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, MSc

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, MSc

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)

Contents

CHAPTER 1

Introduction

The main purpose of 3D computergraphics is to represent image data. To make these data examinable, user interaction becomes an important part. Hence, the user should be able to translate, rotate and scale the 3D object. With the complexity of the object, the desire to examine just parts of it rises. The examination and analysis of the objects' inner structures can also be of interest. That leads to the a complex problem that needs to be solved: How can we reveal the inner sturctures? By simply removing the outer structures, the context would also get lost that could be important for scientific findings. Hence, the idea is to reveal the point of interest whilst keeping the context. Therefor, several scientific approaches exist. One idea is to simply raise the opacity of the outer structures. For example to reveal a brain, the approach would be to simply raise the opacity of the scull and the skin respectively for making the parts of secondary interest semi-transparent. Another approach is to cut out parts of the outer structure of the object so that no information gets lost and still the inside would be fully visible. A different approach are so called explosion views that unfold the inside by breaking the outer struktures into multiple parts and shifting those apart whilst keeping the point of interest in the center.

This paper will examine several approaches to reveal inner structures of complex 3D models retaining the context to be able to examine the object in respect of its surroundings.
TOEDIT In section BLA you will learn how to BLA and BLA BLA

CHAPTER 2

Analysis of existing approaches

Exploring volume data 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* (DOI) [10]. A high DOI means a region is of high interest, a low DOI stands for a region of secondary interest.
There are tree main methods for revealing inner structures of an object:

- Transparency/Ghosted views
- Cut-away views
- Explosion views

2.1 Transparency/Ghosted views

These techniques do not discard any data points 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 [4].
Ghosting is often used in combination with cut-away views. This would mean that the region cut out is replaced by faded duplicate of the original. Features such as edges are attempted to be preserved. Therefor the *ghost* stands for the original region before cutting [1].
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 [9].
Bruckner and Gröller [1] 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 will be 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 [1] [3] [4] [7]. 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 [9].

Li et al. [9] 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 handeled carfully that the user can mentally reconstruct the missing geometry. Also, the cut is being made 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.
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, muliple 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 archieved 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 pre-elected by the user. Therfor the user selects a set of targets from a list. The system then determines the cutting parameters as well as a viewpoint. This works by fully expanding all parts of the model space cutting volume that are above a target structure in the occlusion graph and the parts below the target structure as well as the cuts for 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 [3]. Their method allows interactive rendering of adaptive cutaways. It ensures that objects of interest are not obscured by objects of secondary importance (secondary objects). Depending on the position and the projection of the camera the cataways have to be adapted to guarantee free sight of the objects of interest.
The main part of the work is the depth image cutaway respresentation. It is based on an aproximation of the chamfer distance transform algorithm. The depth values of the rendered back hulls of the objects of interests is used as in input. The final result is a depth image containing the objects of interest and addition „drill holes“ around them with a certain slope depending on the camera position and projection.
The depth buffer is used for an addional depth check in the fragment shader, unfortunately the OpenGL pipeline only supports one depth buffer. If a fragement of a secondary object has a depth value smaller than the depth value in the cutaway depth image it is discarded because it would interfere with objects of interest. As a last step the objects of interst are rendered without checking against the cutaway depth buffer.
The cutaway depth image must be computed every frame to ensure interactive realtime 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 geometry is clipped the secondary objects would have a hollow appearance. To avoid this, fragments of backfacing polygons of the secondary objects refer to the normals of the cutaways shape. The result is the appearance of 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 [10]. BLABLA

Knödel et al. present an approach with strong focus on user-modified cutaways [7]. BLABLA BLABLA.

2.3 Explosion views

A system for creating and viewing interactive exploded views of complex 3D models is introduced by Li et al [8]. BLA

Bruckner and Gröller concentrate on avoiding occlusion when rendering 3D information in their work about „Exploded Views for Volume Data“ [2]. BLABLA.

CHAPTER 3

Methodology

3.1 Plug-ins in VolumeShop

VolumeShop is an interactive hardware-accelerated application for direct volume illustration [1]. It is designed for developers to have maximum flexibility for visualization research. Its objects can be dynamically created and accessed [11].

Plug-ins are functionally independent and can be dynamically loaded. One main advantage in development is that the application does not need to be closed when a plug-in is recompiled.

Properties

The complete state of a plug-in is defined by its properties which constitute the plug-ins' functionality [11]. A plug-in can be easily created with the following command.
An example for an integer property in the range [0,255]:

```
GetPlugin().GetProperty("Test2") = 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 GUI: [FIGURE] Creating links programmatically:

```
// 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 [11].
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();
}
```

For this work the display() function is of most importance. It is responsible for the rendering.

Languages

The plug-ins are written in C++ using OpenGL that is a successful cross-platform graphics application programming interface (API) for 2D and 3D computer graphics [6].
For shading and texturing OpenGL Shading Language (GLSL) is used [11].

3.2 Concept of Shaders

In GLSL there are four different 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.

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 [5]. With GLSL the shading of the polygons can be modified directly, replacing the default shading function of OpenGL.

Vertex shader

The vertex data is taken as input. A single vertex can consist of several attributes include position, color and normals.
The vertex shader can perform tasks such as [6]:

- transforming vertex positions
- transforming the normal vectors and normalizing them
- generating and transforming texture coordinates
- applying light models such as ambient, diffuse and specular per vertex
- computing color

Fragment shader

After the vertices have been transformed into the viewplane they are rasterized. The result are fragments which contain information about screen coordinates, depth, color, texture coordinates and so on. The value of the pixel color is determined by interpolation of the vertex colors.
The fragment shader can perform tasks such as [6]:

- applying light values
- computing shadows
- adding complex texture(e.g. Bump Mapping)

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 halfs 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 halfs, but the model is rendered twice at different positions in space parallel to the plane. The final step is shading of the models' surface as well as the back facing triangles. The latter will be shaded with a different color, but with the same amount of lighting to create the illusion of depth.

4.1 Definition of the plane

A plane in this context is a square of infinite size. It is used as a visual help to define where the mesh will be cut.
To be able to intervene with VolumeShop, it is necessary to provide properties to set the translation, rotation, and scale vectors as well as the color. Each property requires a name and a type. Here an example of a property named „Plane Translation Vector" of the type Vector (in this case vec3):

```
GetPlugin().GetProperty("Plane Translation Vector").require(
   Variant::TypeVector(Vector(0.0f, 0.0f, 0.0f)));
```

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 tranforming the coordinates from world space into viewing space.[BOOK: Hearn, Baker]
Then the plane is being adjusted by its affine transformations. All passed parameters 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());
```

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 archieved 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 looks like the following [6]:

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

As stated above our plane will be indefinite, but for the purpose as a visual helper it is displayed from -1 to 1 as a default. Note that the plane is also scaleable for bigger or smaller meshes.

Figure 4.1 shows a few examples of possible plane positions.

4.2 Splitting the mesh

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

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

Note that of course an observer needs to be added as well (see 4.1 and 3.1).

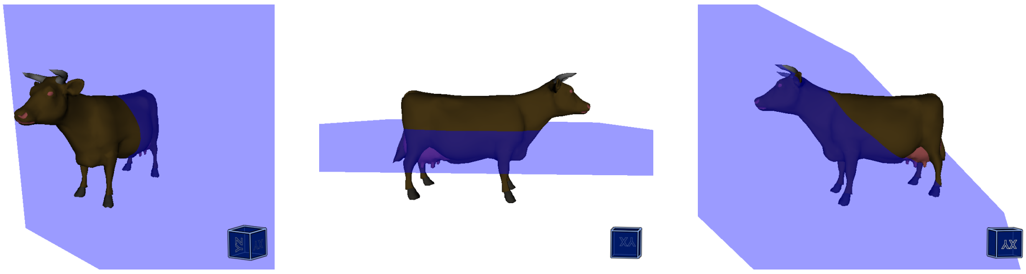

Figure 4.1: Example of a plane with different rotation vectors. From left to right: (0,1,0), angle = 90; (1,0,0), angle = 90; (1,0,1), angle = 90

Rotation of the planes' normal vector

The normal of the plane has been defined with '(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
   ());
```

Of course, the rotation matix could also be calculated using the generally known formulas [6]:

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:

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

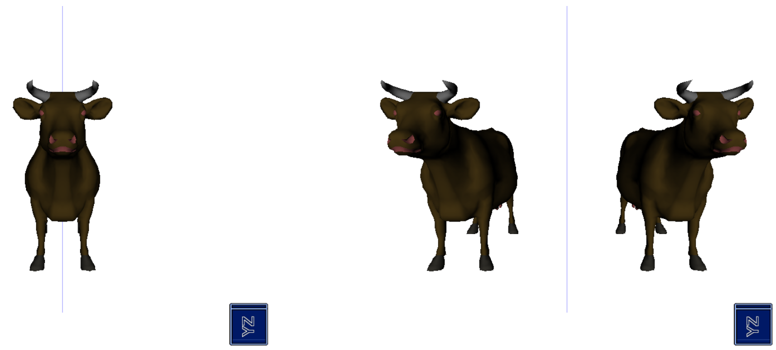

Figure 4.2: The mesh with no offset (left) and with offset applied (right)

However, it should be considered that 3D rotation matrices do not commute [6], so the order of multiplication matters. For details on matrix multiplication and applying affine transformations please look up the referenced literature. [6] [5]
Next the rotated normal vector should be multiplied with the modelview matrix to get the vector in modelview space before the it is normalized. After normalization the normal vector has a value between 0 and 1.

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

Translating the mesh

After the light calculations 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. So the mesh needs to be rendered again with a translation in the opposite direction.
In VolumeShop, the two meshes displayed with a plane in between. The distance of the meshes is twice the offset (see 4.2).

Discarding dispensable pixel

To eliminte 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.
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 passed to the shader as uniforms. For the first half of the mesh the normal vector is untouched, 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. Therefor the back faces have to be shaded in a different color. OpenGL has a function to check if the fragment is front facing:

```
if(!gl_FrontFacing){
      gl_FragColor = vec4(1.0, 0.0, 0.0, 1.0) //shades all
         inner facing vertices red
}
```

Using the phong shading model for the back facing fragments

To add the impression of depth, lighting of the back facing fragments is being done with the phong shading model [5].

```
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;
}
```

The result is a nicly shaded inside of the mesh (see Figure (see 4.3).

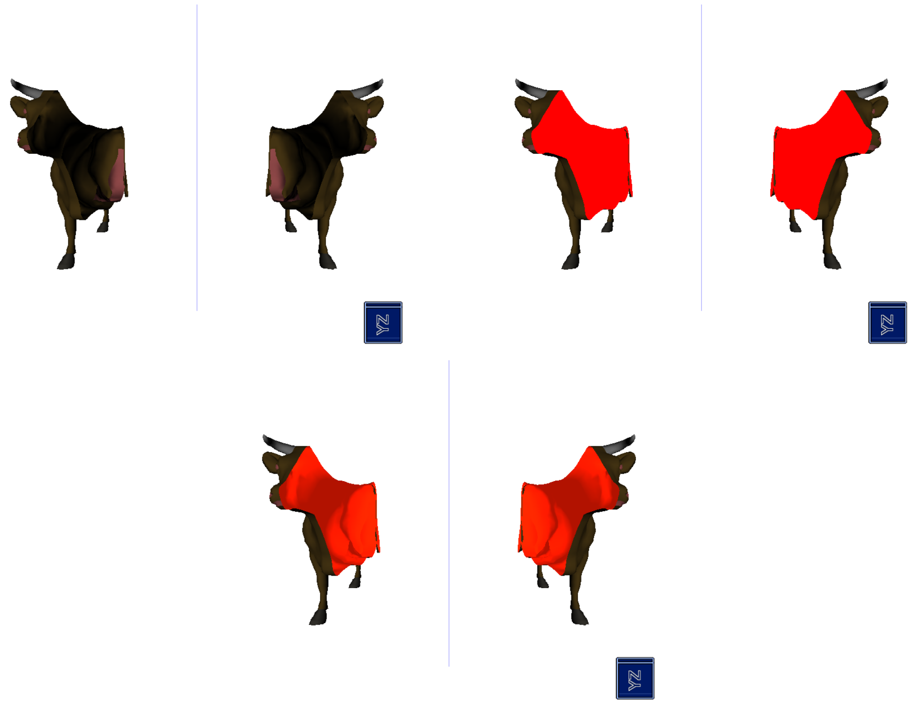

Figure 4.3: Different methods of shading the back faces: no special treatment (left), flat shading (middle) and phong shading (right)

CHAPTER 5

Critical reflection

CHAPTER 6

Summary

Bibliography

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

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

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

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

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

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

[7] 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.

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

[9] 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.

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

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