and C: Same principle, but the color information of the context discard and then inverted.*

3.5 Color Lens

A *color lens* is able to integrate different renderings of the same scene geometry using the same projection. It is an image-based technique that allows *hybrid rendering*, i.e. mixing of photorealism and NPR [93, 92]. It facilitates the implementation of rendering techniques which support preattentive perception and allows the extension of the available expression dimensions in a visualization environment. The encoding of information by different *rendering styles* is a common method. When appropriately used, graphical features such as shape, size, color, and position have proved to be effective in information visualization because they are mentally economical, rapidly and efficiently processed by the preattentive visual system rather than with cognitive effort [4]. In the context of geovirtual environments, these encodings can be used to identify, localize, correlate, and categorize city model objects as well as to find data distributions. For example: the visualization of data significance in 3D, the catchment area of schools, hospitals, supermarkets or the coverage of telecommunication antennas. Color lenses also enable the visualization of spatial-temporal aspects or differences. This lens approach addresses a *global user activity* and applies the *image-based* focus and context separation/integration method presented in section 3.2.3. The technique works under the assumption that the geometrical model of the focus and the context rendering is coherent and not deformed.

Figure 3.5.1: *Example of a single color lens. The focus area appears normal while the context is blurred after sepia color transformation.*

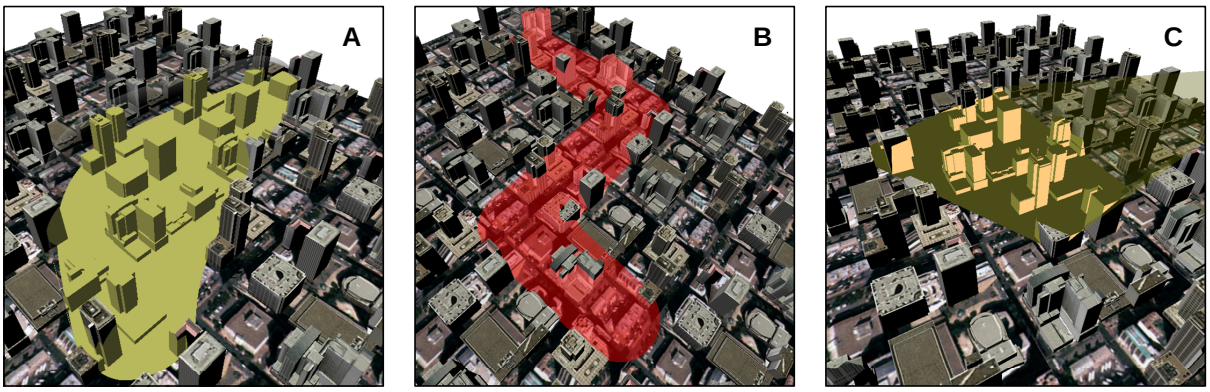


Figure 3.5.3: *Examples of color lenses with different render styles and lens shapes. A: Flat shading scene render style in combination with standard shading and texturing. B: Color overlay post render style. C: Like subfigure A but with inverted flat shading.*

3.5.1 Render Styles

The potential of color lens addresses the selective and associative characteristics of visual variables like color and texture on a high level. Figure 3.5.2 shows the impact of different styles for focus and context rendering. It demonstrates the application of some principles for enhancing preattentive perception after [35]. One can coarsely distinguish between two factors that are able to emphasize or strengthen the visual differences between the lens focus and its context:

- **Shading:** Determines how the scene is shaded and lit. Strong discontinuities in shading or complementary shading colors can be easily perceived by humans []. This addresses mainly the differences between NPR shading styles like cel-, gooch-, and standard shading. Figure 3.5.3 A and C shows two examples.

- **Image Filtering:** These image-based post-processing methods [48] can be applied after shading of the scene geometry. Color allow the encoding of correlations between objects with equal properties. This visual variable can be used to identify and localize buildings or other object in the scene. Figure 3.5.2 show examples of color transformations. Figure 3.5.1 demonstrate the application of *convolution filtering* [24] in order to implement *semantic depth-of-field* (SDOF) [90, 89].

To allow an application to deal with these factors on a high level, the term *render-style* (RS) introduced. A RS is a group of rendering controls for objects. According to the above list we can identify two classes of render styles: *scene render-styles* (SRS) and *post render-styles* (PRS). A *render style configuration* consists of a single scene render-styles and n post render styles: $RSC = (SRS, \{PRS_0, \ldots, PRS_n\}$. Render-styles can be represented and implemented by using uber-shader programs (see section 3.7). This allows various combinations of different RS.

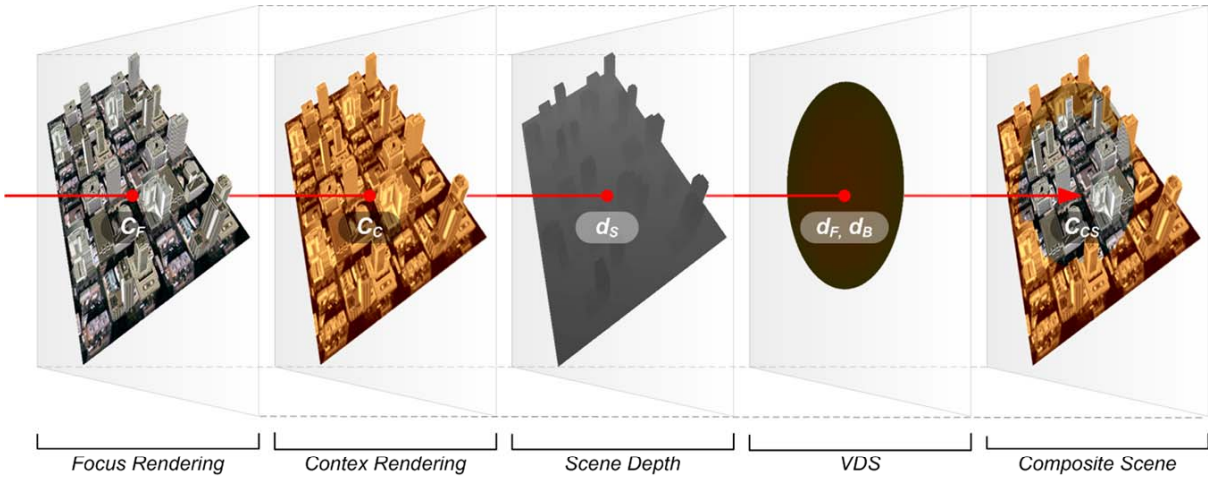


Figure 3.5.4: *Color lens integration process and participating components.*

3.5.2 Lens Model

A color lens allows the association between a *renderstyle configuration* and a *volumetric depth sprite*. Hereby, the configuration is directly associated with the object identity *id* described in section 3.1.2. Consequently, the number of simultaneous available render style configurations is limited to the maximal number of VDS object identities. Following to that, there is a specific render style configuration for the scene. More formally a color lens L_l can be defines as:

$$L_l = (RSC_l, VDS_l) \tag{3.27}$$

The position of the lens is inherent given by the position of the VDS shape.

3.5.3 Rendering of Color Lenses

The rendering of color lenses uses *multi-pass* RTT. It assumes the same camera setup that is used for VDS creation. Given a list of color lenses $\mathcal{L} = L_0, \ldots, L_n$ and the context render-style configuration $RSC_C = (SRS_C, \{PRS_{C_0}, \ldots, PRS_{C_n}\}$, the rendering algorithm for n color lenses can be outlined as follows (compare to figure 3.5.4):

1. **Focus Rendering:** For each lens $L_i \in \mathcal{L}$ do: set the SRS of the lens RSC_i and render the scene into a texture T_i. Usually, this texture has the dimensions of the viewport. After this, apply all PRS_i to T_i. This can be done by texturing a screen align quad with T_i and apply the render-style shaders successively.

2. **Context Rendering:** Set SRS_C and render the scene rendering into a color map T_C and a depth map T_D. Apply all PRS_{C_i} to T_C.

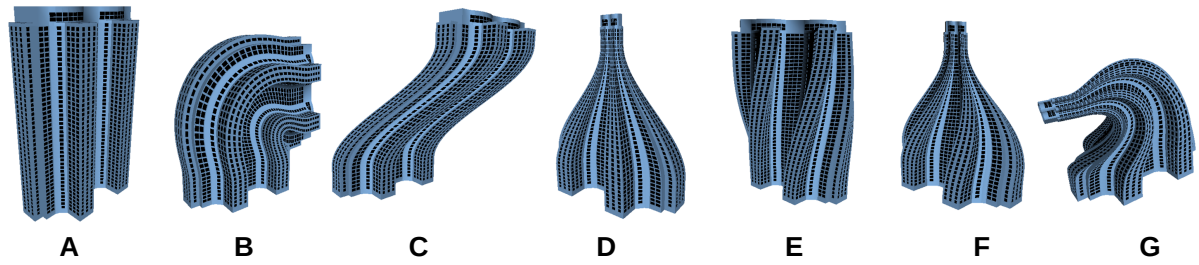


Figure 3.6.1: *Examples of global deformation operators. A: undeformed reference building, B: vertical bending, C: horizontal shearing, D: vertical tapering, E: horizontal twisting, F: combination taper and twist operator, G: combination of taper, twist, and bending operator.*

3. **Compositing:** Finally, integrate all lens rendering T_i and the context rendering T_C into an output image. Figure 3.5.4 demonstrate the integration process for a single color lens L_i. The integration of the context rendering T_C and a focus renderings T_i can be achieved by performing a *volumetric depth test* (see equation 3.10) using the depth map T_D and the particular volumetric depth sprites VDS_i. Consider the scene depth d_s of T_C and the front and back depths d_{F_i}, d_{B_i} of the lens VDS_i. The following equations determine the lens output color C_{CS_i} from the input colors $C_{F_i} \in T_i$ and C_C:

$$C_{CS_i} = \begin{cases} C_{F_i} & , if(\delta_{Inside}(d_S, d_{F_i}, d_{B_i}) = 1 \\ C_C & , otherwise \end{cases} \tag{3.28}$$

This test is performed for all L_i. The resulting C_{CS} will be blend over successively.

3.6 Deformation Lenses

This experimental technique tries to provide additional information about the scene by simultaneously preserve its spatial coherence. This method has mainly two possible applications: resolving occlusions through object deformation and highlighting of user defined space. This approach uses global deformations applied in a vertex shader. This assumes appropriated tessellated shapes. The control of the deformation operations is done by utilizing 2D texture maps. Therefore it utilize *vertex-based decomposition* as described in section 3.2.2.

Global Deformations Generally, space deformations can roughly divided in *axial space deformations*, *surface space deformations*, *lattice space deformations*, and other *specialized space deformations*. A *global deformation* is an axial space deformation that takes a vertex V and outputs a deformed vertex V':

$$\gamma : \mathcal{WCS} \longrightarrow \mathcal{WCS}, \qquad V' = \gamma(V) \tag{3.29}$$

γ is hereby denoted as *global deformation operator* [**?**]. Figure 3.6.1 demonstrate some examples of global deformation operators applied to a single building. Figure 3.6.2 demonstrates the application of global deformations to a more complex model. Deformation operators can be combined by successively applied them to a vertex.

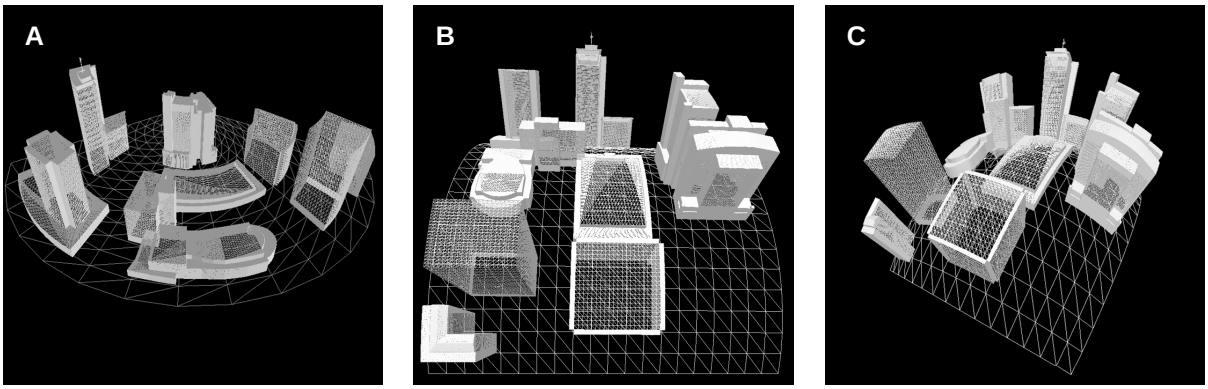


Figure 3.6.2: *Global deformation operations applied to a simple city model. A: Bending around the z-axis, B: bending around the x-axis, C: bending around x- and z-axis.*

Lens Model One can distinguish lenses models on the basis of the coordinate system in which the deformation is applied. Usually, this will be the world- coordinate-system. However, there are possible applications for deformation in camera space $\mathcal{CCS}$. This work concentrates on deformation in world coordinate system $\mathcal{WCS}$. Figure 3.6.3 shows some examples of deformation lenses. The rendering of these can be done within a single pass by using vertex shader. They apply *vertex-based decomposition* to determine the affiliation of a vertex to the context or focus. The lens shape is represented by a monochrome 2D texture. Simultaneously, this texture allows the parametrization of the deformation operator. A deformation lens is a tuple:

$$\mathcal{L} = (\mathcal{P}, T, \gamma) \tag{3.30}$$

$\mathcal{P}$ is a plane described in section 3.2.2. It defines the lens position, dimension, and orientation. T represents the 2D texture that control the deformation operator γ. For an incoming vertex $V \in \mathcal{WCS}$ the model transformation $\mathbf{M}$ is applied to V so that $V_M = \mathbf{M} \cdot V$. Dependent of the implementation, it can be necessary to extract the model matrix from the model-view matrix $\mathbf{MV}$ by calculating $\mathbf{M} = \mathbf{MV} \cdot V^{-1}$. Considering a parametrized deformation operator γ the projected vertex V' can then be calculated as follows:

$$V' = \mathbf{VP} \cdot \Big(\gamma\big(\mathbf{T}(V_M)^{-1}\big) \cdot \mathbf{T}(V_M)\Big) \tag{3.31}$$

First, V_M is translated into the coordinate origin. The result represent the input of the deformation operator γ. Afterwards, the deformed vertex is translated back. Finally, the deformed vertex is multiplied with view-projection matrix $\mathbf{VP}$

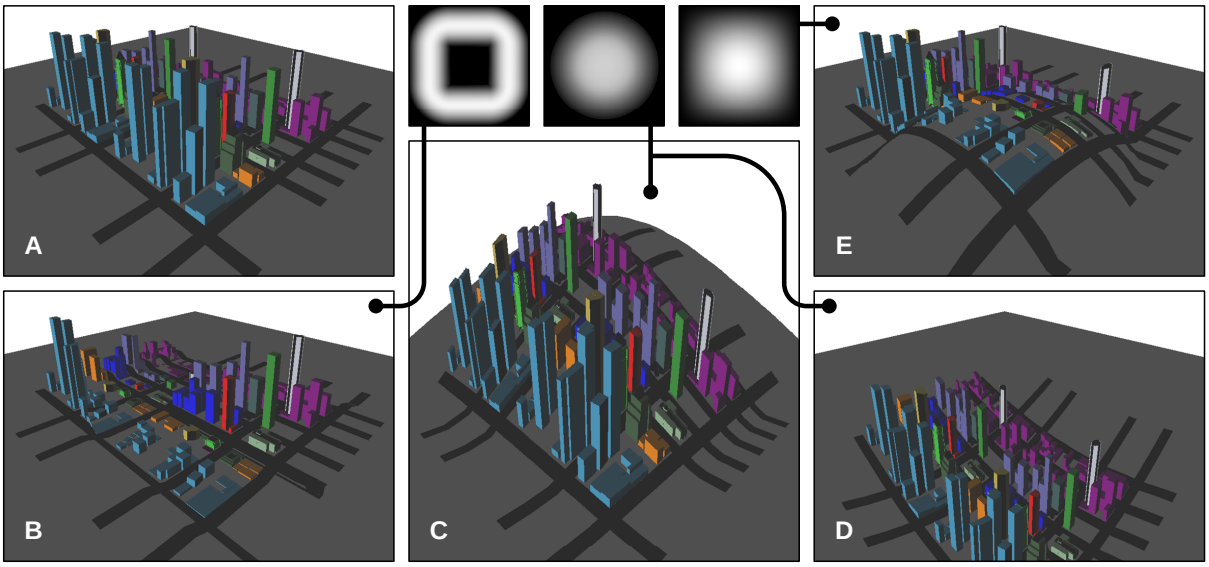


Figure 3.6.3: *Example for global deformations in world coordinates.*

3.7 Shader Management

Programmable hardware came along with a main conceptual problem: it is only possible to have a single active shader that replaces parts of the fixed function rendering pipeline and become part of the rendering context [**?**]. This results in multiple independent shaders for multiple variations of rendering. Consider an encapsulated functionality of a shader, which is integrated into a scene-graph based high-level graphic API like VRS or OpenSG. The combination of such shaders requires the work of an engineer that is able to develop a new shader which functionality is a conglomerate of several features. This aspect is an antagonism to the generic characteristics of such an API. The aim is to achieve logical decoupled functionality and implementation.

Problem Statement To achieve decoupled functionality within a scenegraph based rendering platform, one have to solve the following two problems [34]:

1. **Permutation Problem:** In current engines or frameworks many shaders are variations and combinations of basic functionality (e.g. material LOD approximations, lighting models, animation skinning etc.). It can be expected that the total number of combinations will increase in the future. Creating and managing these permutations would be time consuming, error prone, and hard to maintain.

2. **Independence Problem:** By paying respect to the increasing general propose GPU computation trend, shader move away from tasks like lighting and animation to the more general and complex applications. Many modern shader-driven engines [5] or frameworks utilize the concept of a shader library [1]. To enable a generic solution one have to ensure that multiple instances of shader permutations can interact and perform independent from each other.

It can be assumed that hardware restrictions like the limitation of shader instructions and constant/varying registers will decrease in the future. This will lead to growing complexity and size per shader. Since the introduction of shader model 3.0, GPU programs support instructions for flow control (loops and branching).

Static vs. Dynamic Branching Todays hardware support two different types of branching: *static* and *dynamic branching* [49].
With dynamic branching, the comparison condition resides in a variable. The comparison is done for each vertex or each pixel at run time (as opposed to the comparison occurring at compile time, or between two draw calls). The performance hit is the cost of the branch plus the cost of the instructions on the side of the branch taken.
Static branching denotes the capability of a shader model that allows blocks of code to be switched on or off based on a boolean shader constant. This is a convenient method for enabling or disabling code paths based on the type of object currently being rendered. Between draw calls, one can decide which features have to support with the current shader and then set the boolean flags required to achieve that behavior. Any statements that are disabled by a boolean constant are skipped during the execution of the shader.

Generic Uber-Shader The concept of *generic uber-shaders* (US) represents the technical backbone of the *color lens* and *deformation lens* implementation. The basic idea is a generic approach for uber-shaders, denoted as *dynamic uber-shader construction*. Besides low-level approaches like *micro-shader* and *shader-fragments* which use script-languages in pre-processing [34], this approach is able to solve the permutation and independence problems. An US is a single monolithic and independent shader for multiple geometry using *static* or *dynamic branching* to control the execution of the particular code paths. McGuire [74] demonstrated this by creating an uber-shader that is able to render several effects. The so called *SuperShader* allows arbitrary combinations of rendering effects to be applied to surfaces simultaneously. It uses *run-time code generation* to produce optimized shaders for each surface and a cache to re-use shaders from similar surfaces.
The presented method uses static branching because dynamic branching is currently available for vertex shader only. It is applicable to high-level shading languages like GLSL. Furthermore, it acts on the assumptions that no special shader compiler and no special syntax is necessary. The integration and concatenation of the shader source code is done by an *shader-management- system* (SMS). It combines several *shader-handler* (SH), grouped by *uber-shader programs* into a single US controlled by the SMS.

Handler Concept Automatic source code generation solves the permutation problem. To achieve a generic conglomerate of independent functionality, the shaders are split into functional components: *vertex shader-handler* (VSH) and *fragment shader-handler* (FSH). These components deal for example with animation, transformation, and lighting. The handlers are an quadruple of:

$$VSH_{id_V} = (id_V, name, mode, source), \quad FSH_{id_F} = (id_F, name, mode, source) \tag{3.32}$$

The identifiers id_V, id_F are unique properties of the shader handler. The *name* denotes the functionality and is important for the automatic combination of shader handler. The *source* attribute contains the GLSL shader source code that implements the particular

functionality. Finally, $mode \in \{Local, Global, Optional, Ignore\}$ defines the execution mode of a handler. The following execution modes can be distinguished:

- **Global:** The handler will always be invoked during the execution of the uber-shader. Examples can be: clipping, fog or writing to multiple render-targets.
- **Local:** The handler will only be invoked when it is set to active. The interpolation for the generic mesh refinement described in section 2.3 is a local handler.
- **Optional:** The handler will only be invoke of no handler of its prototype was invoked before.
- **Ignore:** The shader handler will never be invoked.

The set of all available unique vertex shader hander is denoted as $\mathcal{VSH}$. The set of all fragment shader hander is denoted as $\mathcal{FSH}$. These handler can communicate using a predefined interface, i.e. by reading and writing a context which encapsulate the output variables of the particular shader type. This made it possible to access the results of the previous handler to save calculation costs. A vertex context can encapsulate the position, normal, point size, and clip vertex coordinates. A fragment context can store the fragment output targets and the fragment depth (see section 4.4.2 for details). There are two special handler for each shader type. The *init* handler sets up the particular context at the beginning of a shader. The *finish* handler set the particular shader output state to the results of the respective vertex or fragment context. All VSH and FSH will be integrated into a single uber-shader which consist of one vertex shader and one fragment shader. The integration is controlled by the uber-shader system. VSH and FSH can be grouped in uber-shader programs. These programs control the invocation of the particularly shader handlers. An uber-shader program $\mathcal{USP}_i$ consists of the following components:

$$\begin{aligned} \mathcal{USP}_i &= (\mathcal{VSH}_i, \mathcal{FSH}_i, VHIT_i, FHIT_i), \quad \mathcal{VSH}_i \subseteq \mathcal{VSH}, \mathcal{FSH}_i \subseteq \mathcal{FSH} \\ VHIT_i &= \{id_{V_x} | \forall x : VSH_{id_{V_x}} \in \mathcal{VSH}_i\} \\ VFIT_i &= \{id_{F_x} | \forall x : FSH_{id_{F_x}} \in \mathcal{FSH}_i\} \end{aligned} \tag{3.33}$$

The global uber-shader $\mathcal{US}$ is described as:

$$\mathcal{US} = (\mathcal{VSH}, \mathcal{FSH}, VHHT, FHHT) \tag{3.34}$$

The global order of SH execution is controlled by the *vertex-handler hook-table* ($VHHT$) and *fragment-handler hook-table* ($FHHT$):

$$VHHT = id_{V_0}, \ldots, id_{V_a}, \quad FHHT = id_{F_0}, \ldots, id_{F_b} \tag{3.35}$$

with $VSH_{id_{V_i}} \in \mathcal{VSH}$ and $FSH_{id_{F_j}} \in \mathcal{FSH}$. The length of the global hook-tables is given by: $a = |\mathcal{VSH}|$ and $b = |\mathcal{FSH}|$. These tables are generated by the application. This process is transparent for the developer. To determine the order of shader-handler within the $VHHT$ and $FHHT$, the SMS manages an ordered list of so called *prototype handlers*:

$$\mathcal{VHP} = VHP_0, \ldots, VHP_s \qquad \mathcal{FHP} = FHP_0, \ldots, FHP_t \tag{3.36}$$

These prototypes consists of the *name* and the default *execution mode*: $VHP = (name, mode)$, $FHP = (name, mode)$. An instance of VSH or FSH is associated with a particular prototype handler by its *name*. So, these global list of vertex- ($\mathcal{VHP}$) and fragment-handler prototypes ($\mathcal{FHP}$) inherent represents the execution order of the shader handlers with the same type within a uber-shader program. The SMS specifies a number of vertex and fragment prototype handler a priori.
The $VHHT$ and $FHHT$ are configured by an corresponding *vertex-handler invoker-table* ($VHIT_i$) and *fragment-handler invoker-table* ($FHIT_i$) for each $\mathcal{USP}_i$. These invoker-table arrays represent the boolean state for the main vertex and fragment shader that implement the $VHHT$ and $FHHT$. They denote the particular shader handler of an $\mathcal{USP}_i$, which are active during its execution.

Chapter 4

Implementation of 3D Lenses

4.1 Development Environment

The implementation of the 3D information lenses presented in the previous chapter is done by using the C++, the VRS class library with an OpenGL 2.0 binding. The implementation makes some general assumptions on the graphic hardware. It requires a GPU that supports shader model 3.0 or fulfill GLSL 1.10 specification. The GPU programming is done with several non-vendor specific ARB extensions [72].

General Design Decisions The Virtual Rendering System (VRS) [42], is an object-oriented 3D computer graphics library. It provides building blocks for composing 3D scenes, animating 3D objects, and interacting with 3D objects. It serves as a framework and testbed for application-specific or experimental rendering, animation, and interaction techniques. Figure 4.1.2 shows the class diagram of the basic static structure which is important to understand the VRS integration of the lens techniques. The next sections describe shortly this set of base classes.
Among others, VRS distinguishes between geometry (`Shape`), its attribution (`Attribute`) and rendering algorithms (`Shader`, `Technique`). Inherited objects of the `Attribute` type encapsulate graphical attributes that can be stored in contexts under a given category. VRS differentiates between two types of attributes. Only one class instance of the `MonoAttribute` type can be active simultaneously while multiple instances of the `PolyAttribute` type can be active at the same time. Thus, poly-attribute objects allow to represent sets of attributes (e.g light sources). The subclasses of the `Shape` base class rep-

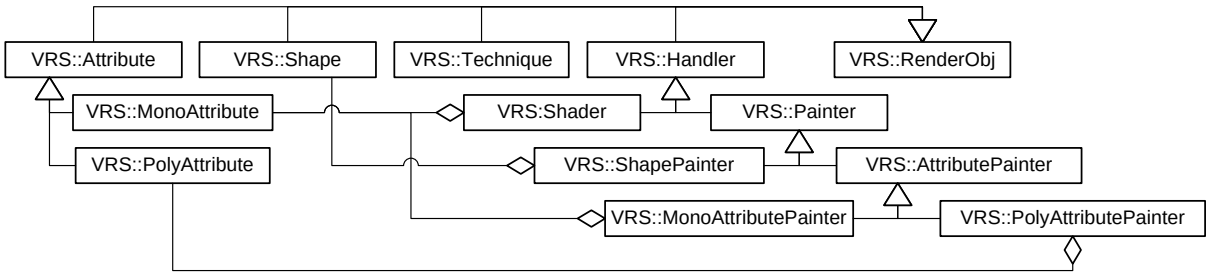


Figure 4.1.1: *Part of the VRS class hierarchy.*

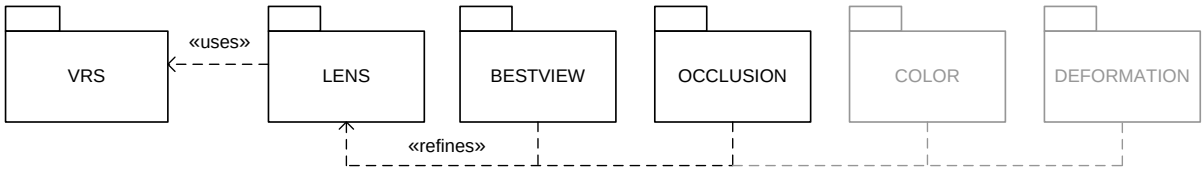


Figure 4.1.2: *Package hierarchy of the lens framework.*

resent renderable geometry. The abstract base class `Handler` is an interface that provides services to engines. The base classes `MonoAttributePainter`, `PolyAttributePainter` and `ShapePainter` are used to evaluate attributes and shapes while the `Shader` class encapsulates *multipass shading and rendering* of a subscenegraph. The interface `Technique` enables combinations of *multipass rendering strategies* for the whole scene graph.
Due to the design rules of the VRS API, most of the object compositions are implemented as weak aggregations [32] using the `SO<>` *smart-pointer* template class. Further, programming rules for effective class implementations are considered [77, 75].

Naming Conventions The implementation applies the usual naming conventions for VRS. Namespaces as well as constants are written in uppercase letters. Each word of a class name begins with a large letter. Names are chosen to be significant. The class member variables are designated with an underline at the end.

3D Information Lenses Figure 4.1.2 shows the logical architecture model [32] of the implementation into four subpackages that are integrated into the main package `LENS`. The base class `LENS::Lens` inherits the `VRS::PolyAttribute` class in order to enable multiple lens instances. This aims basically at the *best-view lens* and *color lens* subsystem. The base class encapsulates the properties described in section 3. During the evaluation of the poly-attributes, a corresponding `VRS::PolyAttributePainter` handler will be invoked that registers the particular lens instance to a corresponding *lens manager*. Each lens type possesses its own manager class which is implemented by using the *singleton pattern* [20]. In general, the implementation does not consider the rendering of the terrain model. It assumes that the terrain is rendered before applying the lens techniques. This section focuses on the implementation of *best-view lenses* and *occlusion lenses* which represents the main results of this work. The development of *color* and *deformation lenses* is a *proof-of-concept* and does not result in a stable software architecture.

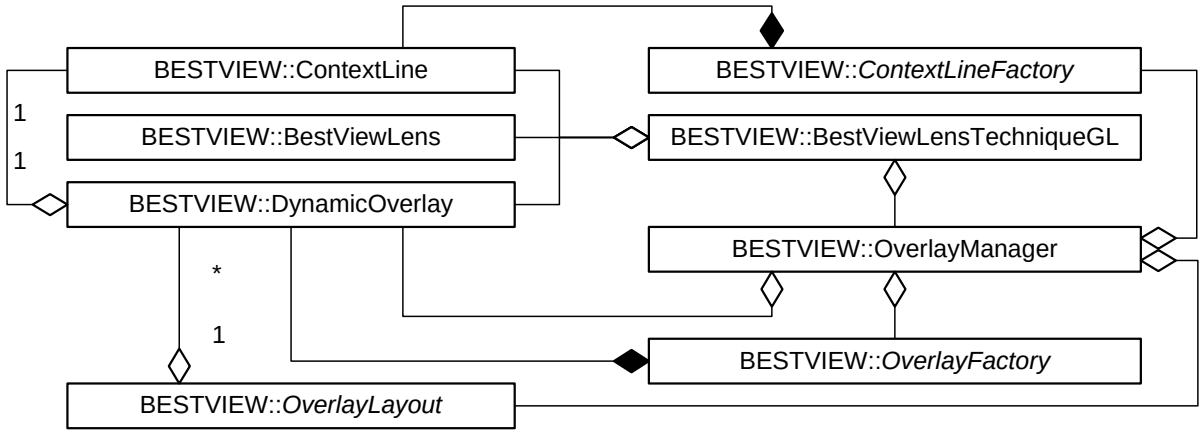


Figure 4.2.1: *Main view on the static architecture of the BVL framework.*

4.2 Best-View Lens

This section covers the static architecture model of the BVL implementation. All classes are embedded in the `LENS::BESTVIEW` namespace. Due to the design of different lens types, this subsystem is of high complexity. First, the main classes, interfaces, and their cooperation are introduced (see figure 4.2.1). Further, the classes that extend these interfaces will be described in detail.

4.2.1 Main Classes and Interfaces

Figure 4.2.1 presents the class structure of the BVL framework and considers only the main classes. Table 4.4.1 gives a short explanation and overview of the underlying design principle. The central class `OverlayManager` coordinates the cooperation between the interfaces and the rendering technique (`BestViewLensTechniqueGL`). It is possible to extend the framework with new layouts and other types of overlays or context-lines. The `OverlayManager` manages ordered lists of context lines (see section 3.4.3). A specific BVL will be registered to the manager during the traversal of the scenegraph. This process is invoked by the particular BVL painter. Figure 4.2.2 shows a corresponding sequence diagram. The manager controls the creation of dynamic overlays and its corresponding context line by using the overlay and context line factories. After the registration of all lenses, the rendering technique initiates the layout algorithm for the overlays by delegating this task to an registered instance of an `OverlayLayout` subclass. Dependent on the lens type, the technique initiates *offscreen multi-pass rendering* to create the overlay annotation data. Afterwards, it applies all context-line constraints before rendering them (see section 4.2.5). Finally, the rendering of all overlays will be invoked. A token in form of a `BestViewLensEvaluation` attribute, which contains a reference to the current evaluated lens, is pushed before every offscreen rendering pass. This enables a binding of a user-defined `VRS::MonoAattributePainter` or a rendering techniques to each lens.

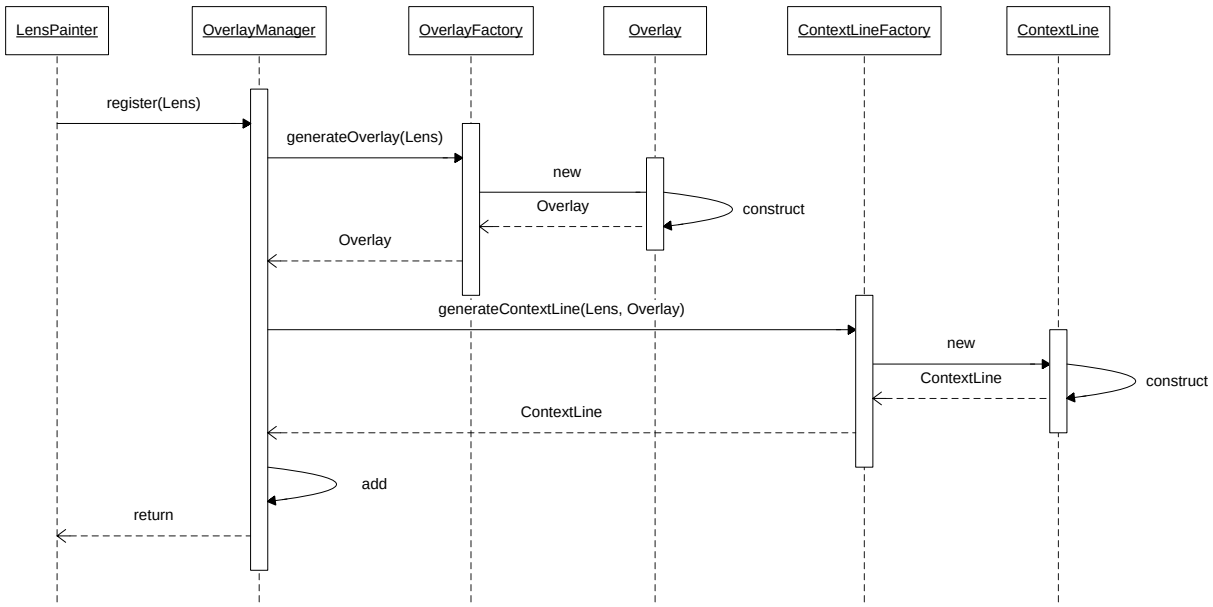


Figure 4.2.2: *Sequence diagram for lens registration.*

4.2.2 Best-View Lens Types

The `BestViewLens` class extends the `LENS::Lens` base class with a boolean `hasContextLine` and a color attribute. Additionally, it forms the base class for the different best-view lenses. Figure 4.2.3 reflects the BVL taxonomy presented in section 3.4. The `StaticBestViewLens` subclass encapsulates basically a 2D texture that contains the annotation data provided by the user. Thus, the creation of complex annotation data can be assigned to other systems or components. The `DynamicBestViewLens` class and its child classes manages a local scene camera. This `VRS::Camera` is used for the dynamic creation of the overlay content which is initiated by the `BestViewLensTechniqueGL` class. Therefore, the camera scope will be set to `Camera::LOCAL`. The subclasses `SCOPBestViewLens`, `MCOPBestViewLens`, and `MapViewLens` apply the orientation transformations described in section 3.4.1 by overloading the `getCamera()` method.

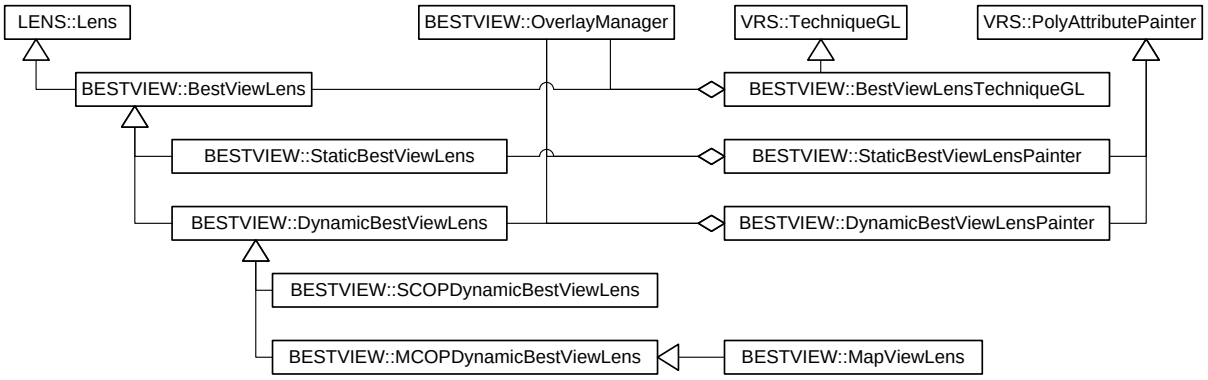


Figure 4.2.3: *Inheritance hierarchy of the different BVLs and the integration into the VRS API.*

Class	Function
`BestViewLens`	Represents the BVL base class.
`OverlayManager`	Encapsulates BVL resource management and application logic.
`DynamicOverlay`	Represents a non-abstract overlay interface that encapsulates mainly a 2D texture.
`OverlayFactory`	Provides an abstract interface to the overlay manager that creates and configures dynamic overlays.
`OverlayLayout`	Provides an abstract interface to the manager which subclasses enable the implementation of different layout strategies.
`ContextLine`	Represents the graphical association of a BVL and an dynamic overlay.
`ContextLineFactory`	Provides an interface for the creation and configuration of context-lines.
`BestViewLensTechniqueGL`	Controls the rendering process of BVLs, overlays and context lines, as well as the application of layout strategies and context line constraints.

Table 4.2.1: *Functionality of BVL classes and interfaces.*

4.2.3 Dynamic Overlays

In order to support a wide range of overlay-types with different appearances and functionalities, the `DynamicOverlay` class provides an interface to the `OverlayManager`. The class diagram in figure 4.2.4 shows the provided overlay types that are necessary for the particular BVL types. Each overlay type will be instantiated by its corresponding factory. Therefore, a potential factory has to subclass the `OverlayFactory` interface and to overload the `generateOverlay()` function. It is invoked after registering a BVL to the overlay manager (see figure 4.2.2). The factory must be registered to the manager before. The `DynamicBoxOverlay` provides a simple non-transparent overlay while the `DynamicAlphaMaskOverlay` class implements the overlay appearances shown in section 3.4.1 and 3.4.3. The `MapViewOverlay` subclass extends the features of this class by redefining the overlay rendering to support the different *map-view modes* described in section 3.4.1.

4.2.4 Overlay Layout

The abstract base class `OverlayLayout` represents the overlay layout interface to the `OverlayManager`. The main tasks of this class are the calculation of all overlay dimensions and the management of *context anchors* (CA) of the particular context-lines (see section 3.4.3). These features are implemented in subclasses by overloading the respective functions `updateLayout()` and `updateContextLineAnchor()`. The calculation of the overlay dimensions depends on the current viewport size and can be parametrized according to minimal, maximal, and preferred size constraints which are encapsulated by the `OverlaySizeConstraint` class. The class diagram in figure 4.2.5 shows the two supported overlay types `VerticalOverlayLayout` and `HorizontalOverlayLayout`. These

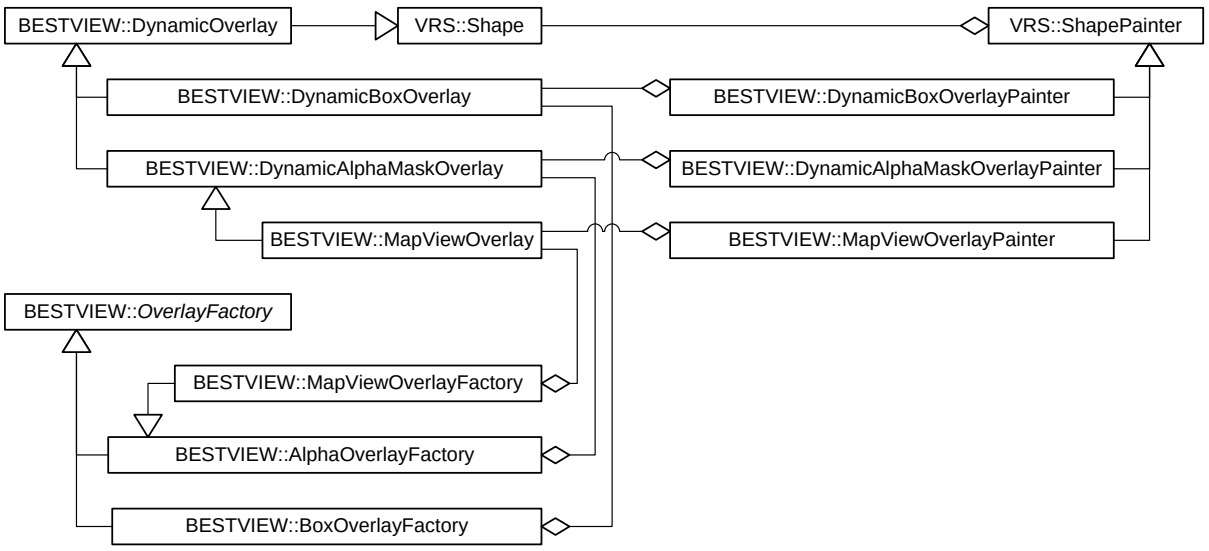


Figure 4.2.4: *Different characteristics of dynamic overlays and their embedding into the VRS framework.*

classes support left, right, top, and bottom vertical and horizontal overlay aligns as well as parameters for the overlay margins.

4.2.5 Context Lines

The association class `ContextLine` implements a *binary association* between an overlay and a BVL. It also represents the base class for different types of context-lines. The implementation currently supports a straight context line `StraightContextLine` (as described in section 3.4.3). The instantiation and initialization of a context line type is handled by the `StraightContextLineFactory`. The appearance can be modified by using the `StraightContextLineStyle` class. The rendering of a straight context-line is controlled by the `StraightContextLinePainter`. The behavior of context-lines can be manipulated via *constraints* (see section 3.4.3). Figure 4.2.7 shows the embedding of the `ContextLineConstraint` class. Subclasses can implement specific functionality by overloading the `applyConstraint()` function. Two constraints are currently available: The `ContextLineVisibilityConstraint` hides the particular context line if the focus-anchor (FA) is outside the viewport area. The `ContextLineOcclusionConstraint` fades the

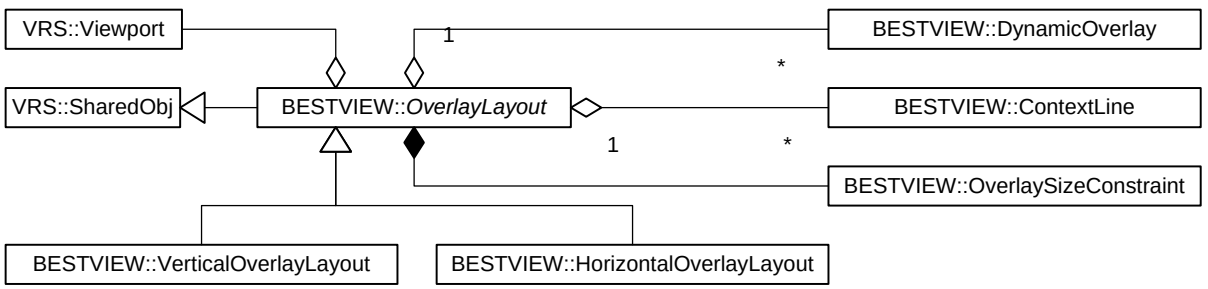


Figure 4.2.5: *Horizontal and vertical overlay layouts.*

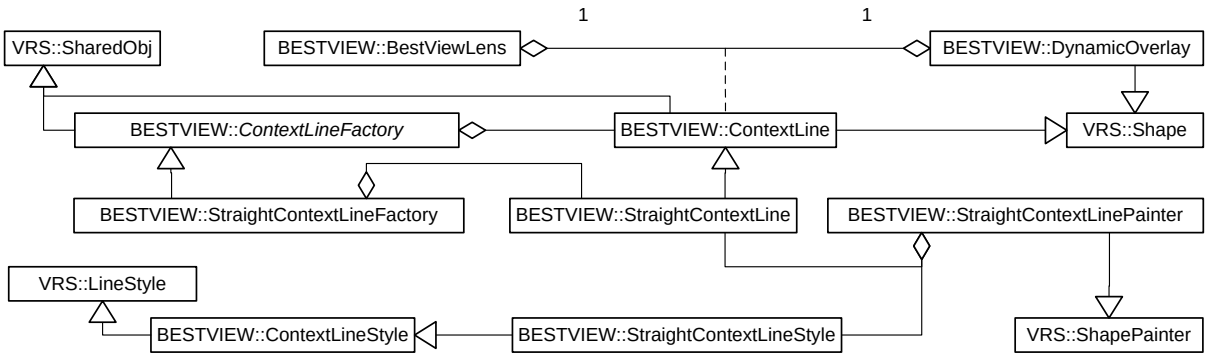


Figure 4.2.6: *Embedding of the context-line association class into the BVL and VRS framework*

transparency of the context line if it occludes an designated area on the viewport (see figure 3.4.8).

4.3 Occlusion Lenses

The implementation classes of the *3D occlusion lenses* are embedded in the `OCCLUSION` namespace. It uses VRS shader because the conception requires a multiple evaluation of a scene graph (see section 3.3). A VRS shader is a handler that provides a service to the rendering engine and encapsulates *multi-pass shading and rendering* of a sub-scenegraph. Since VRS shader can only provide services for mono attributes, it becomes necessary to lever the inheritance hierarchy that was designed for multiple instances of lenses. This can be acceptable under the circumstance of the application (permitted for a single center-of-projection usage). The integration into VRS is divided according to the concept represented in section 3.3.

4.3.1 Intra-Object Occlusion Lenses

The *intra-object occlusion lenses* are implemented as a subclass of a general *depth peeling* shader (compare to section 2.3). It uses the VDS data structure which implementation which is described in section 3.1. Usually, depth peeling is implemented by using a modified *shadow mapping* approach (in combination with a dot-product depth-replace texture shader [72]) to achieve a second depth test [8]. Since *high-precision textures* and programmable GPUs are available, this depth test can be done in a fragment shader

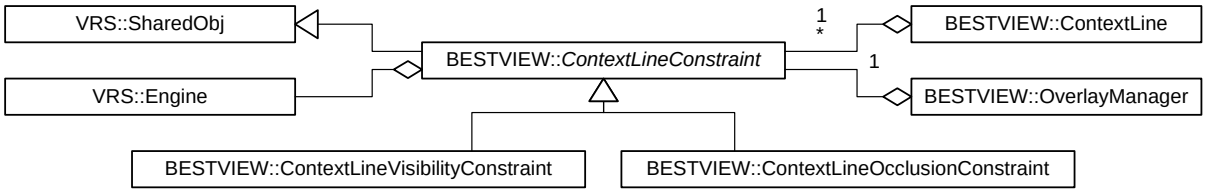


Figure 4.2.7: *The static structure of context-line constraints which are supported by the implementation.*

program [69]. Listing 4.3.1 shows the shader source code that accomplishes this depth test.

```
uniform sampler2D depthMap;
uniform float viewportWidth;
uniform float viewportHeight;
uniform float pass;
varying float depthInCamera;

void main(void)
{
    float z = depthInCamera;
    if(pass > 0.0)
    {
        float w = gl_FragCoord.x / viewportWidth;
        float h = gl_FragCoord.y / viewportHeight;
        // Perform first depth test with depth map
        if(z <= texture2D(depthMap, vec2(w, h)).x) discard;
    }//endif
    gl_FragData[0] = gl_Color;
    gl_FragData[1] = vec4(z, z, z, 1.0);
    return;
}
```

Listing 4.3.1: *Fragment shader for depth-peeling technique.*

The `DepthPeelingShader` creates color and depth maps according to the number of layers using *offscreen rendering*, *framebuffer objects* (`VRS::FrameBufferObjectGL`), and *and multiple-render targets* (`GL2::DrawBuffers`). These 2D textures can be accessed and processed by a particular subclass. The postprocessing and integration of the results into the previous rendered scene can be done by texturing a screen-aligned quad. Alternatively, this shader could be implemented by *ping-pong rendering* as described in section 2.3 (if no depth maps are needed). The behavior of the `DepthPeeling` shader can be controlled by various parameter combinations of texture width, height, and the number of peeling layers. If the developer does not limit the number of layers, the `DepthPeelingShader` employs an occlusion query [72] to determine the number of necessary peeling passes. A general *order-independent transparency* shader for VRS (`OrderIndependentTransparency`) is an additional result of this implementation approach.

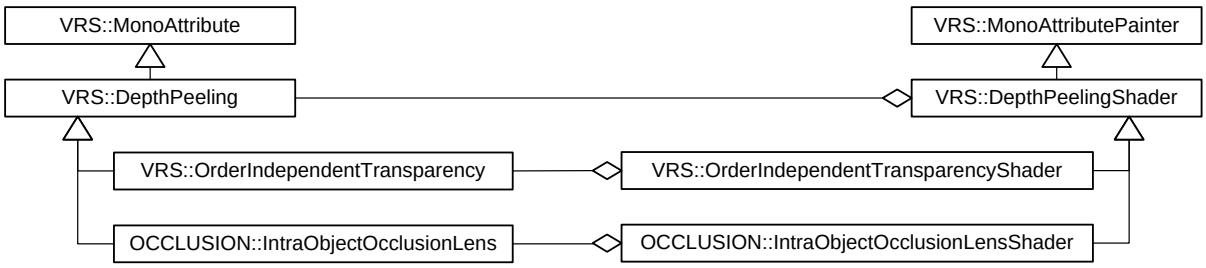


Figure 4.3.1: *Static class structure for the intra-object occlusion lens.*

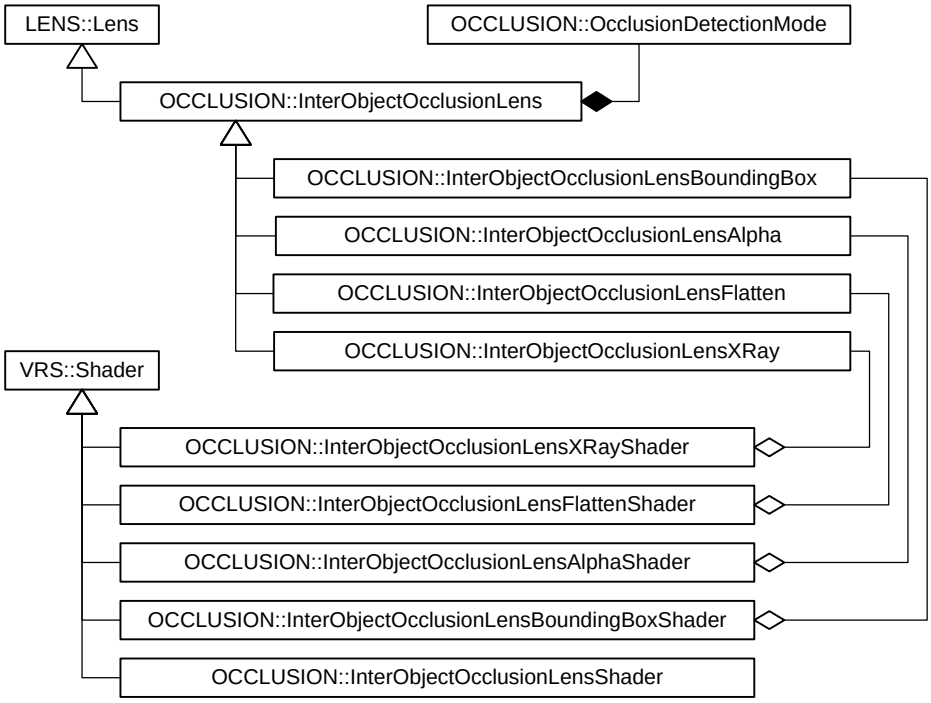


Figure 4.3.2: *Inter-object occlusion lens classes and shader.*

4.3.2 Inter-Object Occlusion Lenses

Figure 4.3.2 shows the attributes and the corresponding shader of the *inter-object occlusion lens* implementation. Except for the transparency occlusion approach, the shaders need two rendering passes. The `InterObjectOcclusionLensBoundingBoxShader` class renders the bounding box of an occluder shape instead of its geometry. The line style can be controlled by a `VRS::LineStyle` attribute. The flatten shader (`InterObjectOcclusionLensFlatten` performs the flattening operation as described in section 3.3.4. A standard shader (`InterObjectOcclusi` omits the rendering of an occluder shape. The `InterObjectOcclusionLensXRayShader` applies the visual abstraction as described in section 3.3.3. Finally, the transparency shader (`InterObjectOcclusionLensAlphaShader`) implements a reduced *depth peeling* algorithm using the *ping-pong rendering* technique for the second depth test (see section 2.3). This technique needs to perform five passes. The first pass renders the occludees and the remaining four passes [8] accomplish *order-independent transparency*.

4.3.3 Occlusion Detection Test

The *inter-object occlusion detection tests* are implemented as subclasses of the abstract base class `OcclusionDetectionMode`. These classes have to overload the *pure virtual function* `occlusion()` to apply their particular functionality. Instances of these tests are part of a specific `InterObjectionOcclusionLens`. The `occlusion()` function is called by the `eval()` method of a particular inter-object occlusion lens shader. A boolean return value categorizes the evaluated shape as occluder (`true`) or occludee (`false`). The class diagram in figure 4.3.3 shows the embedding and the provided occlusion detection tests as described in section 3.3.1. This modality allows the extension of the framework with more advanced detection tests.

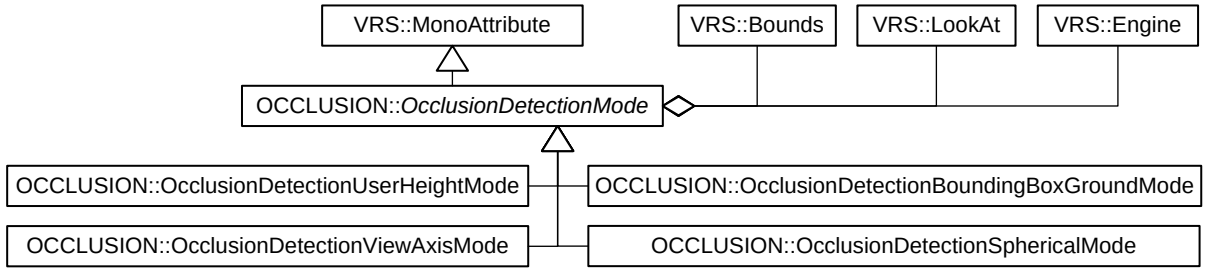


Figure 4.3.3: *Implementation and embedding of different occlusion detection tests.*

4.4 Selected Representations

This section covers different elementary and essential implementation issues. It focuses on methods and technologies that are contributed to the VRS API or essentially used by the specific lens implementations.

4.4.1 Volumetric Depth Sprites

The implementation of *volumetric depth sprites* as introduced in section 3.1 is part of the lens framework and therefore located in the `LENS` package. Figure 4.4.1 describes the static structure of the participated classes. Two approaches for the creation of a VSD are evolved in the implementation. The `SimpleVolumetricDepthSprite` inherits the `VRS::Shape` base class and enables the encoding of standard VRS shapes into a VDS. It encapsulates a 2D texture and an integer object ID. The corresponding painter implements the creation process described in section 3.1.2. The `ComplexVolumetricDepthSprite` attribute enables the transformation of sub-scene graph contents into a VDS by configuration and invocation of the `ComplexVolumetricDepthSpriteShader`. The shape displayed in figure 3.1.2 is of such a complex kind. The `VolumetricDepthSpriteContext` class enables a global VDS configuration concerning the precision (see section 3.1.3) as well as managing the object identities. It is possible to combine different VDS. The combination of two object IDs is expressed as their sum. Due to the lack of bitwise operators in GLSL [50], a workaround has to be developed to invert the identity mapping function γ in equation 3.2. Listing 4.4.1 describes a part of the fragment shader that is able to test if a particular object ID is a summand of a combined object ID.

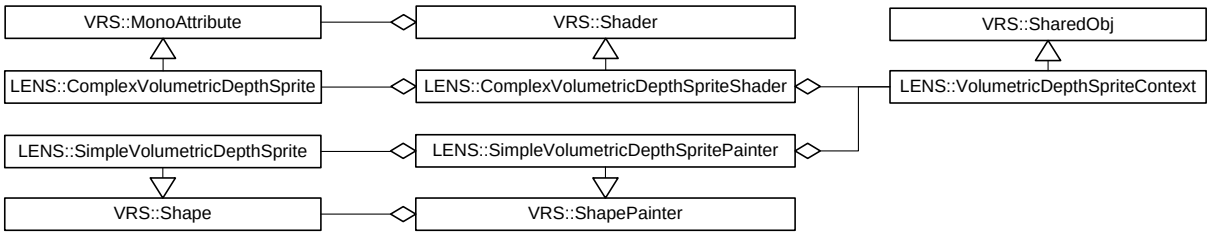


Figure 4.4.1: *Implementation of volumetric depth sprites.*

```
float encodeID(float lensID, float maxID)
{
    return pow(2.0f, lensID) / pow(2.0f, maxID+1);
}

bool testID(float encodedIDs, float ID, float maxID)
{
    float test = encodedIDs; float remainder;
    for(float i = maxID; i > ID; i--)
    {
        float encodedID = encodeID(i, maxID);
        remainder = test - encodedID;
        if(remainder >= 0)
        {
            test = remainder;
        }//endif
    }//endfor
    remainder = test - encodeID(ID, maxID);
    if(remainder >= 0) return true;
    return false;
}
```

Listing 4.4.1: *VDS Identity encoding and decoding*

4.4.2 Shader Management

For standard shader applications the `VRS::GL2` shader API is re-used. These shader sources are linked statically into the particular classes. The VRS uber-shader API is embedded in the `GL2::EXTSHADER` namespace. Figure 4.4.2 shows its static structure and table 4.4.1 summarizes the functions of the participating classes. Additional thereto, the classes `UberShaderUniformVariable` and `UberSchaderSamplerVariable` have the same functionality like their `GL2` pendants. The usage of US is also analog to `GL2` shaders.

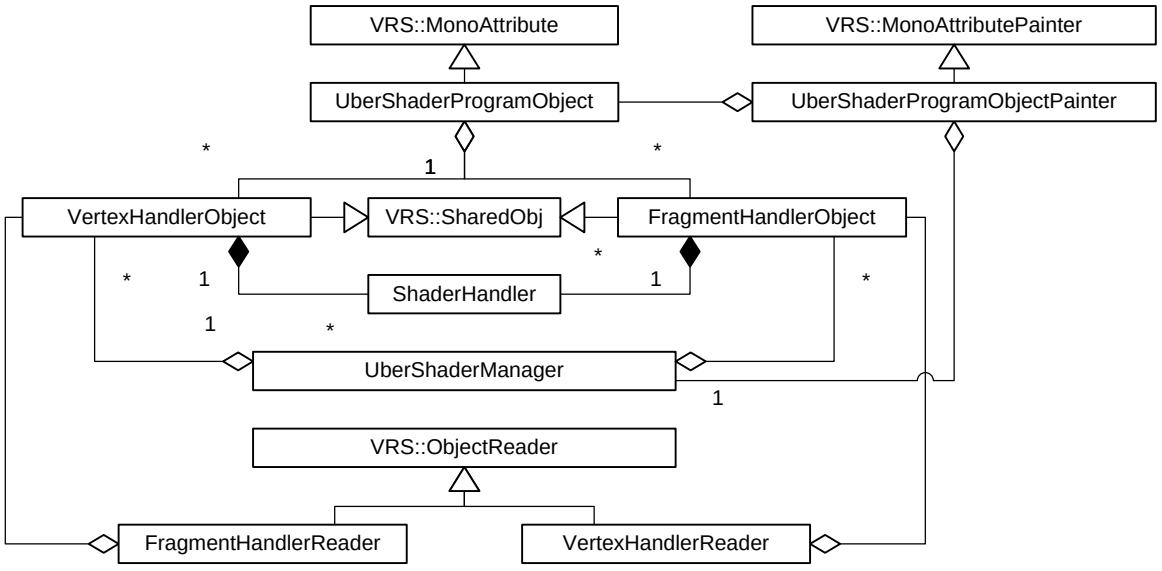


Figure 4.4.2: *Architecture of the uber-shader system.*

Class	Function
`UberShaderManager`	Encapsulates the program logic and manages resources.
`VertexHandlerObject`	Represents a vertex shader that could contain one or more vertex shader handler.
`FragmentHandlerObject`	Represents a fragment shader that could contain one or more fragment shader handler.
`UberShaderProgram`	Represents a single uber-shader and consists of multiple vertex- and fragment-hander objects.
`PrototypeHandler`	Defines a prototype of a shader handler. A prototype consists of a unique name and a default handler mode.
`ShaderHandler`	Is an concrete instance of a prototype handler and encapsulates a single vertex or fragment handler.

Table 4.4.1: *Uber-shader classes and their function*

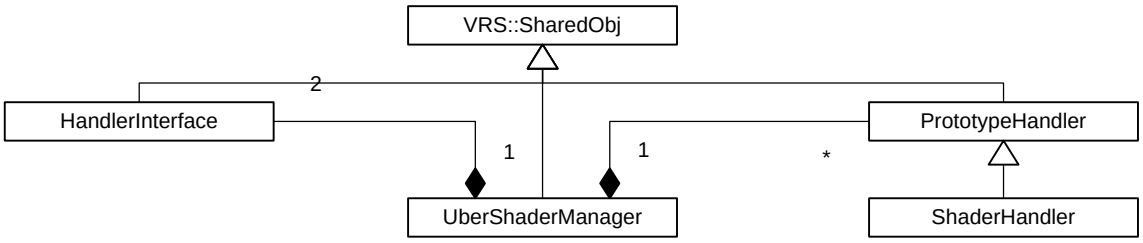


Figure 4.4.3: *Implementation of the handler concept*

Instances of `VertexHandlerObject` and `FragmentHandlerObject` are attached to an instance of an `UberShaderProgram`. To enable the design of a custom uber-shader framework, the communication between the particular handlers of the same type can be defined by using the `HandlerInterface` class. It encapsulates the program state that is necessary for data exchange between successive shader-handlers. A possible complex vertex hander interface is shown in listing 4.4.2. To define a custom handler interface, the full interface code block, the interface type name as well as the instance name, have to be specified.

```
struct us_VertContext //interface type name
{
  vec4  us_Position;  //interface code block
  vec3  us_Normal;
  float us_PointSize;
  vec4  us_ClipVertex;

} VertContext;        //instance name
```

Listing 4.4.2: *Vertex Handler Context*

Figure 4.4.3 shows the static structure of the handler concept as described in section 3.7. This concept is necessary for the automatic generation of the global vertex-handler hook-table (VHHT) and fragment-handler hook-table (FHHT) respectively. The `UberShaderManager` encapsulates the preprocessing functions and contains ordered lists of vertex and fragment shader handler prototypes (`PrototypeHandler`). By creating an instance of a `VertexHandlerObject` or a `FragmentHandlerObject`, the object will be registered and

the shader source code will be parsed by the `UberShaderManager`. According to the registered VHP and FHP, the manager tries to extract handlers of a specific type from the shader source and determines their execution modes. If an handler of a specific prototype is found, an instance of `ShaderHandler` will be created that represents the particular function in the shader source. During this process, the handler names in the source code will be qualified with an unique name using string manipulation. After this, the particular shader-handler-object represents a list of `ShaderHandler`. The creation of the global uber-shader is done by the `UberShaderManager` during the first evaluation of the scene graph. For all registered `VertexHandlerObject` and `FragmentHandlerObject` instances a corresponding `GL2` shader object will be created and attached to a main `GL2` shader program. Finally, the source code of the global VHHT and FHHT will be generated and attached. For each prototype handler, all shader handlers of the registered VHO and FHO will be analyzed. If an particular shader handler has the same type (has the same name) as the prototype, its qualified handler name is gathered in the hook-table. Listing 4.4.2 shows the vertex handler hook table generated by the `UberShaderManager` class.
If an `UberShaderProgramObject` is evaluated, the `UberShaderManager` retrieves the list of vertex and fragment shader-handler of the evaluated uber-shader program and modify the boolean state of the particular invoker table.

```
uniform bool vertexHandlerInvokerTable[3];

void shadertest2global1Bvert2onTransform(inout us_VertContext);
void shadertest2global1Avert1onLighting(inout us_VertContext);
void shadertest2global1Bvert3onFinish(inout us_VertContext);

void hookTableVertex(void)
{
  if(vertexHandlerInvokerTable[1])
    shadertest2global1Bvert2onTransform(VertContext);
  if(vertexHandlerInvokerTable[0])
    shadertest2global1Avert1onLighting(VertContext);
  if(vertexHandlerInvokerTable[2])
    shadertest2global1Bvert3onFinish(VertContext);
}
```

Listing 4.4.2: *Example of a VHHT that contains three vertex shader-handler.*

4.4.3 Generic Mesh Refinement

Figure 4.4.4 presents the embedding of a *generic mesh refinement* approach (see section 2.3) into VRS. The implementation consists mainly of the combination of a *refinement pattern*, a *refinement shape*, and a *LOD mapping*. Due to the possibility of using triangles and quads as refinement patterns, the abstract base class `RefinementPatternGL` is introduced. The `TriangleRefinementPatternGL` subclass represents a triangular refinement pattern for a specific sub-division level. It uses a `VRS::MappedVertexAttributeGL` for the representation in the GPU memory. This is the VRS encapsulation of a vertex buffer object (VBO) [72]. The generation of vertices according to a sub-division level is done in the class constructor.
With regards to new generations of GPUs or other refinement primitives, the abstract

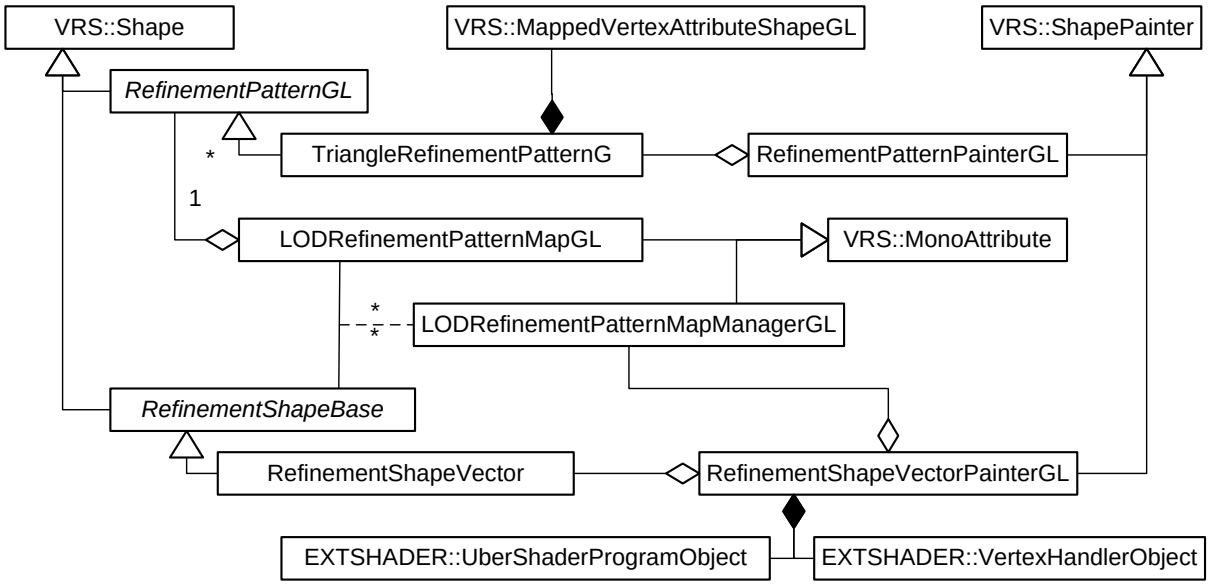


Figure 4.4.4: *Mesh refinement classes and embedding.*

base class `RefinementShapeBase` was introduced. Analog to the `VRS::PolygonSet`, the `RefinementShapeVector` subclass encapsulates a set of triangles with their corresponding color, normal, and texture coordinate attributes. The `RefinementPatternMapGL` represents an $1 - n$ mapping of a `RefinementShapeGL` and a particular distance. The LOD support is managed by the `RefinementPatternMapManagerGL` association class that associates a `RefinementPatternMapGL` with a specific `RefinementShapeGL`. It delivers a refinement pattern according to the distance between a given bounding box of the refinement shape and a user defined reference point $P \in \mathcal{WCS}$. The manager is invoked by `RefinementShapeVectorPainterGL` during the evaluation of a `RefinementShapeVector`. The painter transforms the vertex attributes into an array of type `std::vector<VertexData<4,float>>` and sets it as global shader state by using the `EXTSHADER::UberShaderUniformVariable` class (see section 4.4.2). After that it determines the adequate refinement pattern and activates the rendering of this pattern. The implementation uses an `UberShaderProgram` with a corresponding `VertexHandlerObject` to perform the attribute interpolation described in section 2.3 (see listing 4.4.3).

```
uniform vec4 attr[12];

void onInit(inout us_VertContext context)
{
  #define V gl_Vertex;
  vec4 v0 = attr[0]; vec4 v1 = attr[1]; vec4 v2 = attr[2];
  vec4 c0 = attr[3]; vec4 c1 = attr[4]; vec4 c2 = attr[5];
  vec4 t0 = attr[9]; vec4 t1 = attr[10]; vec4 t2 = attr[11];
  vec3 n0 = attr[6].xyz;
  vec3 n1 = attr[7].xyz;
  vec3 n2 = attr[8].xyz;
  gl_FrontColor =  (V.x * c0) + (V.y * c1) + (V.z * c2);
  gl_TexCoord[0] = (V.x * t0) + (V.y * t1) + (V.z * t2);
  context.us_Normal = (V.x * n0) + (V.y * n1) + (V.z * n2);
```

```
15      context.us_Position = (V.x * v0) + (V.y * v1) + (V.z * v2);

        return;
    }

20  #define ONFINISH optional
    void onFinish(in us_VertContext context)
    {
        gl_Position = context.us_Position;
        return;
25  };
```

Listing 4.4.3: *Vertex handler object for generic mesh refinement.*

4.4.4 Multiple Render Targets

The implementation of 3D information lenses uses MTRs to create color and depth maps for focus or context renderings within one pass. The usage of this feature requires a hardware that supports framebuffer objects (FBO). MRTs have consequences for the output variables of the fragment shader [50]. Instead of writing to `gl_FragColor`, the user has to use `gl_FragData[]` variable. The implementation in VRS is straight forward. Figure 4.4.5 shows the class diagram for the integration. The class `DrawBuffers`, which part of `GL2` namespace encapsulates an array of color buffer handler (`VRS::Array<GLenum>`). The `DrawBuffersPainter` class applies the buffer setting in each pass by performing a call to the `glDrawBuffers()` function. The FBO render-targets must have the same size and internal format.

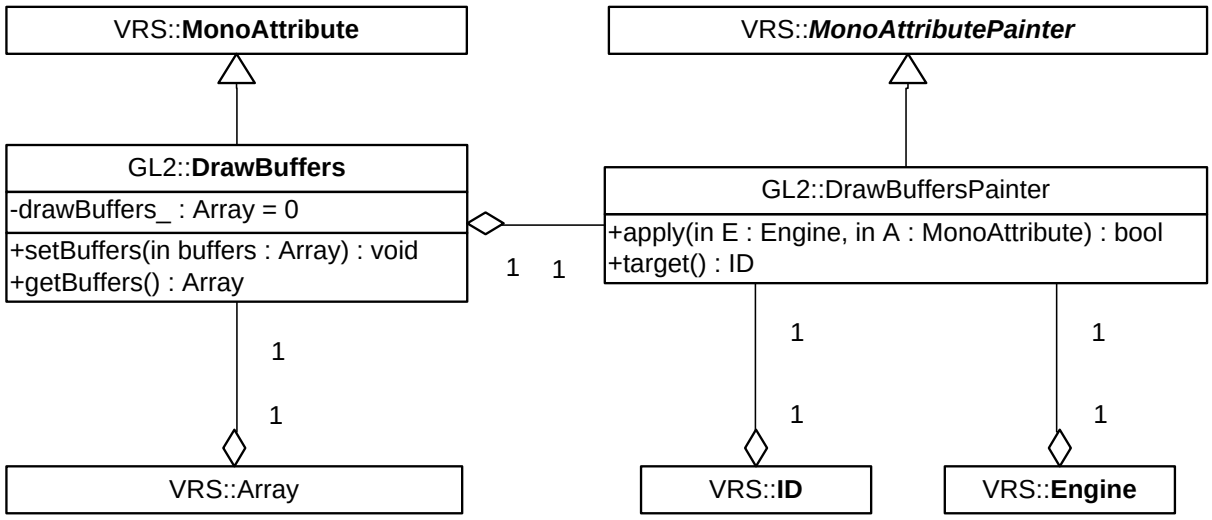


Figure 4.4.5: *Class diagram for integration of multiple render targets.*

Chapter 5

Analyse & Discussion

5.1 Performance

This section gives an overview of the runtime behavior of the presented lens techniques. An accurate performance test of the software components is disclaimed since no particular optimizations of the solutions are achieved. For *best-view*, *occlusion*, and *color lens techniques*, the runtime complexity T will be approximated and discussed. In principle, the performance of the presented solutions depends on CPU and, GPU speed, number of scene objects, and number of vertices per object. Table 5.1.1 gives the mapping of the main used symbols. In principle, the values of T_F, T_C, and T_{VDS} depend on the number of

Symbol	Denotation
T_F	Costs for focus rendering.
T_C	Costs for context rendering.
T_I	Integration costs for focus and/or context rendering.
T_{VDS}	Creation costs for a single VDS.

Table 5.1.1: *Symbols for runtime approximations.*

vertices of the scene and the lens shape, as well as the viewport dimensions $w, h \in \mathcal{SCS}$. Since a VDS is represented as a single 2D RGBA texture, its space complexity amounts into $S_{VDS} = whC$ bit for C bits per pixel (bpp). The quality, especially of the contour, of a VDS depends on the tesselation level of the shape. The implementation was tested by using the dataset described in table 5.1.2.

Dataset	Vertices	Faces	Originator
Maybeck Studio	3388	6596	K. Matthews & Artifice, Inc.
San Diego	29184	25920	Planet 9 Studios
Vancouver	2910	9220	Wizard Solutions Inc.
Kopenhagen	272219	464771	Kobenhavens Kommune
Griebnitzsee	45534	92063	Hasso-Plattner Institute

Table 5.1.2: *Test datasets for the implementation.*

Occlusion Lenses The depth-peeling technique, which is utilized to implement *intra-object occlusion lens* possesses a high space and runtime complexity. The performance of this technique depends on the number of peeling-layers that are necessary to achieve sufficient visual quality for *order-independent transparency*. The number of required layers depends on the *depth complexity* of the model. The depth complexity denotes the maximal number of overlapping polygons in a scene. A scene with n polygons can have a maximal depth complexity of $O(n^2)$. For a given model with a depth complexity of n the runtime can be approximated with:

$$T = n \cdot (T_F + T_I) + T_{VDS} + T_C \tag{5.1}$$

The runtime for one depth peeling pass is denoted as T_F while T_I is the runtime for the integration of n peeling layer in postprocessing. The detailed model of the *Maybeck Studio*, which was used to create the figures 3.3.1 and 3.2.3, possesses a depth complexity of 28. Using an occlusion query to terminate peeling, usually results in a non-interactive rendering speed. Although, the number of layers that deliver sufficient visual results are smaller than the maximal depth complexity. The corresponding space complexity in bits for the viewport dimensions w, h, the color range C and the depth range D bit per pixel (bpp) can be calculated by:

$$S = n \cdot \big((w \cdot h) \cdot (C + D)\big) + S_{VDS} \tag{5.2}$$

It is possible to compensate S by using compressed textures, lower texture precision, and smaller texture sizes. This leads to a decrease of image quality for the benefit of improved runtime speed. Furthermore, using n MRTs will reduce the fillrate to $1/n$. This is caused by the limitation of the memory bandwidth. Thus, reading and writing of the data is the bottle neck. To compensate the fillrate limitation one can choose a lower range for color C and depth maps D. However, the usage of fixed-point or 16bit floating-point textures is not sufficient enough for the depth test (see section 3.1.3). It causes unwanted artifacts near or far away form the COP. Thus, the usage of MTR should be avoided in applications that use extensively *multiple pass rendering* (like depth peeling). Alongside this, if the number of peeling passes is known, instances of FBOs should be created in advance to avoid unnecessary OpengGL state changes.
The performance of *inter-object occlusion lenses* depends on the *granularity* of the city model. If each building can be represented by a shape with a single AABB the low granularity is low. Vice versa, the granularity is high if the buildings is described by more than one AABB. The latter case will produce visual artifacts. For each shape of the scene the membership to the *focus* or *context area* has to be determined by applying an *occlusion detection test*. A runtime approximation for the rendering of an *inter-object occlusion lens* with respect to the number of shapes n can be given trough:

$$T = 2 \cdot (T_O \cdot n) + (T_C \cdot n) + (T_F \cdot n) \tag{5.3}$$

T_O is the runtime of a specific *occlusion detection test*. It is CPU bound for 3D city model with a high granularity. When applying more complex visual abstractions which require multiple-pass rendering on large amounts of city model objects, it is recommended to cache these objects according to their categorization (*occludee* or *occluder*). In contradiction to *intra-object occlusion lenses*, the *inter-object occlusion lenses* posses a low space complexity because they do not store any texture maps for later compositing.

Best-View Lenses The runtime performance of *static* and *dynamic best-view lenses* depends mainly on the number of lenses n and context-lines m. Except for *map-view lenses*, it usually applies $n = m$. An approximation can be given by:

$$T = n \cdot T_F + n \cdot T_O + T_L + m \cdot (T_{CL} + T_{CLC}) \tag{5.4}$$

T_O denotes the costs for integrating a dynamic overlay to the viewport, T_L the runtime of the layout algorithm, T_{CL} expense for rendering a context-line, and T_{CLC} context-line constraints. T_L and T_{CLC} are CPU bound for large number of overlays and context-lines. To reduce the costs of T_F which is GPU bounded by the fillrate, it is recommend to set the width and height of the target texture to the exact overlay dimension.

Color Lenses The performance of color lenses is conceptually limited by the number of lenses and the number of post render-styles per lens. The runtime T, required to render n color lenses, can be estimated by:

$$T = n \cdot T_{VDS} + n \cdot T_F + T_C + T_I \tag{5.5}$$

T_F represents the runtime that is necessary to create the focus rendering. It is composed of the runtime for scene RTT and the rendering of a particular number of post render-styles. Color lenses draw their benefit from *volumetric depth sprites* and the corresponding *volumetric depth test*. This concept enables a low runtime for each processed fragment by taking a high space complexity into account. Assuming a single VDS per lens, the space complexity S for a number of color lenses n can be appraised by:

$$S = n \cdot S_{VDS} + n \cdot S_F + S_C \tag{5.6}$$

The space complexity of the particular focus and context rendering is $S_F = S_C = (w \cdot h) \cdot C$. Here, the scene depth is encoded in the alpha channel. The usage of high-precision textures in combination with a linearized depth buffer (see section 3.1.3) for the volumetric depth test proves to be correct. Figure 5.1.1 shows the comparison of different depth ranges for a spherical VDS. The subfigure A uses 8bit range in combination with a non-linear depth calculation. It causes artifacts near as well as far from the COP. In contradiction thereto, 16bit floating-point textures already deliver sufficient visual results (see sub-figure B).

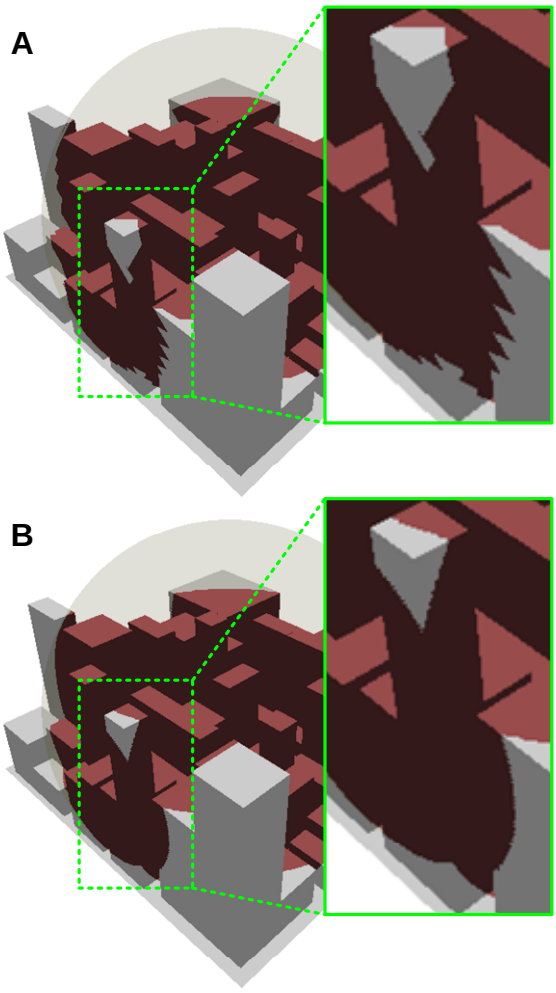


Figure 5.1.1: *Comparison of fixed-point (A) and floating-point (B) depth ranges.*

Deformation Lenses The performance of deformations lenses is mainly limited by the generic mesh refinement approach which is used to refine the geometry before a deformation. The following observations were made: The mesh refinement is CPU limited for the number of triangles that have to be refined and GPU limited for level of subdivision per triangle. The current implementation stalls at nearly 20,000 triangles on a subdivision level of 30. Further, the LOD approach for refinement-pattern selection is too costly. A general subdivison level between 10-15 is sufficient in most cases.
Finally, this approach is not applicable to large numbers of triangles, i.e. to large 3D city models. The usage of geometry shaders [57] for complete triangle subdivison on GPU will overcome this limitation.

5.2 Limitations

Some of the introduced 3D lens concepts posses conceptual and technical limitations which are described in this section. Most of this limitations concern the GPU. Since all of the concepts of this thesis are developed under heavy usage of programmable GPUs, no fallback code can be granted or incorporated. Due to the hardware architecture, reading from and writing to the same texture is not possible simultaneously. Such a feature would be useful to reduce the number of rendering passes for VDS creation or the depth peeling techniques.
The accessibility of raster data in the vertex shader is limited to four texture units and there is no support for filtering of high-precision textures. This can be handled by implementing bilinear filtering, using multi sampling (MS) [85](see listing 5.2).

```
uniform float texWidth;
uniform float texHeight;

vec4 texture2DBiLinear(sampler2D sampler, vec2 st )
{
    float stepW = 1.0 / texWidth;
    float stepH = 1.0 / texHeight;
    float fs = fract(st.s * texWidth);
    float ft = fract(st.t * texHeight);

    vec4 s0 = texture2D(sampler, st);
    vec4 s1 = texture2D(sampler, st + vec2(stepW, 0.0));
    vec4 s2 = texture2D(sampler, st + vec2(0.0, stepH));
    vec4 s3 = texture2D(sampler, st + vec2(stepW, stepH));

    vec4 sA = mix(s0, s1, fs);
    vec4 sB = mix(s2, s3, fs);

    return mix(sA, sB, ft);
}
```

Listing 5.2: *VTF and MS for bi-linear filtering in a vertex shader*

The VTF is still slow, but the texture look-up speed will increase prospectively. Further, GLSL is not well implemented by current drivers. This addresses mainly the uber-shader implementation and leads to minor problems if the shader-handler wants to modify the

global GLSL shader state. The current driver does not fully compile shader objects until the linkage of the program object. Currently, all the shader object sources for a single program are concatenated and then compiled and linked. This means, there is no efficiency from compiling shader objects once and linking them in multiple program objects.
The lack of the availability for conditional returns in shader programs results often in a performance loss. Conditional returns can be used to skip instructions prematurely.
The application of best-view lenses is limited by the available screen size. A *trade-off* between the overlay dimension and the occluded parts of the scene can be observed. With an increasing overlay dimension the number of displayable BVL decreases while the occluded scene area increases simultaneously. The perceived information in the overlay decreases with the dimension of the overlay area. Furthermore, the graphical representation of the straight context-line is suboptimal. The texture filtering causes artifacts when it is bulged too much.

5.3 Future Work

Since most of the lens techniques are proof-of-concepts there is still room for optimizations in order to grant their optimal performance. The application and integration of *dynamic occlusion*, *view-frustum*, and *back-face culling* is an open issue as well. The mesh refinement approach can be enhanced by using geometry shader [95]. Besides several technical ameliorations of the implementation, there are many possibilities to enrich the functionality of the introduced lens types. The theoretical and technical concepts can be extended in order to support *continuous degree-of-interest* (DOI). Currently, the research of combination of the developed lens-techniques and the interdependence of navigation and lens interaction is the most important issue. Further, the development of LOD lenses could be a possible aim for future work. The methods of LOD lenses are able to represent the fundamentals for applications like *semantic zooming* for city models. Lens techniques are in demand of suitable interaction concepts which are able to operate in combination with the different existing navigation approaches. The next sections present an overview of possible enhancements for the particular lens techniques.

5.3.1 Occlusion Lens

Concerning the *inter-object occlusion lenses*, the development of more accurate *occlusion detection algorithms* can be a topic for further research. Therefore, additional meta information, like building type, adjacencies, or statistical values as well as raster data can be incorporated. It is also advantageous to compare the introduced 3D occlusion lenses with their 2D screen-aligned pendants (compare figure 3.2.3 and 3.3.1) in order to gain information regarding the differences of effects on the user and the lens interaction. Alongside this, the answer to the question, what rendering techniques can enhance the perception of the occludees in combination with the presented occlusion lens methods, could be of interest. Luft [107] introduced an approach for image enhancement by unsharp masking the depth buffer that could be applied to the occludees.

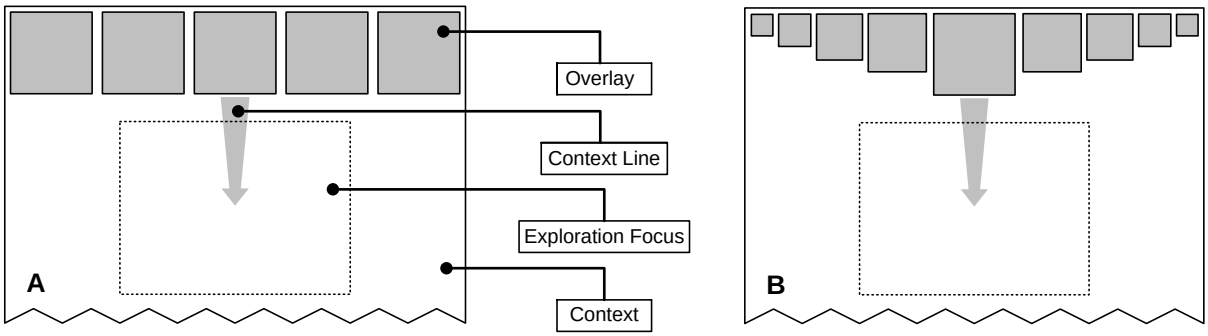


Figure 5.3.2: *Orientation- and distortion-based overlay layouts. A: The overlay which is associated with the POI in the exploration focus is horizontally centered. B: Increase screen-real-estate by using discrete non-linear resizing of the overlay areas.*

5.3.2 Best-View Lens

Best-view lenses should reference areas and not just lens positions. Figure 5.3.2 depicts some possible improvements to compensate the lack of viewport space by applying *discrete distortions* [65] to the overlays. This can also be utilized to resize the overlay in dependency of the distance between its associated location and the viewers position. It could be useful to research smart dynamic positioning of overlays in order to void occlusions with lens POIs. Also, the integration of non-linear distortion of the overlay content may be of interest (see figure 5.3.1).

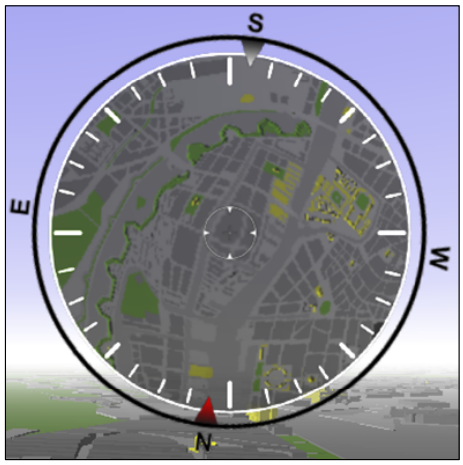


Figure 5.3.1: *Map-view lens with applied non-linear distortions.*

The concept of best-view lenses can be coupled with automatic identification of landmarks in 3D city models [28]. The identified landmark can be pre-processed and organized in a database. The *map-view* lens can be enriched by implementing a lazy position tracking that minimizes the movement of the context. Further, it could be auspicious to add LOD functionality to context-lines. If the distance between the viewer and the POI passes a threshold, the representation of the context line could swap into an arrow that indicates the direction of POI. Additionally, one can think of non-straight context lines represented by curves. They may be able to solve the occlusion problem for context lines.

5.3.3 Color Lens

The main task concerning color lenses is the research of render style configurations and their combinations. Possibly, the combination of *photorealistic* and *non-photorealistic rendering* (e.g. sketchy drawings, blue print rendering, and edge-enhancement) discloses advantages for focus and context visualization. Studies that identify and define possible semantics could be conducted. A result can be the design and activation of an automated mapping between these semantics and the render-style configurations. A class framework

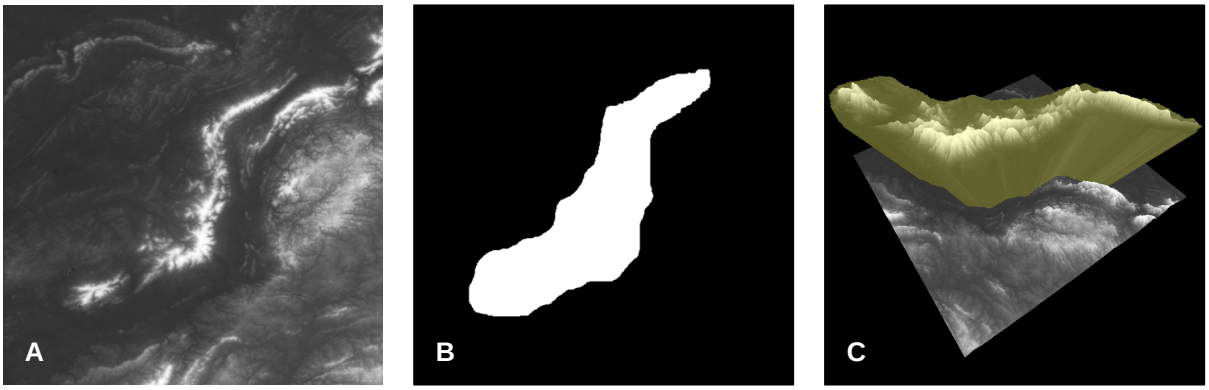


Figure 5.3.3: *Application of deformation lenses for terrain rendering. Creation of an 3D call-out rendering (C) by using mesh refinement in combination with displacement map (A), and a corresponding lens shape (B).*

needs to be developed that allows the combination of rendering styles as well as their mapping to a fixed number of semantics. Unfortunately, a solution for a OO encapsulation and combination of render styles seems hard to achieve. This applies especially to multi-pass techniques. A modern shader driven engine architecture in combination with the uber-shader approach could be able to overcome this limitation. Color lenses draw their benefits from *volumetric depth sprites* (VDS). Admittedly, the interactive manipulation and creation of their shape has to be examined.

5.3.4 Deformation Lens

Some types of global deformations in camera space can be utilized for the creation of *interactive panoramic maps* [98, 7]. This addresses especially the bend-operator. By bending the displayed surface it is possible to achieve the combination of plan and a perspective view. This provides the base to research the relevance of global deformation for focus and context navigation in geovirtual environments. Further, the unification of occlusion detection tests with deformation lenses can be researched. Figure 5.3.3 shows an example for the combination of the translation and scaling operator that are both controlled by textures.

Chapter 6

Conclusions

This thesis presented several *out-of-core techniques* for 3D information lenses and their application to virtual 3D city models. It proved reasonable applications of different lens metaphors and pointed out possible use-cases. These techniques addressed mainly the resolving of object occlusions and facilitated the usage of focus and context visualization. The concept of best-view lenses enables the unification of three different focus & context approaches: *world-lets, through-the-lens metaphor*, as well as *dislocated annotations*. It allows exploring distant locations or hidden features of the city model surrounding the user, without having to move to the remote location.
A class framework was presented that allows the generic creation, the layout, and integration of such lenses. By resolving occlusions between arbitrary objects of a city model, occlusion lenses allow gain of more spatial related information. Different methods for high-level occlusion detection and occlusion resolving were presented. *Color lenses* facilitate the integration of different rendering-styles into one result image. This possibility poses a problem which could be meaningful for a future research: What are the semantics of such a mixture of rendering styles ?
Except for intra-object occlusion lenses, all approaches are capable of real-time rendering. They cover *point-based*, *local*, and *global degrees of user activity*. Some of the presented techniques, such as best-view lenses and deformation lenses, are generally applicable to the geo-spatial domain.
Furthermore, this work introduced *volumetric depth sprites* and the *volumetric depth test* image-based integration technique. This concept has proven to be an efficient way to represent convex geometric lens volumes as raster data. The advantages compared to other approaches [109] lie in its scalability, persistence, and flexibility.
This work included a proposal for a flexible and extensible implementation framework that is able to deal with multiple instances of the particular lens types. In general, the framework extended the VRS API with useful development features like uber-shader, generic depth peeling, and an approach for generic mesh refinement as well. The application of modern programmable graphics hardware allows to enable these concepts *without* any data preprocessing. The presented methodologies and techniques comprehend potential for further research and development. Although each of the proposed techniques has some limitations, the possible combination of them will provide a powerful set of tools.

Bibliography

[1] ATI TECHNOLOGIES INC. Alex Vlachos. Designing a game's shader library for current and next generation hardware. In *GDC Game Developers Conference*, 2002. 2

[2] Andreas Becks, Christian Seeling. Swapit: A multiple views paradigm for exploring associations of texts and structured data. 2004. 3.4

[3] Alan H. Barr. Global and local deformations of solid primitives. *Computer Graphics*, 18(3), July. 3.6

[4] Lyn Bartram. Perceptual and interpratative properties of motion for information visualization. Technical report, School of Computer Science, Simon Fraser University, 1997. 3.5

[5] Steffen Bendel. First thoughts on designing a shader-driven game engine. In Wolfgang Engel, editor, *Direct3D ShaderX: Vertex and Pixel Shader Tips and Tricks*. Wordware, Plano, Texas, 2002. 2

[6] James D. Hollan Benjamin B. Bederson, Larry Stead. Pad++: Advances in multiscale interfaces. In *SIGCHI*, 1994. 3.4.3

[7] ETH Zurich (Switzerland) Bernhard Jenny, Institute of Cartography. Design of a panorama map with plan oblique and spherical projection. In *5th ICA Mountain Cartography Workshop, Bohinj, Slovenia*, 2006. 5.3.4

[8] Cass Everitt. Interactive Order-Independent Transparency. Technical report, NVIDIA OpenGL Applications Engineering, 2001. 2.3, 3.3, 4.3.1, 4.3.2

[9] Tobias Höllerer Chris Coffin. Interactive perspective cut-away views for general 3d scenes. In *IEEE Symposium on 3D User Interfaces*, 2006. 2.2

[10] Andreas Becks Christian Seeling. Analysing associations of textual and relational data with a multiple views system. March 2004. 3.4

[11] Colin Ware. *Information Visualization - Perception for Design*, volume 2nd (Juni 2004) of *Morgan Kaufmann Series*. Morgan Kaufmann, 2nd edition, 2004. 3.4.1

[12] Dan Su, PhilipWillis. Image Interpolation by Pixel Level Data-Dependent Triangulation.

[13] David Baar. Questions of Focus: Advances in Lens-based Visualizations for Intelligence Analysis. 2.2

[14] IDELIX Software Inc. David Baar. Questions of focus: Advances in lens-based visualizations for intelligence analysis. April 2005. 2, 2.1

[15] Dominik Göddeke. Playing Ping Pong with Render-To-Texture. Technical report, University of Dortmund, Germany, 2005. 2.3

[16] Dominik Göddeke, University of Dortmund, Germany. Playing Ping Pong with Render-To-Texture.

[17] Doug Rogers. W-Buffering in Direct3D. Technical report, NVIDIA Corporation, 2000. 3.1.3

[18] T. Todd Elvins, David R. Nadeau, Rina Schul, and David Kirsh. Worldlets: 3D thumbnails for 3D browsing. In *Proceedings of the Conference on Human Factors in Computing Systems CHI'98*, 1998. 2.2, 3.4.1

[19] Ken Pier William Buxton Tony D. DeRose Eric A. Bier, Maureen C. Stone. Toolglass and magic lenses: The see-through interface. In *SIGGRAPH*, pages 73–80. ACM Press, 1993. 2.1

[20] Ralph E. Johnson John Vlissides Erich Gamma, Richard Helm. *Design Patterns. Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995. 4.1

[21] Eugene Lapidous, Guofang Jiao. Optimal Depth Buffer for Low-Cost Graphics Hardware. In *Eurographics*. Trident Microsystems Inc., ACM, 1999. 2.3, 3.1.3

[22] Eugene Lapidous, Guofang Jiao, Jianbo Zhang, Timothy Wilson. Quasi-Linear Depth Buffers With Variable Resolution. In *HWWS*. ViewSpace Technologies Inc., Trident Microsystems Inc., ACM, 2001. 2.3, 3.1.3

[23] Cass Everitt. Projective texture mapping. April 2001. 3.2.2

[24] Foley, van Dam, Feiner, Hughes. *Computer Graphics, Principles and Practise*. Addison Wesley, 1996. 3.1.1, 3.2.1, 3.2.3, 3.4.2, 3.5.1

[25] H. Schumann Universität Rostock G. Fuchs, H. Griethe. Definition allgemeiner linsentechniken auf unterschiedlichen stufen des visualisierungsprozesses. 2006. 3

[26] G. Fuchs, M. Kreuseler H. Schumann. Extended Focus & Context for Visualizing Abstract Data on Maps. In *CODATA Prague Workshop Information Visualization, Presentation, and Design*, 29-31 March 2004. 2.2

[27] Gaël Guennebaud, Loïc Barthe and Mathias Paulin, IRIT - CNRS - Université Paul Sabatier - Toulouse - France. Splat/Mesh Blending, Perspective Rasterization and Transparency for Point-Based Rendering. In M. Botsch, B. Chen, editor, *Eurographics Symposium on Point-Based Graphics*. The Eurographics Association, 2006. 2.3

[28] Iris Galler. Identifikation von landmarks in 3d-stadtmodellen. Master's thesis, Rheinische Friedrich-Wilhelms-Universität Bonn, Landwirtschaftliche Fakultät, Institut für Kartographie und Geoinformation, Oktober 2002. 5.3.2

[29] Alan Dix Geoffrey Ellis, Enrico Bertini. The sampling lens: Making sense of saturated visualisations. In *ACM CHI*, 2005.

[30] Georg Fuchs, University of Rostock, Institute for Computer Graphics. Cartographic Lenses as Focus & Context Techniques. 2.2

[31] Gerhard Gröger, Thomas H. Kolbe und Lutz Plümer. Zur Konsistenz bei der Visualisierung multiskaliger 3D-Stadtmodelle. *Mitteilungen des Bundesamtes für Kartographie und Geodäsie*, 31:Verlag des Bundesamtes für Kartographie und Geodäsie, 2004. 3.3

[32] Ivar Jacobson Grady Booch, Jim Rumbaugh. *Das UML Benutzerhandbuch.* Addison Wesley, 1999. 1.4, 4.1, 4.1

[33] H. Buchholz, J. Döllner, M. Nienhaus, F. Kirsch. Real-Time Non-Photorealistic Rendering of 3D City Models. In *Proceedings of the 1st International Workshop on Next Generation 3D City Models*, June 2005. 3, 3.3.3

[34] Shawn Hargreaves. Generating shaders from hlsl fragments. 3.7, 3.7

[35] Christopher G. Healey. Perception in visualization. 3.5.1

[36] Heidrun Schumann, Matthias Kreuseler, Universität Rostock. Fokus & Kontext-Darstellung im geographischen Kontext. 2.2

[37] Helwig Hauser. An Interaction View on Information Visualization. (based on a STAR of R. Kosara, HH, and D. Gresh, EG 2003). 2.2

[38] Helwig Hauser. Generalizing Focus+Context Visualization. 2.2

[39] Henning Griethe. Einsatz von Linsentechniken für Ikonen über interaktiven Kartendarstellungen. Master's thesis, Universität Rostock, 2004. 1.1

[40] IDELIX® Software Inc. Pliable Display Technology® ¡PDT¿ White Paper v4.0. 2.1, 2.2

[41] J. Döllner. Raumbezogene Informationsvisualisierung mit dynamischen, interaktiven 3D-Karten. *Kartographische Nachrichten*, 51(4):80–85, 2001. 1.1

[42] K. Hinrichs J. Döllner. The virtual rendering system - a toolkit for object-oriented 3d rendering. In *EduGraphics - CompuGraphics Combined Proceedings*, pages 309–318, 1995. 4.1

[43] J. Döllner, H. Buchholz, F. Brodersen, T. Glander, S. Jütterschenke, A. Klimetschek. SmartBuildings – A Concept for Ad-Hoc Creation and Refinement of 3D Building Models. *Proceedings of the 1st International Workshop on Next Generation 3D City Models*, June 2005. 1.1, 3.3

[44] J. Döllner, H. Buchholz, M. Nienhaus, F. Kirsch. J. Döllner, H. Buchholz, M. Nienhaus, F. Kirsch. In *Proceedings of Visualization and Data Analysis 2005 (Electronic Imaging 2005, SPIE Proceedings)*, pages 42–51, 2005. 3

[45] J. Döllner, K. Baumann. Geländetexturen als Mittel für die Präsentation, Exploration und Analyse komplexer räumlicher Informationen in 3D-GIS. *3D-Geoinformationssysteme*, pages 217–230, 2005. 2.2

[46] J. Döllner, K. Baumann, O. Kersting. LandExplorer - Ein System für interaktive 3D-Karten. *Kartographische Schriften*, 7:67–76, 2003. 2.2

[47] J. Döllner, M. Walther. Real-Time Expressive Rendering of City Models. In *Seventh International Conference on Information Visualization, Proceedings IEEE 2003 Information Visualization*, pages 245–250, London, Juli 2003.

[48] ATI Research Jason L. Mitchell. Real-time 3d scene post-processing. In *Games Developer Conference GDC*, 2003. 3.5.1

[49] ATI Research Jason Mitchell. *ShaderX2 - Introduction And Tutorials with DirectX 9.0*, chapter Introduction to the DirectX® 9 High Level Shading Language. Wordware Publishing, 2004. 3.7

[50] John Kessenich Dave Baldwin Randi Rost. *THE OPENGL SHADING LANGUAGE Version 1.10*, 59 edition, April 2004. 3.1.2, 3.3.4, 4.4.1, 4.4.4

[51] John Viega, Matthew J. Conway, George Williams, and Randy Pausch. 3D Magic Lenses. 2.1

[52] Jon Kreuzer, Josh Hess. Line Box Intersection. Januar 2006. 3.3.1

[53] Jürgen Döllner and Henrik Buchholz. Non-Photorealism in 3D Geovirtual Environments. University of Potsdam - Hasso-Plattner-Institute, January 2005. 3.3.3

[54] Catherine Plaisant Kasper Hornbæk, Benjamin B. Bederson. Navigation patterns and usability of overview+detail and zoomable user interfaces for maps. 3.4

[55] CATHERINE PLAISANT KASPER HORNBÆK, BENJAMIN B. BEDERSON. Navigation patterns and usability of zoomable user interfaces with and without an overview. In *ACM Transactions on Computer-Human Interaction*, volume 9, page 362–389., December 2002. 3.4.3

[56] Keith Lau. Perceptual Invariance of Nonlinear Focus+Context Transformations. 2.2

[57] John Kessenich. *The OpenGL® Shading Language Language Version: 1.20 Document Revision: 8*, September 2006. 5.1

[58] Ladan Shams, Christoph von der Malsburg. Acquisition of visual shape primitives. *Vision Research*, 42:2105–2122, 2002. 3.3

[59] Lars Kornelsen. Informationsvisualisierung mit geographischem Bezug. Master's thesis, Universität Rostock, 2002.

[60] Anthony Lovesey. A comparison of real time graphical shading languages. Technical report, Faculty of Computer Science, University of New Brunswick, Canada, March 2005.

[61] Lujin Wang, Ye Zhao, Klaus Mueller, Arie Kaufman (Center for Visual Computing, Computer Science, Stony Brook University). The Magic Volume Lens, An Interactive Focus+Context Technique for Volume Rendering.

[62] M.-A.D. Storey, K.Wong F.D. Fracchia H.A. Müller. On Integrating Visualization Techniques for Effective Software Exploration.

[63] M. Nienhaus, J. Döllner. Sketchy Drawings - A Hardware-Accelerated Approach for Real-Time Non-Photorealistic Rendering. In *ACM SIGGRAPH - Sketches and Applications*, 2003.

[64] M. Nienhaus, J. Döllner. Visualizing Design and Spatial Structure of Ancient Architecture using Blueprint Rendering. In *5th International Symposium on Virtual Reality, Archaeology and Cultural Heritage (VAST 2004)*, 2004. 3

[65] David J. Cowperthwaite M. Sheelagh T. Carpendale and F. David Fracchia. Distortion viewing techniques for 3-dimensional data. 5.3.2

[66] M. Sheelagh T. Carpendale, David J. Cowperthwaite, F. David Fracchia. 3 Dimensional Pliable Surfaces: For the Effective Presentation of Visual Information. 2.2

[67] M. Sheelagh T. Carpendale, David J. Cowperthwaite, F. David Fracchia. Making Distortions Comprehensible. 2.2

[68] M. Sheelagh T. Carpendale, David J. Cowperthwaite, F. David Fracchia. Multi-Scale Viewing. 2.2

[69] MARC NIENHAUS. *REAL-TIME NON-PHOTOREALISTIC RENDERING TECHNIQUES FOR ILLUSTRATING 3D SCENES AND THEIR DYNAMICS*. PhD thesis, UNIVERSITÄT POTSDAM, 2005. 2.3, 2.3, 3.3.3, 4.3.1

[70] Marianne Sheelagh Therese Carpendale. *A Framework for Elastic Presentation Space*. PhD thesis, Simon Fraser University, 1999. 2.2

[71] Joseph L. Gabbard Tobias H. Hoellerer Deborah Hix Simon J. Julier Yohan Baillot Dennis Brown Mark A. Livingston, J. Edward Swan II. Resolving multiple occluded layers in augmented reality. In *International Symposium on Mixed and Augmented Reality (ISMAR)*, 2003. 2.2

[72] Mark J. Kilgard. NVIDIA OpenGL Extension Specifications. Technical report, NVIDIA, November 2006. 2.3, 2.3, 3.1, 4.1, 4.3.1, 4.3.1, 4.4.3

[73] Mark J. Kilgard,. NVIDIA OpenGL Extension Specifications. Technical report, NVIDIA Corporation, May 19, 2004.

[74] Morgan McGuire. *The SuperShader*, chapter 8.1, pages 485–498. 2005. 3.7

[75] Scott Meyers. *More Effective C++: 35 New Ways to Improve Your Programs and Designs*. Professional Computing Series. Addison Wesley, March 1996. 4.1

[76] Scott Meyers. *Effective C++: 50 Specific Ways to Improve Your Programs and Designs*. Addison Wesley, 1998.

[77] Scott Meyers. *Effective C++ : 55 Specific Ways to Improve Your Programs and Designs (3rd Edition)*. Addison-Wesley Professional, May 2005. 4.1

[78] Milan Ikits Charles D. Hansen Scientific Computing and Imaging Institute, University of Utah ? A Focus and Context Interface for Interactive Volume Rendering. 2.2

[79] M.S.T. Carpendale and Catherine Montagnese. A Framework for Unifying Presentation Space. In *ACM User Interface Software and Technology UIST'01,*, 2001. 2.2

[80] Philippas Tsigas Niklas Elmqvist. A taxonomy of 3d occlusion management techniques. Note, December 2006. 2.2

[81] NVIDIA Coorporation. Release Notes for NVIDIA OpenGL Shading Language Support.

[82] R. Scopigno P. Cignoni, C. Montani. Magicsphere: an insight tool for 3d data visualization. *Computer Graphics Forum*, 13(3):317, August 1994. 2.1

[83] Petra Neumann Sheelagh Carpendale. Taxonomy For Discrete Lenses. Technical report, University of Calgary, 2003. 2.2

[84] Son Phan. Focus+context sketching on a pda. Master's thesis, San Francisco State University, San Francisco, California, December 2003. 3.4.1

[85] Philipp Gerasimov, Randima (Randy) Fernando, Simon Green. Shader Model 3.0, Using Vertex Textures, Whitepaper. Technical report, NVIDIA Corporation, 2004. 5.2

[86] Prof. Dr. Jürgen Döllner. Informationsvisualisierung mit dynamischen, interaktiven 3D-Karten. Hasso-Plattner-Institut an der Universität Potsdam.

[87] QuangVinh Nguyen, Mao Lin Huang. A Focus+Context Visualization Technique Using Semi-transparency. In *IEEE Proceedings of the Fourth International Conference on Computer and Information Technology (CIT)*, number 0-7695-2216-5/04, 2004.

[88] Rahul Swaminathan, Michael D. Grossberg, Shree K. Nayar. A Perspective on Distortions.

[89] Robert Kosara, Silvia Miksch, Helwig Hauser. Focus+Context Taken Literally. 2.2, 3.5.1

[90] Robert Kosara, Silvia Miksch, Helwig Hauser. Semantic Depth of Field. 2.2, 3.5.1

[91] Mackinlay Robertson, Card. Information visualization using 3d interactive animation. *Communications of the ACM*, pages 57 – 71, April 1993. 1.1

[92] Bernd Nettelbeck Thomas Strothotte Roland Jesse, Tobias Isenberg. Dynamics by hybrid combination of photorealistic and non-photorealistic rendering styles. Technical Report 5, Department of Simulation and Graphics, Otto-von-Guericke University of Magdeburg, 2004. 3.5

[93] Tobias Isenberg Roland Jesse. Use of hybrid rendering styles for presentation. *Journal of WSCG*, 11(1), February 2003. 3.5

[94] Tobias Höllerer Ryan Bane. Interactive tools for virtual x-ray vision in mobile augmented reality. August 2004. 2.2

[95] Microsoft Corporation Sam Z. Glassenberg. Directx graphics: Direct3d 10 and beyond. In *WinHEC*, 2006. 2.3, 5.3

[96] Dieter Schmalstieg. Augmented reality techniques in games. July 2005. 3.4

[97] Seppo Äyräväinen. 3D Magic Lenses, Magic Lights, and their application for immersive building services information visualization. Technical report, Helsinki University of Technology, telecommunications software and multimedia labratory, 2003. 2.1, 3.2.1

[98] University of Utah Simon Premoze, Computer Science Department. Computer generation of panorama maps. 5.3.4

[99] Wolfgang Straßer Stanislav L. Stoev, Dieter Schmalstieg. The through-the-lens metaphor: Taxonomy and application. *Proceedings of the IEEE Virtual Reality*, 2002. 2.2, 3.4

[100] Stefan Maaß, Jürgen Döllner. Ein Konzept zur dynamischen Annotation virtueller 3D-Stadtmodelle. In *Deutsche Gesellschaft für Kartographie, Kartographische Schriften, Band 10: Aktuelle Entwicklungen in Geoinformation und Visualisierung.*, 2006. 3.3.3

[101] Stefan Maass, Jürgen Döllner. Dynamic Annotation of Interactive Environments using Object-Integrated Billboards. In *WSCG*, 2006. 3.3.3

[102] T. Alan Keahey. The Generalized Detail-In-Context Problem. In *IEEE Visualisation*, 1998. 2.2

[103] T. Alan Keahey, Edward L. Robertson. Nonlinear Magnification Fields.

[104] G. Gröger T. H. Kolbe. Towards unified 3d city models. July 2003.

[105] K. Hinrichs T. Ropinski, F. Steinicke. An efficient approach for emphasizing regions of interest in ray-casting based volume rendering. September 2005. 2.2

[106] Tamy Boubekeur and Christophe Schlick. Generic Mesh Refinement On GPU. In *Proceedings of ACM SIGGRAPH/Eurographics Graphics"Proceedings of ACM SIGGRAPH/Eurographics Graphics Hardware*, 2005. 2.3

[107] Oliver Deussen Thomas Luft, Carsten Colditz. Image enhancement by unsharp masking the depth buffer. 5.3.1

[108] Klaus Hinrichs Timo Ropinski, Frank Steinicke. Visual exploration of seismic volume datasets. *Journal of WSCG, ISSN 1213-6972, Vol.14, 2006 Plzen, Czech Republic. Copyright UNION Agency – Science Press*, 14, 2006. 2.2

[109] Timo Ropinski, Klaus Hinrichs. Real-Time Rendering of 3D Magic Lenses having arbitrary convex Shapes. *WSCG*, 12(1-3), February 2004. 2.1, 6

[110] Philippas Tsigas Ulf Assarsson, Niklas Elmqvist. Image-space dynamic transparency for improved object discovery in 3d environments. Technical Report 2006-10, Department of Computer Science & Engineering, Chalmers University of Technology and Goeteborg University, Sweden, 2006.

[111] Uwe Rauschenbach Tino Weinkauf Heidrun Schumann. Interactive Focus and Context Display of Large Raster Images. In *WSCG Proceedings*, 2000. 2.2

[112] Verena Giller , Manfred Tscheligi , Johann Schrammel , Peter Froehlich , Birgit Rabl , Robert Kosara , Silvia Miksch , and Helwig Hauser. Experimental Evaluation of Semantic Depth of Field, a Preattentive Method for Focus+Context Visualization. In M. Rauterberg et al., editor, *Human-Computer Interaction – INTERACT'03*, pages 888–891. IOS Press, 2003. 2.2

[113] Y.K. Leung, M.D. Apperley. A Review and Taxonomy of Distortion-Oriented Presentation Techniques. *ACM Transactions on Computer-Human Interaction*, 1:126–160, 1994. 1.1, 2.2, 3.4

[114] Yonggao Yang Jim X. Chen Mohsen Beheshti. Nonlinear Perspective Projections and Magic Lenses: 3D View Deformation. *IEEE Computer Graphics and Applications*, pages 76–84, 2005.

Fragment- and Vertex-Shader

```
#ifndef FRAGMENTINTERFACE
#define FRAGMENTINTERFACE
struct us_FragContext
{
    bool     us_useMRT;
    vec4     us_FragColor;
    vec4     us_FragData[gl_MaxDrawBuffers];
    float    us_FragDepth;

};//endstruct us_FragContext
#endif

void onLighting(inout us_FragContext context)
{
    context.us_FragColor = gl_Color;
    return;
}

#define ONFNISH optional
void onFinish(in us_FragContext context)
{
    gl_FragColor = context.us_FragColor;
    return;
}
```

Listing 6: *Basic fragment handler for directional lighting*

```
#ifndef VERTEXINTERFACE
#define VERTEXINTERFACE
struct us_VertContext
{
    vec4    us_Position;
    vec3    us_Normal;
    float   us_PointSize;
    vec4    us_ClipVertex;

};//endstruct us_VertContext
#endif

varying vec3 normal;

#define ONINIT optional
void onInit(inout us_VertContext context)
{
    context.us_Position = gl_Vertex;
    context.us_Normal   = gl_Normal;
}

void onTransform(inout us_VertContext context)
{
    context.us_Position =
        gl_ModelViewProjectionMatrix * context.us_Position;
    context.us_Normal   =
        normalize(gl_NormalMatrix * context.us_Normal);
}

void onLighting(inout us_VertContext context)
{
    vec3 direction =
        normalize(vec3(gl_LightSource[0].position));
    float NL = max(dot(context.us_Normal, direction), 0.0);
    gl_FrontColor =
        NL * gl_FrontMaterial.diffuse * gl_LightSource[0].diffuse;
}

#define ONFINISH optional
void onFinish(inout us_VertContext context)
{
    gl_Position = context.us_Position;
    normal      = context.us_Normal;
}
```

Listing 6: *Basic vertex handler for directional lighting*

Eidenstattliche Erklärung

Ich versichere hiermit, die von mir vorgelegte Arbeit selbständig verfasst zu haben. Alle Stellen, die wörtlich oder sinngemäß aus veröffentlichten oder nicht veröffentlichten Arbeiten anderer entnommen sind, habe ich als entnommen kenntlich gemacht. Sämtliche Quellen und Hilfsmittel, die ich für die Arbeit benutzt habe, sind angegeben. Die Arbeit hat mit gleichem Inhalt bzw. in wesentlichen Teilen noch keiner anderen Prüfungsbehörde vorgelegen.

Potsdam, the
Matthias Trapp

Conceptual Formulation

Diplom-Arbeit

Analyse und Exploration virtueller 3D-Stadtmodelle durch 3D-Informationslinsen

Themenstellung für Herrn Matthias Trapp

Stand 04.04.2006

1. Zusammenfassung

In der Diplomarbeit von Herrn Matthias Trapp sollen 3D-Informationslinsen, die auf der Fokus-&-Kontext Metapher basieren, auf virtuelle 3D-Stadtmodelle übertragen und dabei analysiert, konzipiert, implementiert sowie bewertet werden. Die neu zu entwerfenden computergrafischen 3D-Linsentechniken sollen Objekte und Strukturen von 3D Stadtmodellen sowie deren Umgebung einbeziehen. Ziel ist es, mit den 3D-Linsentechniken die perzeptorische, kognitive und grafischen Qualität der Visualisierung im Vergleich zu bisherigen perspektivischen Projektionen zu erhöhen. Eine Auswahl von unterschiedlichen 3D-Informationslinsen soll praktisch umgesetzt werden.

2. Gegenstand

Hauptgegenstand der Diplomarbeit von Herrn Trapp bilden virtuelle 3D-Stadtmodelle. Sie präsentieren eine spezialisierte Form geovirtueller Umgebungen und sind gekennzeichnet durch ein zugrunde liegendes 3D-Geländemodell, einer darin befindlichen 3D-Bebauung und 3D-Vegetation sowie des dazu komplementären Strassen- und Grünflächenraumes. 3D-Stadtmodell-Systeme ermöglichen es dem Nutzer, sich im Modell interaktiv zu bewegen und sie stellen die Grundfunktionen für die Exploration, Analyse, Präsentation und Editierung der raumbezogenen Information bereit.

Die Detail-&-Kontext Visualisierung ist in den Bereichen 3D-Geländemodelle, Graphenvisualisierung, etc. bereits grundlegend untersucht. Im Gegensatz dazu gibt kaum Ansätze zur Übertragung und Spezialisierung dieses Ansatzes auf virtuelle 3D-Stadtmodelle. Gerade hier ist jedoch eine gezielte Visualisierung von z. B. kontextbezogenen Daten zu Objekten (z. B. Points-of-Interest oder Landmarken) von großer Bedeutung für die interaktive Exploration und Analyse.

Die Diplomarbeit soll eine Einbettung der Detail-&-Kontext Visualisierung in 3D-Stadtmodell-Systemen im Bereich der Gebäude im Level-of-Detail 1 bis 2 (i.S.v. CityGML) untersuchen und erweitert damit den Umfang an interaktiven 3D-Visualisierungstechniken. Im Mittelpunkt der Diplomarbeit stehen interaktive, computergrafische 3D-Informationslinsen, die Geometrie- und Metainformationen virtueller 3D-Stadtmodelle besser und gezielt vermitteln. Dabei sollen diese Techniken u. a. die Geometrie, Gestaltung und Projektion der virtuellen 3D-Stadtmodelle modifizieren, um beispielsweise Gebäudeverdeckungen aufzulösen oder Gebäudekomplexe aufzuschlüsseln.

3. Aufgabenstellung

3.1 Stadtmodelle

Zu Beginn der Arbeit sind konkrete virtuelle 3D-Stadtmodelle zu sichten und aufzubereiten. Diese Modelle sind allgemein aus folgenden Komponenten zusammengesetzt:

- Geländemodell
- Gebäudemodelle
- Strassenraummodelle
- Vegetationsmodelle

Stadtmodelle werden vom HPI für die Städte Berlin, Potsdam und München bereitgestellt. Weiter existiert ein 3D-Modell des HPI-Campus. Zusätzlich können im Rahmen der Arbeit eigene, prototypische 3D-Stadtmodelle erstellt werden (z. B. Reihenhauszeilen, Doppelhaus-Siedlung, etc.).

3.2 Linsentechniken

Den Hauptteil der Arbeit nimmt die Erstellung von Visualisierungstechniken ein. Entwickelt werden sollen diese wie folgt:

- **Allgemeines 3D-Informationslinsen-Handling** als grundlegende Renderingtechnik und genutzt als Ausgangsbasis für die Entwicklung der spezifischen Linsentechniken. Das Handling umfasst sowohl das Rendering wie auch die interaktive Steuerung der Linse durch den Nutzer.
- **Konkrete 3D-Informationslinsen.** Die jeweiligen konkreten Linsentechniken nehmen Bezug auf die Semantik der dargestellten Objekte des virtuellen 3D-Stadtmodells und nutzen generell Distortions-, Deformations- und Displacement-Verfahren, um den Linseninhalt gezielt geometrisch-grafisch aufzubereiten. Geplant sind folgende Techniken:
  - **Bending-Informationslinse**: Sie „verbiegt“ Stadtmodell-Objekte, um besseren Einblick in die räumliche Situation zu gewähren.
  - **Flattening-Informationslinse**: Sie verflacht Bereiche, um dazu komplementäre Bereich hervorzuheben.
  - **Best-View-Informationslinse**: Sie zeigt Stadtmodell-Objekte in einer prioritätsdefinierten Weise, um besondere Aspekte bzw. Seiten der Stadtmodell-Objekte vorrangig darzustellen.

  Zu jeder Linsentechniken werden Annahmen über das Vorhandensein spezieller raumbezogener Zusatzinformationen aufgestellt. Diese Informationen können z. B. in Form georeferenzierten Rasterinformationen vorliegen (z. B. Landnutzungsdaten).

3.3 Optionen

Mögliche Erweiterungen der Themenstellung umfassen:

- Einsatz spezieller Gestaltungs- und Illustrationstechniken für die Darstellung des Linsen-Inhalts, z. B. abstrahierende Darstellungen oder Einblendung von Fachinformationen.
- Exploration erweiterter Interaktionstechniken für 3D-Informationslinsen.
- Exploration erweiterter Intergrationstechniken für 3D-Informationslinsen.
- Erweiterung der Einsatzmöglichkeiten für Gebäude mit LOD 3 oder 4.

4. Umsetzung

Die Diplomarbeit beginnt Anfang April 2006. Es wird erwartet, dass die Arbeit in einem Zeitraum von grundsätzlich sechs Monaten abgeschlossen sein wird. Ein Arbeitsplatz wird im HPI bereitgestellt. Die Hardware-Ausstattung umfasst eine Standard-Workstation mit einer OpenGL 2.0 kompatiblen Grafikkarte. Es wird eine Anwesenheit von mindestens drei Tagen pro Woche erwartet.

Die Implementierung der notwendigen Algorithmen und Datenstrukturen erfolgt nach dem OOP Paradigma und wird technisch auf die Programmiersprache C++ abgebildet. Die Implementierung der Renderingverfahren erfolgt unter Nachnutzung des VRS sowie des OpenGL API. Die Software-Architektur soll so gewählt werden, dass das Ergebnis als eigenständiges Sub-System verwaltet wird sowie eine spätere Nachnutzung und Integration als Komponente des LandXplorer-Visualisierungssystems erfolgen kann.

5. Betreuung der Diplom-Arbeit

Zur Sicherstellung einer angemessenen Betreuung und der Qualität der Arbeit sowie zur Gewährleistung der Planbarkeit der Ressourcen des Fachgebiets wird folgendes vereinbart:

- Die Betreuung der Arbeit erfolgt gemeinsam durch Prof. Dr. Döllner, Haik Lorenz sowie Henrik Buchholz.
- Herrn Trapp wird empfohlen, den Betreuern seine Fortschritte bei der Bearbeitung *regelmäßig* mitzuteilen und aufkommende Fragen und Alternativen *frühzeitig* zu diskutieren.

6. Schriftliche Arbeit

Als Bestandteil der Diplomarbeit ist eine schriftliche Ausarbeitung anzufertigen, die einen Umfang von 60-80 Seiten nicht übersteigen sollte. Es steht dabei die kompakte und präzise Beschreibung der eigenen Arbeit im Vordergrund. Die Darstellung der Kontexte (Computergrafik, Visualisierung, Geovisualisierung, Rendering allg.) soll auf das notwendige Maß reduziert sein, so dass genügend Raum für die Ergebnisse der eigenen Arbeiten bleibt. Die Diplomarbeit wird aufgrund der erzielten Ergebnisse im systemtechnischen Bereich und der schriftlichen Ausarbeitung beurteilt. Die schriftliche Arbeit soll im Stil eines wissenschaftlichen Fachbeitrages erbracht werden.