FAKULTÄT
FÜR !NFORMATIK
Faculty of Informatics

Splitting of Meshes in Image-Space

BACHELORARBEIT

zur Erlangung des akademischen Grades

Bachelor of Science

im Rahmen des Studiums

Medieninformatik und Visual Computing

eingereicht von

Barbara Schwankl
Matrikelnummer 0852176

an der Fakultät für Informatik
der Technischen Universität Wien

Betreuung: Ao.Univ.Prof. Dipl.-Ing. Dr.techn. Eduard Gröller
Mitwirkung: Projektass.(FWF) Ing. Peter Mindek

Wien, 05.04.2013

(Unterschrift Verfasserin) (Unterschrift Betreuer)

FAKULTÄT
FÜR !NFORMATIK
Faculty of Informatics

Splitting of Meshes in Image-Space

BACHELOR'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Bachelor of Science

in

Media Informatics and Visual Computing

by

Barbara Schwankl
Registration Number 0852176

to the Faculty of Informatics
at the Vienna University of Technology

Advisor: Ao.Univ.Prof. Dipl.-Ing. Dr.techn. Eduard Gröller
Assistance: Projektass.(FWF) Ing. Peter Mindek

Vienna, 05.04.2013

(Signature of Author) (Signature of Advisor)

Erklärung zur Verfassung der Arbeit

Barbara Schwankl
Unionstr. 133, 4020 Linz

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

(Ort, Datum) (Unterschrift Verfasserin)

Kurzfassung

Hier fügen Sie die Kurzfassung auf Deutsch gemäß den Vorgaben der Fakultät ein.

Abstract

With modern technologies, representing complex 3D data is no problem at all. The challenge is to reveal data that is concealed by solid geometry and retaining its context at the same time.
In this paper, several approaches are presented that deal with this problem by finding user-centered solutions that are adjusted for each individual requirement. Moreover, a simple algorithm is proposed that combines existing approaches to reveal occluded structures. Therefore, a descriptive implementation of this algorithm is shown with *VolumeShop*, an application that flexibly supports visualization research.

Contents

CHAPTER 1

Introduction

Data visualization is an substantial part of computer graphics. To make the visualized data easily examinable, user interaction plays an important role. Hence, the user should be able to translate, rotate and scale the 3D object. As technology progresses, the ability to represent highly complex objects increases. This arouses the desire to examine parts of the object in detail and also regarding the individual parts within a specified context. Additionally, the examination and analysis of the objects' inner structures can also be of interest. For example in the medical sector, having all the data digitally available to explore opens up plenty of possibilities. Hence, the challenge is how all these data can be provided with a maximum of interactivity for the user to be adapted optimally.

An idea is to let the user determine a region or a object of interest, regardless if this region is already visible, concealed by a jacket, or simply occluded by another object. In case it is not clearly visible, the jacket or the occluding object could simply be omitted, but this would also cause the context to get lost. As the context can be of significant importance, this approach is not eligible. In case the region or object of interest is already visible, the main aim of the user can be to examine the individual parts of this region or object, or its connection and interaction with the remaining objects. Again, without the context, the information can become less instructive. Hence, the idea is to reveal the region of interest whilst keeping the context.

Illustrative visualisation simplifies the representation of complex data. Several scientific approaches exist for the illustrative visualization. One idea is to simply lower the opacity of the outer structures. For example to reveal a brain, the approach would be to simply raise the transparency of the skull and the skin respectively for making the parts of secondary interest semi-transparent. This approach is called *ghosting*. Another approach, *cutaway views*, is to cut out parts of the object so that no information gets lost and still the region of interest would be fully visible. A different approach are so-called *exploded views* that unfold an object by breaking it into multiple parts and shifting those apart whilst keeping the object of interest in the center.

1.1 Motivation

This thesis examines several approaches to reveal regions of interest including inner structures of complex 3D models. While revealing the regions of interest, these approaches retain the context to be able to examine the region of interest in respect of its surroundings. Furthermore,*VolumeShop*, an application for visualization research, is presented as a powerful tool to implement various approaches to support a convenient level of examination of complex 3D data.

1.2 Problem statement and objectives

The implementation of high-level tools and satisfying various individual needs for examining an object can be cumbersome and time consuming. Furthermore, with increasing complexity of an object the rendering time can rise significantly as well. Therefore, an approach is demanded that considers the need for user-interaction to define regions or objects of interest, and additionally has an invariant computing effort for the rendering, regardless of the complexity of the object.

In section 2, several existing approaches are being presented and analyzed. Furthermore, the operation with VolumeShop and its concept is introduced in section 3. A simple algorithm for revealing occluded objects is proposed in section 4.

CHAPTER 2

Analysis of existing approaches

Exploring volume data and polygonal meshes is a complex task. Every research has different points or *regions-of-interest* (ROI) and the methods to reveal those regions depend on the purpose of the study. Hence, regions are classified by their *degree-of-interest* function (DOI) [11]. A high DOI means a region is of high interest, a low DOI stands for a region of secondary interest.
There are three common methods for revealing occluded structures of an object:

- Transparency/Ghosted views
- Cut-away views
- Exploded views

2.1 Transparency/Ghosted views

These techniques do not discard any parts of displayed objects but let them vanish to a certain degree. Hence, it lowers the opacity of certain data points. For example, the opacity of the outer structure of an object is decreased so that the inner structure is revealed [5].
Increasing the transparency of occluding parts makes ROIs visible but at the same time makes it difficult to distinguish the several semi-transparent layers and identify their spatial composition [10].
Bruckner and Gröller [2] use a ghost object explicitly to preserve the context of illustrations. When the user defines a ROI, several transformations can be applied to it while at the original position, a faded version of the ROI will be visible.
In their work, Bruckner and Gröller describe a method with weighted membership functions of background, ghost and selection respectively to define the color of the resulting illustration. The opacity for a point p is determined by the grade of membership in the union of all sets.

2.2 Cut-away views

Cut-away views, also called cutaways, reveal ROIs that are occluded by objects of secondary interest. The latter are cut out in order to make the ROI visible [2][4][5][8]. Omitting the occluding regions increases comprehension of spatial relationships between the components. Also, the position and orientation of ROIs are shown in context of their surrounding structures [10].

Li et al. [10] present a method based on the ideas that cuts are made in respect to their geometry and that interactive exploration of the 3D models is strongly supported. Removing parts of the object is handled carefully so that the user can mentally reconstruct the missing geometry. Additionally, the cuts are view dependent. The farther a structure is away from the viewpoint, the less it is cut. To increase interactivity the viewpoint and the cutting parameters can be controlled by the user. Li et al. state that low-level controls like cutting planes that need to be precisely positioned require a certain expertise from the user to reveal ROIs and thus should not be used. In their approach, users can set any viewpoint and dynamically manipulate the model with the mouse to define the ideal cut. Their system has two components. The authoring interface and the viewing interface. While the authoring interface enables the user to adjust a 3D model by several parameters, the viewing interface takes this adjusted model and lets the user explore it by providing high-level cutaway tools.
TODO peter Regarding cutaway-views, the viewing interface is in focus. It gives the user the choice between direct manipulation or automatic cutaway generation. Cuts can be directly modified by interaction with the mouse. The cut is resized by snapping the nearest cutting face to a surface point, moving the position of the cut is possible either in only one dimension or in all three directions at once. Also, multiple cuts at once can be accomplished by the use of an occlusion graph[1]. The system updates all cuts by the inset constraints in both directions of the occlusion graph. A fast method to expand or collapse cuts is by cutting out or closing entire layers of structures. This can be achieved by clicking on a structure and all structures above or below in the occlusion graph are cut out or closed. Cross-sectional surfaces are exposed by changing the angle of the cuts. The degree of bevelling can be adjusted by the user with a slider. Subsequently, the faces of the revealed structures turn to the viewer automatically.
Higher-level interfaces involve algorithms for automatically exposing target structures that are pre-selected by the user. Therefore, the user selects a set of targets from a list. The system then determines the cutting parameters as well as a viewpoint. All parts of the cutting volume above a target structure in the occlusion graph are fully expanded, all parts below the target structure and the target structure itself are completely closed. This step assures a maximum exposure of the target structures. To preserve the context from occluding structures, the corresponding cutting volume of each non-target part is being closed as much as possible regarding the inset constraints and considering not to occlude target structures. For this step, the algorithm works from the leaves of the occlusion graph upwards. Those steps finished, all non-target structures are being desaturated to highlight the target structures.

[1] This is a graph generated in respect to the viewpoint that defines the number of occluding structures to be removed in order to become visible. An occlusion graph is assigned to each part of the model.

An interesting approach for adaptive cutaways is presented by Burns and Finkelstein [4]. Their method allows interactive rendering of adaptive cutaways. It ensures that objects of interest are not obscured by objects of secondary importance, so-called *secondary objects*. Depending on the position and the projection of the camera, the cutaways have to be adapted to guarantee free sight of the objects of interest.
The main part of the work is the depth image cutaway representation. It is based on an approximation of the chamfer distance transform algorithm. The depth values of the rendered back hulls of the objects of interests are used as an input. The final result is a depth image containing the objects of interest and additional *drill holes* around them with a certain slope depending on the camera position and projection.
The depth buffer is used for an additional depth check in the fragment shader, but the OpenGL pipeline only supports one depth buffer. If a fragment of a secondary object has a depth value smaller than the depth value in the cutaway depth image, it is discarded, because it would occlude objects of interest. As a last step, the objects of interest are rendered without checking against the cutaway depth buffer.
The cutaway depth image must be computed every frame to ensure interactive real-time cutaways. The approach has the following advantages:

- the shape of the cutaways is defined by the silhouette of the objects of interest
- the cutaway surface is view dependent
- the angle that defines the slope of the drill holes around the objects of interest can be adapted
- multiple distant objects of interest have their own cutaways which can merge when the objects converge

Special attention is drawn to the rendering of the cut surfaces of secondary objects. Because the geometry is clipped, the secondary objects would have a hollow appearance. Therefore, fragments of back facing polygons of the secondary objects are shaded according to the normals of the cutaway shapes. As a result, the geometry cut appears as carved solid objects. Slightly fading out ghost lines of the silhouette of the clipped polygons additionally improve the perception of the embedment of the objects of interest.

Optimizing the visibility of important target sections by defining a DOI function is an approach introduced by Sigg et al. [11]. The user specifies a DOI function for either a polygonal mesh or volume data that assigns a value of importance to each vertex or voxel. Additionally, the shape and the maximum number of geometric primitives being cut out can be selected, but those cutting primitives should be limited to pre-defined shapes in order to be comprehensible. Sigg et al. provide the user with three kinds of shapes: cuboids, cylinders and spheres. Although the user can select the points of interest, the exact position and size of the cutting primitives is computed by the system. That means that the user roughly defines important regions while the optimization is done by the system. For the optimization, the resulting image from the renderer is used what makes it easy to integrate the it into existing rendering frameworks.

The optimization process is iterative. The most important step is the image analysis with the objective function. It takes the viewing perspective into account. The image data is divided into two categories, positive for the important parts and negative for the unimportant parts. To achieve a maximum context for the ROIs the omitted data has to be minimized in those regions. The formula for maximizing the objective function:

$$f(A,P) := \frac{1}{d}\sum_{i=1}^{n}(\alpha \cdot A_{i,pos} - A_{i,neg}) - \beta \cdot V(P) \tag{2.1}$$

A is the image expressed in a pixel array of length d = width x height. Each pixel has an important and an unimportant part, *pos* and *neg*. V(P) measures the amount of omitted data of the set of cutting primitives P. α and β define the weighting of the importance of the target part and the surroundings. α regulates the importance of the target parts being shown, β defines the importance for preserving the surroundings.
With this approach comprehensibility is maintained throughout the complete process and the rendering settings are preserved.
TODO clarify
Knödel et al. present an approach with strong focus on user-modified cutaways [8]. For cutting a model, they use four basic shapes: spheres, cubes, wedges and tubes. To perform the actual cut, they use the principle of *Constructive Solid Geometry* (CSG) [6][7] algorithms that creates the cut by intersecting shapes, taking the union of them or subtracting one shape of the other.

Cur-away views are also used in combination with ghosting. This would mean that the region cut out is replaced by a faded duplicate of the original. Features such as edges are attempted to be preserved. Therefore, the *ghost* stands for the original region before cutting [2].

2.3 Exploded views

TODO peter: objects can be just moved around without actual cutting into smaller parts. An exploded view of an objects shows the object decomposed into several parts so that the inner structures are revealed. Also, cross-sections become visible [3]. The individual parts are separated from each other in respect to the global structure of the object as well as to the local spatial relationships [9]. In contrast to ghosted views or cutaways, no contextual information gets lost, but it may come to visual clutter when every single part of the object is exposed.

A system for creating and viewing interactive exploded views of complex 3D models is introduced by Li et al [9]. Their aim is a system that allows users to explore the spatial relationships between specific ROIs. Target parts can be selected from a list in order to only expose the target parts and not showing anything else from the object. Thereafter, the parts can be directly expanded or collapsed to show a comprehensible spatial relationship.
An iterative algorithm is used to remove unblocked parts from the model and adds them to an

hierarchical, acyclic explosion graph[2]. Initially, all parts of the model are added to a set S. At each iteration, those parts that are unblocked into at least one direction are determined and added to a set P. Further on, a part $p \in P$ is determined that requires the shortest distance to release itself from its adjunctive parts. Within the graph, p is linked to its adjunctive parts with an edge and information about the direction of the release, the explosion direction, is stored. Finally, p is removed from S. When no more parts can be removed from S the algorithm terminates and the explosion graph is complete.
In case a part hierarchy exists, the explosion graph is divided into several overlapping sub-assemblies that allow independent expansion and collapse of a subset of parts of the model. For each sub-assembly an explosion graph is calculated by applying the algorithm described above.

Interaction is guaranteed by direct user controls and higher-level interaction modes. The direct controls include animated expand and collapse, direct manipulations and riffling. With a click of the mouse, the model is fully exploded or entirely collapsed. This is done in reverse topological order with respect to the explosion graph to avoid that blocking constraints are not being violated. By dragging a part with the mouse, it is being slid its explosion direction, continually updating its offset. This way, selected parts can be examined in detail by gradually exploding or collapsing them. The blocking constraints are maintained constantly during the manipulation due to the system that checks for blocking parts within the explosion graph. If there is a blocking part found amongst the descendants, further explosion is inhibited. Riffling means an explosion of those parts regarding their explosion direction where the mouse slides over. As long as the cursor moves over the part, it remains exploded. If the cursor moves on, the explosion is undone. A click prompts the part to remain exploded. This way, the user obtains a quick overview in what direction the parts of the model explode and how certain parts are assembled and connected to each other.
The high-level interaction mode causes the user-selected target parts to explode automatically. After the user chose the desired targets from a list of model parts, those target parts are labelled and exposed. All non-target parts are collapsed. This step is implemented with a smooth animation to assure traceability. Exposure is realized with either explosions or a combination of cutaways and explosions.
Exploded views for non-hierarchical models have to meet two conditions: each part p must not occlude any target part and if p is a target part itself, it must not be occluded by any other part. For this reason, each part of the explosion graph is visited in topological order. If necessary, p is moved in order to not occlude any target part. Additionally, target parts are emphasized by encapsulation. That means they are separated from all touching parts.
For the combination of cutaways and explosions, the selected target part is first exposed by a cutaway within its context, then moved away from the rest of the model through the cutaway hole and finally exploded to reveal all its details. For an effective cutaway, the explosion direction has to be determined before the translation and the cutaway hole has to be large enough to let the target part move through.

Bruckner and Gröller introduce a force-directed layout for exploded views of 3D information

[2] An explosion graph states the order in which parts can be exploded without blocking.

in their work about *Exploded Views for Volume Data* [3].
In their approach, they divide the volume data into regions with a DOI > 0, and regions with a DOI = 0. All regions with a DOI > 0 are part of the selection, the other regions belong to the background that represents the context. While the parts of the background are being transformed, the selection remains static. The background is divided into a user-defined number of parts that do not intersect. Subsequently, those background parts are moved into different directions to reveal the selection. To enable the user to control how far the background is moved away from the selection, a parameter *degree-of-explosion* (DOE) is introduced. If the DOE equals zero, the background is not moved at all and clasps the selection. The higher the DOE, the farther the background parts are moved away from the selection. With the DOE, the view dependency can be reduced and spacing can be increased. Additionally, physical forces are simulated in a force-directed layout. All parts of the object are structured in a graph. Within this graph, repulsive forces are assigned to each node and attractive forces are assigned to adjacent nodes. The aim is to avoid conclusions completely by the minimum displacement. To achieve a steady state, all forces should be in equilibrium.
A number of forces are defined:

- Return force
- Explosion force
- Viewing force
- Spacing force

The *return force* attracts the background parts to their original location. The *explosion force* pushes them off the selection object. The *viewing force* causes the background parts to not occlude the selection for the current viewing transformation. This is essential, as the user can rotate the camera arbitrarily. Finally, the repulsive *spacing force* prevents the parts from clustering. All forces, except the return force that remains constant, are scaled with the DOE parameter.

TODO absätze gleich gestalten: newline or no newline?

CHAPTER 3

Methodology

VolumeShop is an interactive hardware-accelerated application for direct volume illustration [2]. It is designed for developers to have maximum flexibility for visualization research. The functionality of the program is implemented with plug-ins that are functionally independent components, but its properties can also be linked to those of another plug-in. This way, a plug-in has access to the data of another plug-in. Plug-ins are hosted by containers that provide all necessary resources for them. [1].

3.1 Plug-ins in VolumeShop

Plug-ins can be dynamically loaded, and suspended and resumed at runtime. One main advantage in development is that the application does not need to be closed when a plug-in is recompiled. This is possible due to the fact that plug-ins are compiled into Dynamic Link Libraries (DLLs) that are scanned for changes by VolumeShop. When a change is being detected, the plug-in is reloaded.

Types of plug-ins

In VolumeShop, several types of plug-ins exist:

- Renderers
- Interactors
- Compositors
- Editors

In short, *Renderers* are responsible for the way the polygonal objects are displayed, *Interactors* provide common interaction functionality like cameras, *Compositors* combine the output of multiple renderers or interactors and *Editors* are specialized GUI widgets for certain tasks [1].

Properties

The complete state of a plug-in is defined by its properties which constitute the plug-ins' functionality [1]. A property can be easily created with the following command that shows an example for an integer property in the range [0,255]:

```
GetPlugin().GetProperty("Test") = Variant::TypeInteger
   (12,0,255);
```

For extended functionality there is the possibility of linking properties. The change of a property causes linked properties to change as well.
Creating links in the Graphical User Interface (GUI) is performed by simply right clicking the property with the mouse and choosing the desired linking property.
An example for creating links programmatically is stated in the following code fragment:

```
// Link property "MyProperty" to property "LinkedProperty"
PropertyContainer::Link myLink(pTargetObject,"LinkedProperty");
GetPlugin().SetPropertyLink("MyProperty",myLink);
```

Observers

Observers allow tracking changes in properties or other objects. Notifications are being bound to member functions with the class *ModifiedObserver*. This class notifies changes from multiple objects of different types [1].
An example for using observers:

```
// usually a class member
ModifiedObsever myObserver;

// typically in plugin constructur
// connect observer to member function
myObserver.connect(this,&MyPlugin::changed);

// add observer to objects we want to track
GetPlugin().GetProperty("MyProperty1").addObsever(&myObserver);
GetPlugin().GetProperty("MyProperty2").addObsever(&myObserver);

// notification handler
void changed(const Variant & object, const Observable::Event &
   event)
{
      // handle changes, e.g., trigger re-render
```

```
        GetPlugin().update();
}
```

This code snippet connects an observer to a property. Whenever this property is being changed in the GUI, the observer is being informed about it through a callback function and can react appropriately by re-rendering the illustration and updating its attributes. TODO peter

Technologies

The plug-ins are written in C++ using OpenGL which is a successful cross-platform graphics application programming interface (API) for 2D and 3D computer graphics [7]. For shading and texturing OpenGL Shading Language (GLSL) is used [1].

3.2 Concept of shaders

In 3D computer graphics objects are described with a set of polygon surface patches and are called *polygonal mesh* or simply *mesh*. Each polygon has several vertices, edges and faces [6]. With GLSL the shading of the polygons can be modified directly with programmable *shaders*, replacing the fixed function pipeline of OpenGL. These shaders are parallelly executed for every vertex and every fragment in the *graphics processing unit* (GPU) and allow the usage of customized effects.

In GLSL there are four different types of shaders:

- Vertex shader
- Fragment shader
- Geometry shader
- Tessellation shader

The purpose of the two basic shaders, vertex shader and fragment shader, will be described in the following section.

Vertex shader

The main purpose of the vertex shader is the computation of the final vertex position. The vertex data is taken as input. A single vertex can consist of several attributes including position, color and normal vector.
The vertex shader can perform tasks such as [7]:

- transforming the vertex position
- transforming the normal vector and normalizing it
- generating and transforming texture coordinates

- applying light (such as ambient, diffuse and specular) per vertex
- computing color

Fragment shader

After the vertices have been transformed into the view plane they are rasterized. Data defined as output of the vertex shader is automatically interpolated before it is passed on to the fragment shader. The output of the rasterizer are fragments which contain information about screen coordinates, depth, color and texture coordinates. The fragment shader defines the final color of the fragment. As fragments can have the same screen coordinates it is possible that multiple fragments can contribute to the same pixel in the frame buffer [7].
The fragment shader can perform tasks such as [7]:

- per-pixel-lighting (using interpolated normals from the vertex shader)
- normal-mapping (looking up normals from a texture)
- bump-mapping (computing normals based on a hight-map of a texture)

The normals obtained from normal-maps and bump-maps are also used for lighting calculation and result in a better illumination than the computation with per-pixel-lighting that uses interpolated normals.

Discarding fragments

In the fragment shader, a break condition is available. The keyword *discard* drops the current processed fragment and exits the shader[1]. This is useful for speeding up the computation, because fragments that are not meant to become pixels do not have to be passed on to the next stage of the OpenGL pipeline.

Per-fragment operations

After the fragment shader, per-fragment-operations like depth-test and stencil-test are performed, before the fragment color is written to the frame buffer. If blending is enabled the fragment color is blended with the existing pixel color in the frame buffer.

[1] http://wiki.delphigl.com/index.php/Tutorial_glsl

CHAPTER 4

Suggested implementation

In this section a step by step approach to reveal inner structures of a mesh will be discussed. First a plane needs to be defined that represents the position and direction of the cut. To retain interactivity, several parameters in the VolumeShop interface are available to translate, rotate and scale the plane. The color and opacity of the plane should be adaptable to not occlude parts of the mesh. For the mesh splitting an offset is defined that indicates how far the two halves of the mesh shall be apart from the plane. The larger the gap the better the insight into the model. The splitting itself is no real translation of the two halves, but the model is rendered twice at different positions in space parallel to the plane. Respectively, the fragments on the other side of the plane are discarded and not rendered to create the illusion that the mesh has been split. After the cut, the back facing triangles become visible in those areas where the model has been cut. Therefore, the final step is shading these back faces to create shading for the cut surface.

4.1 Definition of the plane

The cutting plane to calculate the cut is defined by a point on the plane and a normal vector. Its dimension is per definition indefinite. Visually, the plane is represented as a rectangle with a certain user-defined dimension that is adaptable to the size of the mesh to be split. This gives the user an idea where the cut in the mesh is performed. Additionally, to increase comprehension for the user, the color of the plane, its opacity and scaling factor can be set by the user. The orientation of the cutting plane can be defined with properties like translation and rotation vectors that influence the calculation of the mesh splitting directly.
Defining a name and a type for a property is mandatory. The following code sample shows the initialization of a property named *Plane Translation Vector* of the type *Vector*.

```
1 "Plane Translation Vector" GetPlugin.GetProperty
```

Algorithm 4.1: Initialization of a property named *Plane Translation Vector* of the type *Vector*

To apply changes made in the interface immediately, an observer for the property has to be added:

```
GetPlugin().GetProperty("Plane Translation Vector").addObserver
   (&m_modVariantObserver);
```

Before drawing the plane, the Viewing Transformation Matrix is loaded. This matrix is for transforming the coordinates from world space into viewing space [6].
Then the plane is being adjusted by its affine transformations. All parameters passed are of the type *float* and taken from the user input.
The commands in OpenGL:

```
glTranslatef(vecPlaneTranslation.GetX(), vecPlaneTranslation.
   GetY(), vecPlaneTranslation.GetZ());
glRotatef(vecPlaneRotationAngle, vecPlaneRotation.GetX(),
   vecPlaneRotation.GetY(), vecPlaneRotation.GetZ());
glScalef(vecPlaneScaling.GetX(), vecPlaneScaling.GetY(),
   vecPlaneScaling.GetZ());
```

TODO explain code

The color of the plane is handed over from the input panel, normalized, and passed on to the renderer. In VolumeShop, this can be easily achieved by calling the function *GetNormalized<ColorChannel>()*.

```
glColor4f(vecPlaneColor.GetNormalizedRed(), vecPlaneColor.
   GetNormalizedGreen(), vecPlaneColor.GetNormalizedBlue(),
   vecPlaneColor.GetNormalizedAlpha());
```

OpenGL supports several basic graphics primitives by default. For the plane a quad is required that is achieved by the following code [7]:

```
glBegin(GL_QUADS);
      glNormal3f(0, 0, 1);
      glVertex3f(-1, -1, 0);
      glVertex3f( 1, -1, 0);
      glVertex3f( 1, 1, 0);
      glVertex3f(-1, 1, 0);
glEnd();
```

In fact, the plane is infinite, but for the purpose as a visual helper it is displayed as a square with side length two. Note that the plane is also scalable to account for meshes of various sizes.

Figure 4.1 shows a few examples of possible plane positions.

4.2 Splitting the mesh

The splitting offset is again implemented as an interface parameter. If the parameter is set to *0.0* the mesh does not seem to be split. The larger the value of the offset the bigger the distance between the two halves of the mesh.

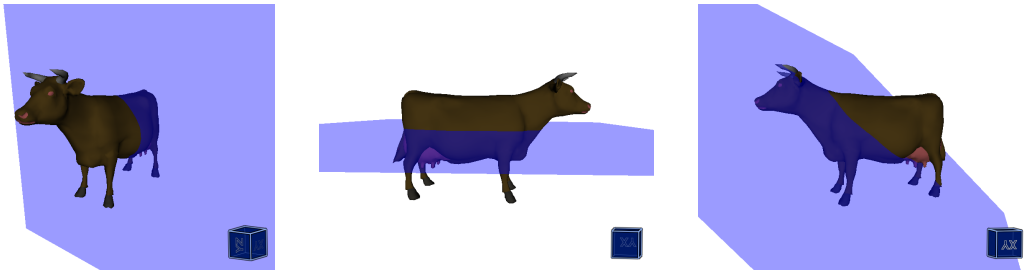

Figure 4.1: Example of a plane with different rotation vectors. From left to right: (0,1,0), angle = 90; (1,0,0), angle = 90; (1,0,1), angle = 90

```
GetPlugin().GetProperty("Offset").require(Variant::TypeFloat
    (0.5f));
```

Note that an observer needs to be added as well (cf. Chapter 4.1 and Chapter 3.1).

Rotation of the planes' normal vector

The normal of the plane has been defined as *(0.0, 0.0, 1.0)*, but regarding that the normal vector is still located in the object space of the plane, it needs to be transformed into the same space as the mesh. This is being done by rotating the normal by the same amount as the plane is rotated in the interface.
The following code reveals the rotation matrix regarding the interface inputs.

```
glLoadIdentity();
glRotatef(vecPlaneRotationAngle, vecPlaneRotationVector.GetX(),
    vecPlaneRotationVector.GetY(), vecPlaneRotationVector.GetZ
    ());
```

The rotation matrix could also be calculated using the generally known formulas [7]:

x-roll:

$$R_x(\beta) = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos(\beta) & -\sin(\beta) & 0 \\ 0 & \sin(\beta) & \cos(\beta) & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{4.1}$$

y-roll:

$$R_y(\beta) = \begin{pmatrix} \cos(\beta) & 0 & \sin(\beta) & 0 \\ 0 & 1 & 0 & 0 \\ -\sin(\beta) & 0 & \cos(\beta) & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{4.2}$$

z-roll:

$$R_z(\beta) = \begin{pmatrix} \cos(\beta) & -\sin(\beta) & 0 & 0 \\ \sin(\beta) & \cos(\beta) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{4.3}$$

However, it should be considered that 3D rotation matrices do not commute [7], so the order of multiplication matters. For details on matrix multiplication and applying affine transformations please look up the referenced literature [6][7].
Subsequently, the rotated normal vector is multiplied with the model view matrix to get the vector in model view space before it is normalized. After normalization the normal vector has a value between 0 and 1.

```
planeNormal.normalize();
```

Translating the mesh

The mesh is translated by the value of the offset in the direction of the normal vector. The command in OpenGL for translation is *glTranslatef*. Before the mesh is rendered the shader needs to be bound.

```
Vector meshTranslation = planeNormal * offset;
glTranslatef(meshTranslation.GetX(), meshTranslation.GetY(),
   meshTranslation.GetZ());
m_shaShader.bind();
renderMesh(*pMesh);
m_shaShader.release();
```

As stated above the mesh is rendered twice to create the illusion of a mesh that is dragged apart. Therefore, the mesh needs to be rendered again with a translation in the opposite direction.
In VolumeShop, the two meshes are displayed with a plane in between. The distance of the meshes is twice the offset (cf. Figure 4.2).

Discarding dispensable fragments

To eliminate the pixels from the other side of the plane of each mesh respectively, it has to be determined on which side of the plane a pixel lies. The computation is performed in the fragment shader.
TODO formula/code

The dot product of the plane normal and the vertex position minus a point on the plane states if the vertex is in front of or behind the plane.If the resulting value is smaller or equal to zero, the point is behind the plane and should not be drawn.

Before the mesh is rendered, the normal of the plane as well as a point on the plane must be

Figure 4.2: The mesh with no offset (left) and with offset applied (right)

passed to the shader as uniforms[1]. For the first half of the mesh the normal vector remains unchanged, for the second one converted. The point on the plane has to be determined in respect of the plane translation.

Definition of the plane normal and point on the plane as uniforms:

```
glUniform3f(m_shaShader.GetUniformLocation("planeNormal"),
   planeNormal.GetX(), planeNormal.GetY(), planeNormal.GetZ());
glUniform3f(m_shaShader.GetUniformLocation("planePoint"),
   vecPlaneTranslation.GetX(), vecPlaneTranslation.GetY(),
   vecPlaneTranslation.GetZ());
```

The fragment shader discards the points behind the plane:

```
float dotProduct = dot((vertexPosition - pointOnPlane),
   planeNormal);

if(dotProduct <= 0.0) {
      discard;
}
```

4.3 Shading the mesh

The back faces of the mesh still have the same color as the front faces, what makes it look unreal (cf. Figure 4.3). Therefore, the back faces have to be shaded in a different color. GLSL has an input variable *gl_FrontFacing* to check if the fragment is front facing. The following code shades all back facing vertices red:

```
if(!gl_FrontFacing){
```

[1] A uniform is a type designation. It is passed from the application to the shader. Its value in the shader is constant and never changes.

```
      gl_FragColor = vec4(1.0, 0.0, 0.0, 1.0)
}
```

Shading the back facing fragments

To add the impression of depth, lighting of the back facing fragments is being done using the Blinn-Phong shading model [6].

```
void directionalLight(in gl_LightSourceParameters light, in vec
   3 N, in vec3 V, in float shininess, inout vec4 ambient,
   inout vec4 diffuse, inout vec4 specular)
{
      vec3 L = normalize(light.position.xyz);

      float nDotL = dot(N, L);

      if (nDotL > 0.0)
      {
            vec3 H = normalize(light.halfVector.xyz);

            float pf = pow(max(dot(N,H), 0.0), shininess);

            diffuse += light.diffuse * nDotL;
            specular += light.specular * pf;
      }

      ambient += light.ambient;
}

void main()
{
      vec3 v = normalize(vVertexPosition);
      vec3 n = normalize(vVertexNormal);

      if(!gl_FrontFacing) {
            directionalLight(gl_LightSource[0], -n, v, 0.7,
               ambient, diffuse, specular);
            color.rgb = ((ambient * vec4(1.0, 0.1, 0.0, 1.0)) +
                (diffuse * vec4(1.0, 0.1, 0.0, 1.0)) + (
               specular * vec4(1.0, 1.0, 1.0, 1.0))).rgb;
            color.a = 1.0;
            color = clamp(color, 0.0, 1.0);
      }
      gl_FragColor = color;
```

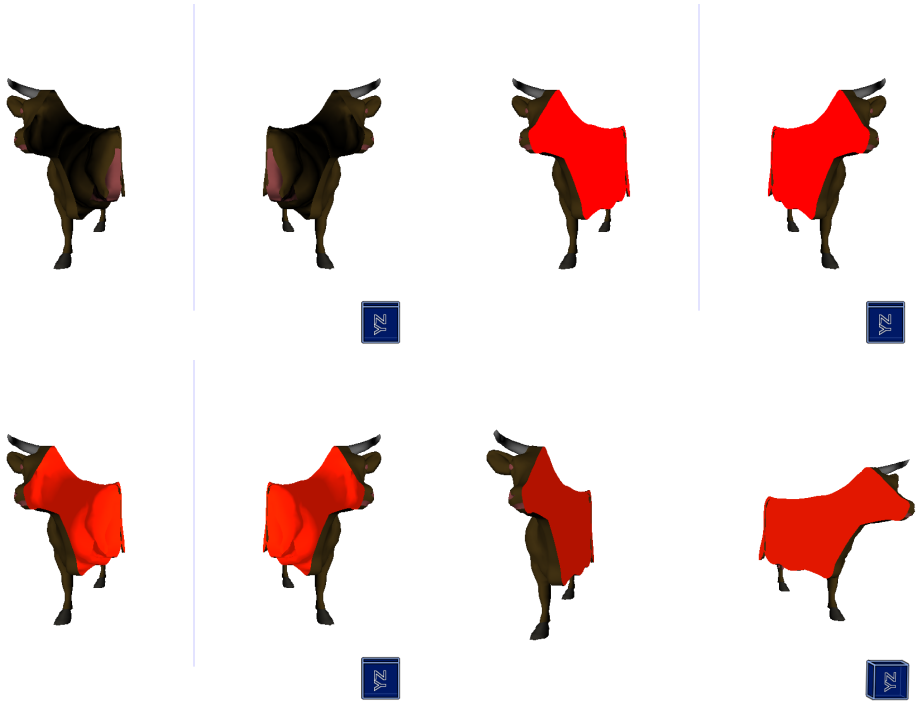

Figure 4.3: Different methods of shading the back faces (from upper left to lower right): no special treatment, flat shading, caved interior shading, cut surface shading

```
}
```

The result is a nicly shaded inside of the mesh with a hollow appearence (cf. Figure 4.3).

Shading the cut surface

"Cut surfaces are interior surfaces of solid objects that are made visible by the cutaway action." [4]
To smoothly shade the cut surface, the normals of the visible back-faces of the mesh are replaced by the inverse normal vector of the plane. With this approach, the shading is dependent on the viewing direction, so the reflection on the cut surface changes when the model is rotated.
For the implementation of shading cut surfaces, the Blinn-Phong shading model is used. The difference to the approach stated above is simply calling the lighting function with a different normal.

```
n = vPlaneNormal;
```

The result looks like in figure 4.3. The illustration shows a slightly rotated model to point out that the light changes with the viewing perspective, so the left half appears less illuminated than the right half.

TODO make newlines after code lines consistent

CHAPTER 5

Conclusion

The revelation of occluded objects in order for examination is a complex challenge. In my thesis I presented several approaches that provide different solutions for revealing regions of interest within complex 3D data. Solutions for both volume data and polygonal meshes were adduced with techniques for ghosting, cutaways and exploded views. Furthermore, VolumeShop, a developer tool for visualization research has been described. Its main advantage is the flexible, adaptive and plug-in based concept that lets users develop the optimal solution for their individual needs. Finally, I demonstrated the implementation of a simple algorithm to split meshes in VolumeShop. My implementation of splitting a mesh combined several existing approaches. The cutaway was covered by adjusting a plane to define the prospective cut and combined with a technique similar to an exploded view, implemented with a property in the user interface to adjust the degree-of-explosion. Moreover, an appropriate shading for the revealed structure can be chosen, depending on individual requirements.
The main advantage of my algorithm is that regardless of the complexity of the mesh, it has a stable performance as the objectionable fragments are simply discarded in the shader.

TODO add disadvantage: the mesh has to be rendered twice. A discussion on the possible use cases where your method is superior to for instance object-space splitting would be very appropriate.

Bibliography

[1] S. Bruckner. http://www.cg.tuwien.ac.at/volumeshop. Accessed: 2013-04-01.

[2] S. Bruckner and M. E. Gröller. VolumeShop: An Interactive System for Direct Volume Illustration. In *Proceedings of IEEE Visualization 2005, Oct, Minneapolis, USA*, pages 671–678, 2005.

[3] S. Bruckner and M. E. Gröller. Exploded Views for Volume Data. *IEEE Transactions on Visualization and Computer Graphics*, 12(5):1077–1084, 2006.

[4] M. Burns and A. Finkelstein. Adaptive cutaways for comprehensible rendering of polygonal scenes. *ACM Transactions on Graphics*, 27(5):154:1–154:7, 2008.

[5] C. D. Correa. Visualizing what lies inside. *SIGGRAPH Computer Graphics*, 43(2), 2009.

[6] D. Hearn and M. P. Baker. *Computer Graphics with OpenGL*. Pearson Education Inc., 3. edition, 2003.

[7] F. S. Hill Jr. and S. M. Kelley. *Computer Graphics Using OpenGL*. Pearson Education Inc., 3. edition, 2010.

[8] S. Knödel, M. Hachet, and P. Guitton. Interactive Generation and Modification of Cutaway Illustrations for Polygonal Models. In A. Butz, B. Fisher, M. Christie, A. Krüger, P. Olivier, and R. Therón, editors, *Smart Graphics*, pages 140–151. Springer Berlin Heidelberg, 2009.

[9] W. Li, M. Agrawala, B. Curless, and D. Salesin. Automated generation of interactive 3D exploded view diagrams. *ACM Transactions on Graphics*, 27(3), 2008.

[10] W. Li, L. Ritter, M. Agrawala, B. Curless, and D. Salesin. Interactive cutaway illustrations of complex 3D models. *ACM Transactions on Graphics*, 26(3), 2007.

[11] S. Sigg, R. Fuchs, R. Carnecky, and R. Peikert. Intelligent cutaway illustrations. In *Pacific Visualization Symposium (PacificVis), 2012 IEEE*, pages 185–192, 2012.