

**APERIO: MANAGING 3D SCENE OCCLUSION USING A MECHANICAL
TOOL ANALOGY FOR VISUALIZING MULTI-PART MESH DATA**

by

David Tran, BSc, Ryerson University, Toronto, Ontario, 2011

A thesis

presented to Ryerson University

in partial fulfillment of the

requirements for the degree of

Master of Science

in the Program of

Computer Science

Toronto, Ontario, Canada, 2015

© David Tran 2015

AUTHOR'S DECLARATION

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I authorize Ryerson University to lend this thesis to other institutions or individuals for the purpose of scholarly research.

I further authorize Ryerson University to reproduce this thesis by photocopying or by other means, in total or in part, at the request of other institutions or individuals for the purpose of scholarly research.

I understand that my thesis may be made electronically available to the public.

APERIO: MANAGING 3D SCENE OCCLUSION USING A MECHANICAL TOOL ANALOGY FOR VISUALIZING MULTI-PART MESH DATA

David Tran

MSc, Computer Science, Ryerson University, 2015

ABSTRACT

This thesis presents a novel interaction model for browsing complex 3D scenes containing numerous layers of occluding and intertwining structures that often hide regions of interest. The interaction model is realized through the development of a custom visualization application, *Aperio*. *Aperio* provides a set of virtual mechanical “metal” tools, such as rods, rings, “cookie” cutters and a knife, that support real-time, interactive exploration. Cutter tools are designed to create easily-understood cutaway views (or context-preserving ribbon slices) and rings and rods provide simple path constraints that support rigid transformations of models via “sliding”, providing interactive exploded-view capabilities. All tools are based on a single underlying superquadric formulation and can “iteratively” be picked up and replanted to generate various views. A multi-pass, GPU-based capping algorithm provides real-time “solid cuts” rendering of surface meshes. We also present a user study to provide supporting evidence of *Aperio*’s interaction simplicity and effectiveness for occlusion management.

ACKNOWLEDGEMENTS

I want to thank my family and friends for their tremendous support throughout these past few years; thank you to my good friends in the lab and the GMU, who have continually provided moral support and fun conversations during the late nights spent coding. I also have a lot of gratitude for my mom and dad, having to deal with many curveballs thrown at them these past few years and having to constantly adapt; they've provided me a lot of support even when I didn't know if I could continue, and remained strong through it all. But also, I want to give an enormous thank you to Dr. Tim McInerney for never giving up on me, for supporting me and for all the vast amount of knowledge he has given me throughout these years. I definitely could not have done it without his guidance and help. I also want to give a warm thank you to anyone else taking the time out of their busy lives to read this thesis. ☺

TABLE OF CONTENTS

ABSTRACT.....	iii
ACKNOWLEDGEMENTS	iv
LIST OF FIGURES	viii
1 Introduction.....	1
1.1 Contributions.....	5
1.2 Thesis Outline	7
2 Literature Survey	10
2.1 Cut-aways	12
2.1.1 ViC (Virtual Cadaver) and RibbonView	12
2.1.2 ZygoteBody.....	14
2.1.3 Interactive Cut-away Illustrations.....	15
2.1.4 Gimlenses.....	17
2.1.5 2.5D Clip Surfaces	20
2.1.6 Adaptive Cut-Aways.....	21
2.1.7 Automated Cutaways for Flow Illustration.....	22
2.2 Transparency and Ghosted Views	23
2.2.1 Interactive Ghosted Views	25
2.3 Explosions.....	25
2.3.1 Expanding Annotations with Explosion Probe	26
2.3.2 Compact Explosion Diagrams	28
2.3.3 HowThingsWork: Autogenerating Assembly Visualizations.....	29
2.3.4 View-Dependent Explosions	29

2.4	Magic Lenses	31
2.5	Deformations.....	32
2.5.1	Illustrative Deformation.....	32
2.5.2	View-Dependent Peel-Aways.....	35
2.6	Elastic Band Metaphor.....	37
3	Methodology and Implementation	40
3.1	Implementation	41
3.1.1	VTK	44
3.1.2	Multi-pass Rendering.....	45
3.2	Superquadrics.....	47
3.3	Constructive Solid Geometry.....	51
3.3.1	Carve CSG	53
3.4	Cutter.....	57
3.4.1	MeshMixer.....	58
3.5	Capping Algorithm	60
3.5.1	OpenCSG	60
3.5.2	Custom Capping.....	61
3.6	Real-time Cut Previewer.....	64
3.7	Explosion on a Constrained Path	67
3.7.1	Ring.....	69
3.7.2	Rod.....	72
3.8	Tool Appearance.....	72
3.9	Ribbon Cuts Algorithm.....	75

3.10	Knife Tool.....	77
3.11	Control Panel and Tooltip Interaction.....	80
4	Evaluation and Results	81
4.1	User Study.....	81
4.1.1	Process	83
4.1.2	Results.....	85
5	Conclusion and Future work	88
5.1	Conclusion	88
5.2	Future work.....	91
	Appendices.....	94
	Appendix A: Aperio Quick Reference (Controls)	94
	Appendix B: User Study Questions	95
	References	99

LIST OF FIGURES

Figure 1: Cutaway, transparency and explosion examples. [2–4]	1
Figure 2: Unreal Engine 4 demo on NVIDIA Tegra K1 tablet. [6]	2
Figure 3: Twisting and peeling examples. [8]	3
Figure 4: Examples of varying superquadric shapes. [10]	4
Figure 5: Aperio’s real-time cookie-cutter preview into heart.	5
Figure 6: Constrained explosion path defined by superquadric ring or rod.	6
Figure 7: Deformations of superquadrics (twist, bend, taper). [9]	6
Figure 8: Aperio’s real-time, superquadric-contained ribbon cuts.	7
Figure 9: Volumetric and surface data (left and right respectively). [26]	10
Figure 10: CT scan, image slices of a brain. [27]	11
Figure 11: ViC’s fragmentation and cut-away phases. [31]	13
Figure 12: RibbonView’s outline preservation using ribbon slices. [11]	13
Figure 13: ZygoteBody; changing transparency using horizontal and vertical sliders. [32]	14
Figure 14: Interactive cutaways defined by parameterization of parts. [15]	15
Figure 15: Box, tube, wedge and window cuts. [15]	16
Figure 16: Automatically generated cutaway views of disk brake model. [15]	17
Figure 17: Gimlenses; drilling into a car engine. [13]	18
Figure 18: Front, side and top orthographic and perspective views. (Blender and Maya).	18
Figure 19: GimLenses; object selector, lens proxy and cone-shaped cut. [13]	19
Figure 20: 2.5D clip-surface example demonstrating curving cuts. [16]	20
Figure 21: Adaptive cut-aways; galleon’s hull is cut away to reveal interior cannons. [17]	21
Figure 22: Ghosted view and faded ghosted view of stairs. [17]	22
Figure 23: Automated cutaways in flow visualizations. © 2012 IEEE [21]	23

Figure 24: Weighted Blended Order-Independent Transparency example. [38].....	24
Figure 25: Ghosted view of a car’s interior. [39]	25
Figure 26: Annotated explosion diagrams. [40]	26
Figure 27: Finding the centroid using a skeleton vs bounding box method. [40]	27
Figure 28: Compact explosion diagram; grouping identical pieces. [18].....	28
Figure 29: HowThingsWork; animated assembly diagram. [22]	29
Figure 30: Exploded view of a turtle. © 2006 IEEE [14].....	30
Figure 31: FlowLens; 3D lens displaying blood flow. © 2011 IEEE [41]	31
Figure 32: Magic Lenses; original data, magnification and fisheye lens. [23].....	31
Figure 33: Illustrative Deformation; peeling and twisting. [8]	32
Figure 34: CT scan of a mouse fetus; the head is occluded. [8]	33
Figure 35: Deformation template examples. [8].....	34
Figure 36: Peel deformation followed by a wave. [8]	34
Figure 37: Automatic placement of the deformation template. [12]	35
Figure 38: Rigid peel, cabinet peel and deformed peel examples. [12].....	36
Figure 39: Lifting and ungrouping an object. © 2014 IEEE [20].....	38
Figure 40: Gradually increasing repulsive force. © 2014 IEEE [20]	38
Figure 41: Contact relationships between parts shown as a graph. © 2014 IEEE [20]	39
Figure 42: Aperio’s cutter previewer; peering into a skull.	41
Figure 43: Example QT widget applications. [43]	42
Figure 44: VTK’s modern volume rendering subsystem (Composite, Maximum Intensity Projection, and Additive Blending methods). [44]	43
Figure 45: Blender’s Boolean Modifier example; intersecting a cube with a sphere.	44
Figure 46: VTK’s pipeline. [46]	44
Figure 47: Depth, positions and normals rendered to textures (FBOs). [48].....	46

Figure 48: Scene without and with SSAO. [48]	46
Figure 49: Superquadric shapes; altering Theta (θ) and Phi (ϕ) roundness (e, n parameters). [51]	47
Figure 50: Axial symmetry around the y-axis. [53]	49
Figure 51: Superellipsoid and supertoroid examples. [51]	50
Figure 52: CSG; (Cube \cup Sphere), (Cube $-$ Sphere) and (Cube \cap Sphere).	51
Figure 53: Water-tight mesh; outer shell with thickness example. [55]	52
Figure 54: Aperio's image-based preview and actual cut-away.	52
Figure 55: Carve; large model intersected with two shifted versions of itself (100k vertices). 8	54
Figure 56: CSG with mesh A and cutter B. (B \cup A), (A $-$ B) and (A \cap B) respectively.	55
Figure 57: Non-manifold geometry; edge sharing (0 faces), (1 face) and (3 faces) respectively.	56
Figure 58: Aperio's steps for performing cuts with CSG.	56
Figure 59: Aperio; ring and rod gliding on the surface of the mesh following surface normals.	57
Figure 60: Aperio; drilling into a heart (interactive cut previewer).	58
Figure 61: MeshMixer; dragging new objects into the scene and using brush select mode. [60]	59
Figure 62: OpenCSG; modeling a dice. [61]	60
Figure 63: Aperio; no capping (hollow) vs with capping algorithm (solid).	62
Figure 64: Selected mesh's front and backfacing depths, colour and cutter's backfacing depth.	62
Figure 65: Aperio; resulting image of capping algorithm.	64
Figure 66: Superquadric's orientation as defined by Right, Up and Forward vectors.	65
Figure 67: Aperio; hinging newly sliced piece of the left cerebral cortex using a ring.	67
Figure 68: Aperio; creating a constrained explosion for multiple meshes along a ring.	68

Figure 69: Aperio; placing a rod on the corpus callosum and spinning it.	69
Figure 70: Finding points for the sliding path on the supertoroid.	70
Figure 71: Using normals (\hat{n}), tangents (t) and bitangents (b) to orient meshes.....	71
Figure 72: Spreading meshes apart on a planted rod vs fanning meshes.	72
Figure 73: Matcaps; result of applying them to models in Luxology's Modo. [68].....	73
Figure 74: Matcaps applied to a ring and rod (metal and paint materials).	74
Figure 75: Aperio; entire cut-away vs ribbon cuts (into the kidney).....	75
Figure 76: Aperio; using a knife to cut the liver in half and hinge one half open with a ring.....	77
Figure 77: Aperio; tapered cutter blades (spreaders) used to widen the cut.	78
Figure 78: Aperio's control panel.	80
Figure 79: Aperio FreeForm (no tools) vs. Aperio Tools.....	81
Figure 80: User study's practice round dataset (heart).	83
Figure 81: User study's two additional rounds (brain and skeletal/intestinal datasets).....	84
Figure 82: User study's perception questions (FreeForm vs Tools).....	85
Figure 83: User study's user interface questions (FreeForm vs Tools).....	86

1 Introduction

Trying to visualize and gain insight into a complex 3D scene containing many layers of occluding¹ structures that intertwine and hide one another can be a difficult task. These systems usually consist of numerous parts, but often we are only interested in discerning the spatial relationships of a connected few. It can be a near impossible task to find what we are looking for when the scene is densely-packed and objects are hidden behind massive amounts of clutter. Although occlusion is an important factor for depth perception [1], as it gives us a cue of how far away objects are from us (being behind or in front), an abundance of occlusion can make it difficult for tasks involving discovery. Since the image the user sees is usually a limited 2D perspective projection of the scene, it can be difficult to gather a contextual understanding of where objects *really* lie spatially in relation to others in the field of view. Many visualization techniques have been developed over the years to deal with the problem of occlusion; some of the most common techniques include cut-aways, transparency and explosions (**Figure 1**).

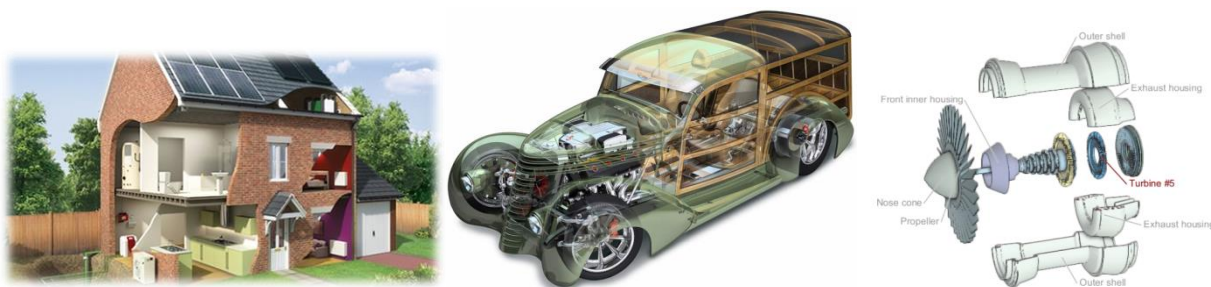


Figure 1: Cutaway, transparency and explosion examples. [2–4]

With the recent surge in development of Graphics Processing Units (GPUs), efficient rendering of very large and complex datasets is now possible; even mobile devices such as phones and tablets have evolved to the extent of mid-range desktops, with the NVIDIA Tegra K1 tablet

¹ Occlude means to block or obstruct in computer graphics (Merriam-Webster Dictionary)

recently being the first mobile processor capable of running OpenGL 4.4/Compute Shaders/CUDA applications and even Unreal Engine 4 [5] (**Figure 2**). This means that we are now capable of visualizing much larger and more detailed datasets than ever before, making occlusion management techniques an important point of discussion.



Figure 2: Unreal Engine 4 demo on NVIDIA Tegra K1 tablet. [6]

The three techniques mentioned above for managing occlusion, (cut-aways, transparency and explosions, in addition to others) are ideally useful for gaining insight into complex datasets such as detailed human anatomy models, industrial engines of automobiles, mechanical assemblies and architectural/urban datasets or even in the field of aerospace engineering where taking apart detailed CAD models of airliners can be useful for education and design.

The first technique, cut-aways, simply removes occluding structures from view; it is effective at solving the occlusion problem, but contextual information (i.e. the outline of where structures were placed originally in the scene) is lost [7]. Transparency is useful for solving occlusion by making occluding objects partially transparent so that objects behind can show through; unfortunately, object depths can be difficult to distinguish among many layers of transparency. Explosions are useful for revealing hidden objects in the scene by radially moving occluding parts out of the way, but often do not allow the user much control (since parts are translated in

every possible direction) and these parts can continually occlude other objects in the scene while they are shifting outwards.

More recent techniques termed *Illustrative Deformation* by Correa [8] include using deformation techniques to physically manipulate the geometry of objects such as peeling, twisting, bending, and splitting objects, etc. (**Figure 3**). Most of these techniques can be performed on the GPU using vertex and geometry shaders for efficiently rendering transformed geometry but some large deformations and cut surfaces may require remeshing [8]. Since these techniques are inspired by scientific illustrations from textbooks, the visualizations need not conform to reality. Users can perform actions not possible in real life, allowing them to generate creative views of the data.

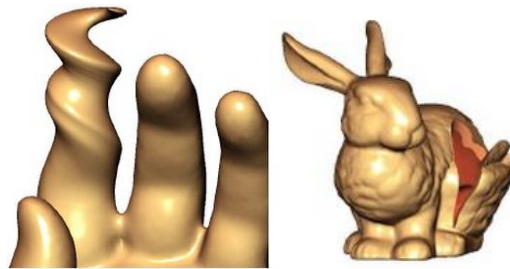


Figure 3: Twisting and peeling examples. [8]

Usually a deformation template must first be placed onto the object to act as a mask defining which parts of the object can be deformed in the visualization. Creation and placement of this template in 3D space can be a difficult task for the user, especially as most common input devices are still 2D in nature. With the advent of multi-touch devices, and with tablets becoming common among the general public, users are now given more degrees of freedom (DOF) in 2D space, allowing them to use multiple-finger gestures such as two-finger pinch zooms and using their opposable thumb as a pivot with the index finger for intuitive rotation control.

Although multi-touch devices give users freedom to directly manipulate objects on-screen, they introduce finger-occlusion problems and other problems such as precision and muscle fatigue.

In this thesis, we use a mechanical analogy for actions that can be performed in our visualization application, *Aperio*. *Aperio* is a visualization system designed to provide users the capability to reveal occluded structures in a complex 3D scene by performing mechanical actions such as slicing with a cookie-cutter, making ribbon-sliced cuts, hinging parts apart, exploding objects along a constrained path (on a ring or rod) and making incisions using a knife. Most of these techniques are performed using superquadric-shaped tools. We chose to use superquadrics because they are flexible in terms of the wide range of shapes that can be created (rounded and non-rounded cubes, boxes, spheres, cylinders, capsules, diamonds, wheels, supertoroids, etc.) as shown in **Figure 4**. They have good shape coverage but are still constrained enough making them suitable for defining simple shapes and explosion path using only a small number of controlling parameters [9].

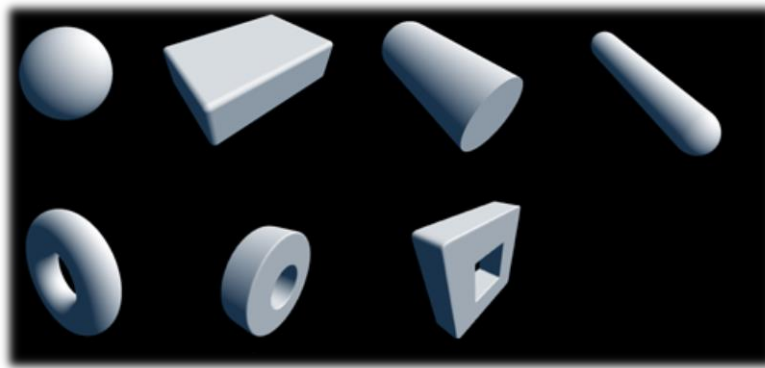


Figure 4: Examples of varying superquadric shapes. [10]

1.1 Contributions

Our primary goal was to create a system that supported both fast data exploration and effective occlusion management. Furthermore, the system should balance the use of constraints with ease of use. In an effort to meet this goal, we have made the following contributions:

i) We have developed a visualization system that provides the capability to reveal occluded structures using a fluid interaction model with mechanical tools; users are able to iteratively and immediately perform actions on the model such as slicing with a cookie-cutter, making ribbon-slices, hinging parts apart, exploding objects along a constrained path (on a ring or rod) and making incisions using a knife. *Aperio* combines both authoring and browsing phases together (creation and exploration of visualizations), allowing users the capability to explore and manipulate the model at the same time supporting fluid and iterative generation of views. A cutter tool can slide along a surface mesh, generating a preview of the cut in real-time (**Figure 5**). The user can interactively explore the model without being restricted by an author’s pre-representational view of the data. Real-time interaction is achieved by developing a GPU-based mesh capping algorithm in a fragment shader. As the cutter slides along the surface of a polygonal data mesh and cuts it open, the algorithm fills the opening with the underside of the cutter’s surface, making the mesh appear solid.



Figure 5: Aperio’s real-time cookie-cutter preview into heart.

ii) This thesis also presents a novel approach for performing path-constrained explosions using rings and rods. Unlike traditional exploding diagrams, where parts move outwards radially with little control over the direction and radial displacement of individual parts, the exploder we introduce in this thesis supports control of each part's movement along a path defined by a superquadric parameterization of the ring or rod (**Figure 6**).



Figure 6: Constrained explosion path defined by superquadric ring or rod.

The ring is parameterized using a supertoroid (one of the possible superquadric shapes). Supertoroids are useful for generating explosion paths because they are flexible in the number of paths that can be created, especially when combined with global deformations (**Figure 7**), to create interesting bending, twisting or tapering paths.



Figure 7: Deformations of superquadrics (twist, bend, taper). [9]

Entire models or model pieces can simply slide along the supertoroid's path like beads on a bendable wire.

iii) We introduce a real-time, ribbon cut-previewer, based on McInerney and Crawford’s RibbonView[11] technique, where the ribbons are contained within the superquadric cutter. However, rather than just creating static ribbons to outline the surface of a cut-away mesh as in [11], Aperio supports real-time, dynamically-adjustable ribbon views. As the superquadric-based ribbon cutter slides over a data mesh, the GPU-based implementation continually generates the ribbons and users can instantly change ribbon orientation, frequency and width. (**Figure 8**).

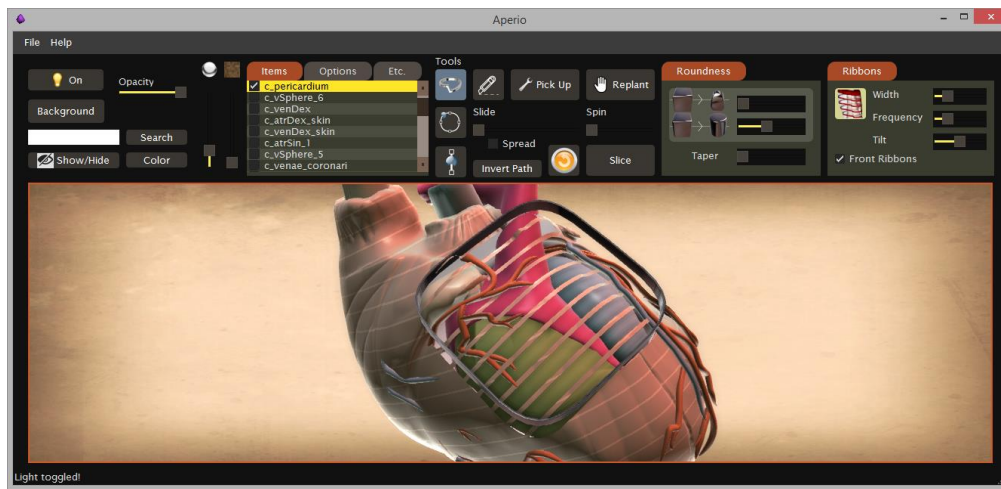


Figure 8: Aperio’s real-time, superquadric-contained ribbon cuts.

1.2 Thesis Outline

Chapter 2 explores existing traditional and more recent solutions to the problem of 3D scene occlusion. This includes various visualization approaches such as traditional cut-away, transparency and explosion techniques, Birkeland’s view-dependent peel-aways [12], Pindat’s multiple detailed-views of holes drilled with GimLenses [13], Bruckner’s view-dependent explosions [14], Correa’s illustrative deformation [8], Li’s generation of interactive cut-aways [15], Trapp’s 2.5D clip surfaces [16], Burn’s view-adaptive cut-aways [17], Tatzgern’s compact explosions on subsets of assemblies of the object [18], Cmolik’s ghosted view implementation

[19], Otsuki’s elastic band metaphor [20], Sigg’s cut-aways for flow visualization [21], Mitra’s HowThingsWork [22] visualization for generating animated motion on mechanical assemblies and Tominski’s survey on Magic Lenses [23].

Chapter 3 delves into the implementation of the application, *Aperio*; including the real-time capping algorithm implemented as an image-based algorithm in a multi-pass deferred rendering pipeline. It also talks about the use of constructive solid geometry and Carve CSG for performing cuts, the mathematics of superquadrics (implicit and parametric equations) along with its use in the cutting, hinging and explosion algorithms. In addition, it discusses shader techniques used to enhance depth and create the metallic appearance of the superquadric-shaped tools (screen-space ambient occlusion for depth, and spherical environment mapping for creating MatCap materials similar to materials in zBrush [24]). The inspiration for the surface normal-aligned orientation of tools in *Aperio* is also discussed along with an implementation of the rod/ring explosions and hinge. The GPU-based ribbon cuts algorithm is also explained.

Chapter 4 contains an evaluation of the 3D interaction model developed by discussing the process and results of a user study performed comparing *Aperio* to free-form techniques in terms of ease of use, intuitiveness (visual affordance) and effectiveness for occlusion management. Users were also asked perception questions (how easy it was to restore positioning and orientation of objects, and to repeat and recreate visualizations, etc.) The learning curve of both systems is evaluated. Users then filled out a survey ranking statements on a 7-point likert scale based on their level of agreement with each statement and were asked for additional comments on which system they preferred.

Chapter 5 concludes the thesis by wrapping up and summarizing results of the user study, describing limitations of the interaction model developed and suggests future work for extensibility and improvement.

2 Literature Survey

With the sheer volume of data available to us today, along with rapid improvements in graphical algorithms and hardware, we are now more capable (than ever before), of rendering immensely detailed datasets. Successively, this means that acquiring useful insight from data would require more effective occlusion management techniques that adapt to the data's size and complexity. Many novel techniques have been developed over the years, building on top of more traditional techniques such as cut-aways, transparency and explosions; when combined with available contextual information, such as the user's viewing pose, mesh geometry/contours, or knowledge of the data's field and domain, we can create visualizations tailored to particular use cases. Visualizations are not meant to be static images and should be interactive, adapting to different situations; this is best summarized using the words of Jacques Bertin [25]:

“A graphic is not ‘drawn’ once and for all; it is ‘constructed’ and reconstructed until it reveals all the relationships constituted by the interplay of the data. The best graphic operations are those carried out by the decision-maker himself.”

This chapter will look at many novel visualization techniques developed over the past few years for managing 3D scene occlusion. The first step involves gathering data; data acquired for visualizations is usually obtained in one of two forms, as surfaces or as volumes (**Figure 9**).



Figure 9: Volumetric and surface data (left and right respectively). [26]

Surfaces are usually represented as polygonal meshes made up of vertices, edges and faces representing only the outer contours (essentially the surface) of an object. Volume data, on the other hand, require a lot more storage space as they contain information on the *inside* of an object; they are made up of voxels, (volumetric elements) which are basically pixels (picture elements) but in 3-dimensional space. They can be gathered from CT (Computed Tomography) or MRI (Magnetic Resonance Imaging) scans in the form of multiple image slices (**Figure 10**), similarly to slicing up a loaf of bread; each slice represents a cross-section of the volume data.

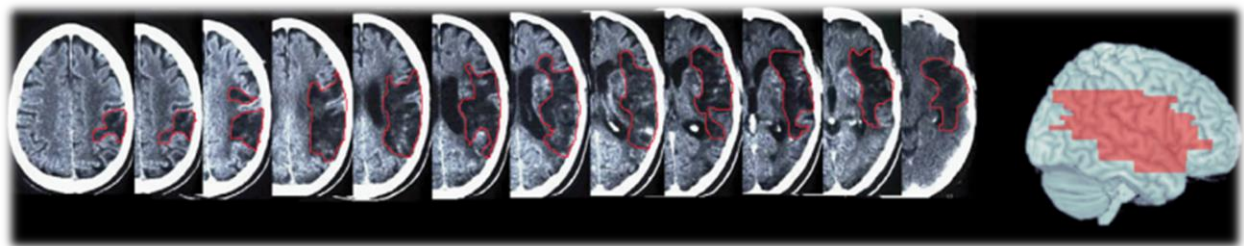


Figure 10: CT scan, image slices of a brain. [27]

In CT scans particularly, the colour of a pixel in each image slice represents the radiodensity of the material at that position (often measured in Hounsfield units). Bone would have the highest radiodensity (around 700-3000 Hounsfield units [28] and therefore its value would be highest or closest to white), whereas tissue would have a lower radiodensity somewhere around 100-300 HU (Hounsfield Units), followed by muscle and blood at 10-45 HU. Air would likely have the lowest radiodensity represented by a colour closest to black.

On the other hand, surface data can either be created manually using CAD software by skilled modellers or acquired through optical devices, such as laser-range scanners or structured light scanners. Data acquired through these devices are usually in the form of surface points (or point clouds) with each point optionally containing additional attributes such as point normals, colours

or material properties [29]. Surfaces can even be acquired from volumetric data (known as isosurfaces) by extracting voxels in the volume with a constant radiodensity value. Polygonal meshes can then be generated from the acquired isosurfaces and rendered to the screen efficiently. This thesis focuses particularly on techniques for managing 3D scene occlusion in polygonal surface mesh data.

2.1 Cut-aways

One traditional technique for managing scene occlusion is simply cutting away or removing occluding objects from view. As stated before, and by McGuffin [7], simply removing the object can make it difficult for users to form an integrated mental picture of the entire volume. This makes it difficult for users to gain a contextual understanding of how the object is connected with inner structures since the outer object is now visually removed. These problems fall into the category of Focus+Context since we want a detailed view of a particular region but at the same time, still want to maintain a contextual overview of the entire scene [30].

2.1.1 ViC (Virtual Cadaver) and RibbonView

ViC was an application developed by Yakobovich [31] that allowed fragments of the arm to be separated and cut away revealing anatomical structures underneath. It was implemented using Microsoft XNA, C# and Synapse's SunBurn Game Engine, and required a pre-processing phase where the arm is first broken up into smaller, more manageable mesh fragments (**Figure 11**). These mesh fragments can then be interactively removed during the browsing phase.

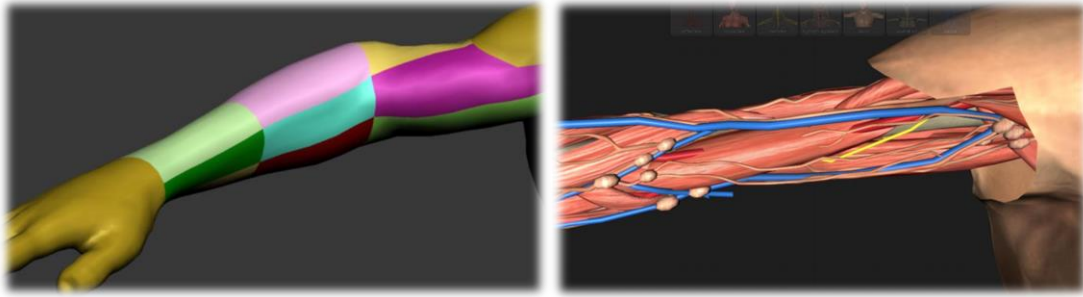


Figure 11: ViC’s fragmentation and cut-away phases. [31]

After preprocessing the data, (fragmentation phase), ViC allows these fragments to be removed revealing hidden structures underneath. The problem with this approach is that once fragments are removed, it can be difficult to imagine where they were placed initially in relation to inner organs, as previously stated. To improve upon the traditional cut-away technique, McInerney and Crawford [11] developed another context-preserving cutaway technique, RibbonView. Rather than removing the entire object from view, an outline is preserved consisting of polygonal strips or ribbons which outline the removed material (**Figure 12**).

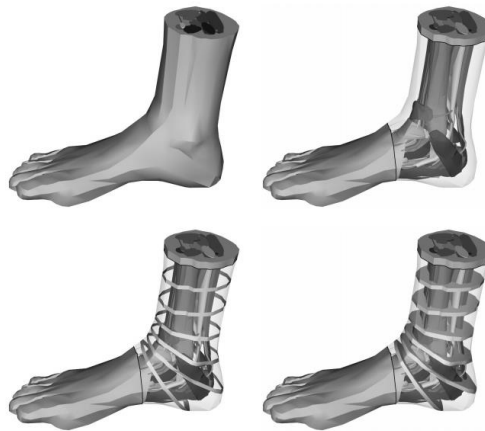


Figure 12: RibbonView’s outline preservation using ribbon slices. [11]

This approach preserves an outline of the removed fragment, allowing the user to mentally and visually reconstruct the original surface while still being able to see the interior. ViC is similar to many visualization packages, in that there is an “authoring” phase for pre-processing the data

first and then a “browsing” phase for visualizing the result. One of the disadvantages to this approach is that users are not able to interactively explore the data in real-time since an authoring phase is required and the resulting visualizations are limited by the author’s view or custom fragmentation of the data.

2.1.2 *ZygoteBody*

ZygoteBody [32] is a similar application to ViC in that it allows exploration of a human anatomy but rather than using fragmentation and cut-aways to manage occlusion, it uses translucency to remove entire layers of the body that occlude regions of interest (**Figure 13**). It is implemented using WebGL and is designed to facilitate human anatomy learning for medical students.

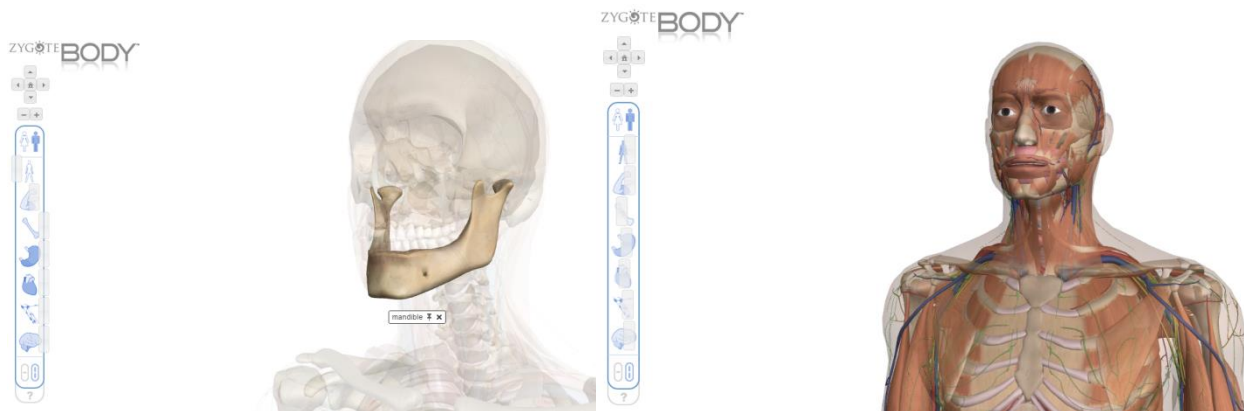


Figure 13: ZygoteBody; changing transparency using horizontal and vertical sliders. [32]

Users can explore the model by dragging the sliders on the left, which control the translucency for each layer of the human body. Layers that can be controlled include skin, muscles, bone, vital organs, veins and arteries, lymph nodes and the nervous system. You can select a male or female body, rotate, zoom and pan similarly to most common visualization packages. You can also focus in on a particular mesh which reduces the opacity of every other mesh in the scene. The

problem with transparency-based techniques is that it quickly becomes ineffective as more layers are added, since it becomes difficult to distinguish between numerous layers when they overlap one another. Recognizing spatial relationships among many partially transparent layers can be a challenging task, especially when layers intertwine and cross over into other layers.

2.1.3 *Interactive Cut-away Illustrations*

Many visualization systems require an authoring stage and a browsing stage; the authoring phase usually requires a skilled 3D modeller familiar with the dataset’s domain. Preparing the data for visualizations can be a tedious task. To resolve this, Li et al. [15] presented a system that helps automate the task of authoring and viewing interactive cutaway illustrations of complex models. Their system contains an authoring stage which they call “rigging”, which is basically the process of splitting up the system into separate meshes, defining possible cuts and viewpoints. They first organize each of the objects into 4 separate categories depending on their geometry: rectangular parts, long narrow tubes, radially symmetric tubes and shells (**Figure 14**). If the object is rectangular, it would be parameterized as a rectangular box, and if it is a narrow tube it would be parameterized as one, etc. The system tries to automate this process but users are allowed to manually override the shape chosen if the system incorrectly identifies the shape.

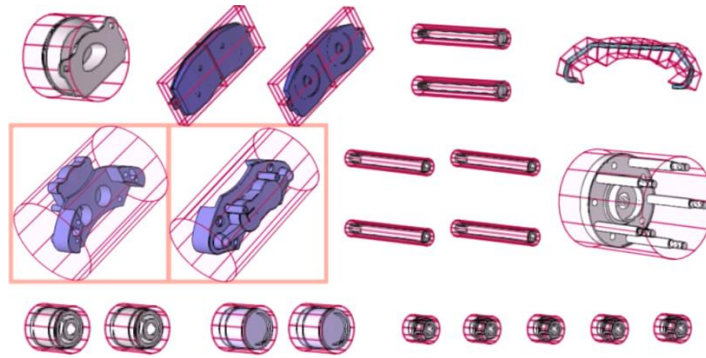


Figure 14: Interactive cutaways defined by parameterization of parts. [15]

The system would then create certain types of cuts based on the parameterization of the parts. These include object-aligned box cuts for boxes, tube and wedge cuts for tubular structures and window cuts for thin-layered shells such as skin (**Figure 15**).

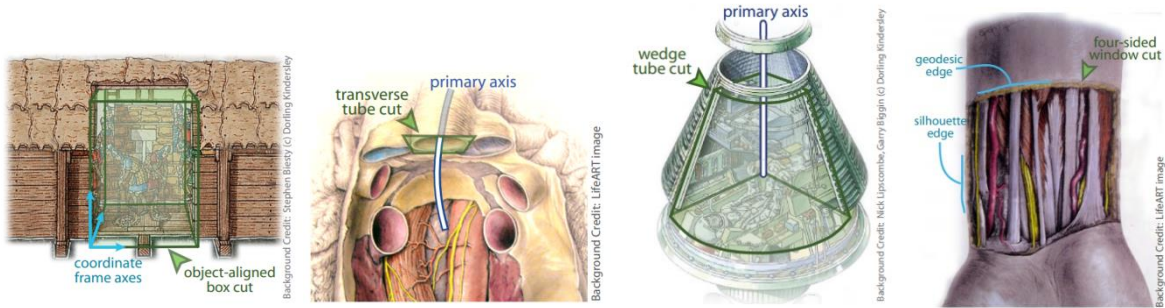


Figure 15: Box, tube, wedge and window cuts. [15]

Wedge cuts are often useful because they do not entirely remove the tube from view, but only cut a portion of the tube, so users can still mentally *complete* the object from the partial information given. This is a form of *Smart Visibility*, (a term coined by Dr. Viola [33]) since it makes use of the expertise of a human observer to preserve context, but still keeps important features visible.

In Li et al.'s paper [15], generated cut-aways are also insetted based on the user's viewpoint; objects of interest are cut less than surrounding or occluding objects. The system makes use of stylized rendering techniques such as edge-shading to create shadows emphasizing edges with depth discontinuities. This way, users would have a cue of how far away objects are from each other and would have a better understanding of the spatial relationships between them. While diffuse shading can provide information on the contours of the object, it can also make areas that are facing away from the light source relatively dark. To solve this, visualizations might only darken some edges of the cut surfaces rather than use diffuse shading.

After the authoring phase, (which can take anywhere from 2-20 minutes according to Li even with the automated system, depending on the number of parts in the system) the user can finally browse the visualization. **Figure 16** below demonstrates a generated view of the disk brake model (which contains 27 parts). Many educators in the user study suggested that the system could be a practical tool for creating teaching aids and user guides for students or act as a support tool for learning.

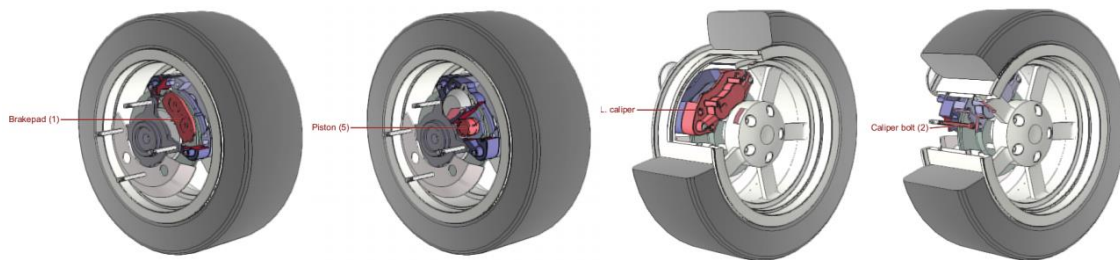


Figure 16: Automatically generated cutaway views of disk brake model. [15]

2.1.4 Gimlenses

Gimlenses is a visualization technique developed by Pindat et al. [13] that combines a cut-away technique with multiple detail-in-context views of the data. Basically, users can navigate a 3D model by drilling holes into outer layers revealing objects deep in the scene (**Figure 17**). Because it generates multiple magnified views of the data, the user is able to compare different areas of the scene at the same time with a detailed perspective, while keeping context of the entire scene. The only occlusion problem might be from the extra space required for the overlays of each view. Every view is connected to its magnified point of focus by two white lines. According to Dr. Bowman [34], using indirect manipulation techniques such as controlling the cursor through an extension of the arm (such as using the tip of a cone pointing to an area of

interest with its base as a control handle) can lead to a more precise way of selecting objects (as compared to more direct approaches where finger-occlusion can be a problem) [34].

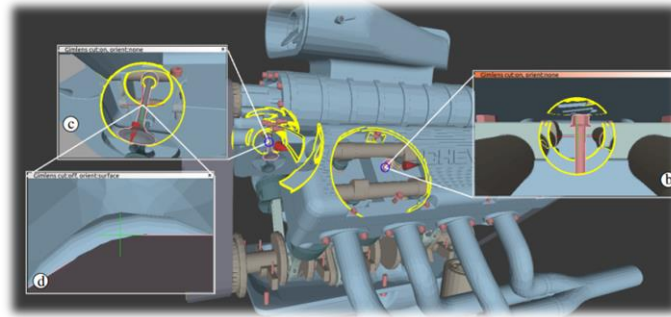


Figure 17: Gimlenses; drilling into a car engine. [13]

These lenses can even be cascaded together, in that a user can create inner lenses within every Gimlens magnifying the scene even further. The visualization technique takes advantage of the idea of using multiple-views into the data to gain context of the scene. This technique of showing multiple views of a dataset at the same time, (at different angles) is commonly used in modelling packages to help users gain a spatial understanding of the scene, making it easier to place objects in 3D space. Modelling packages, (such as Blender and Autodesk Maya) usually display a top, side and front view of the scene (often orthographic views) (**Figure 18**). This is useful because a single view occludes the other half of the model, making it difficult for users to see what direction they are extruding/rotating/scaling if parts of the model are hidden.

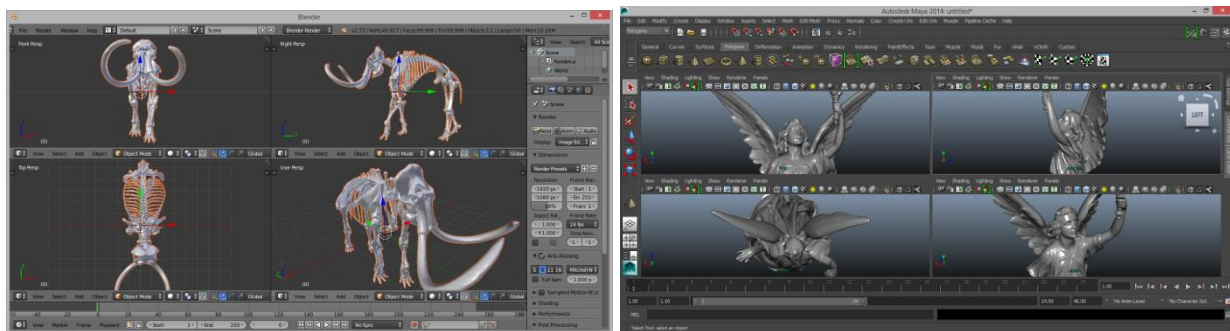


Figure 18: Front, side and top orthographic and perspective views. (Blender and Maya).

In Gimlenses, Pindat et al. define their interface in terms of 3 items: an object selector, lens proxy and cone-shaped cut. The object selector is a blue ring that the user can move around that identifies the object of interest in the model. Both the object selector and view window are connected by two white lines as mentioned above. But since the scene is 3-dimensional and not in 2D, we also need to know the lens' (or camera's) orientation; this is achieved by rotating a lens proxy, which is represented as a green cone in the scene that can be rotated around the object selector (blue ring) (**Figure 19**). This would give the user a better sense of the view's spatial orientation.

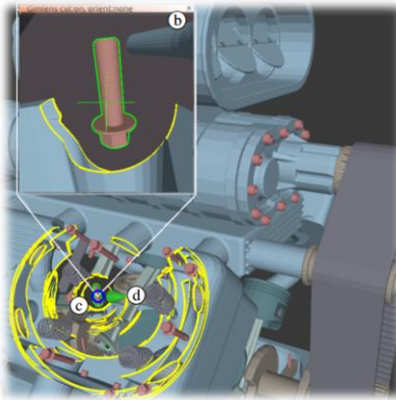


Figure 19: GimLenses; object selector, lens proxy and cone-shaped cut. [13]

Finally, the cone-shaped cut is simply a cut-away but in the shape of a cone. The tip of the cone would be at the object selector's position (the area of interest) and the base of the cone would be facing towards the viewer; this way, because the tip of the cone is more narrow, objects near the area of focus would be cut very little whereas outer objects closer to the viewer would be cut a larger amount. The author uses the term “drilling” because the cone-cut is essentially in the shape of a drill with the tip of the cone facing the area of focus. One of the possible applications of Gimlenses is for solving the problem of when several designers work collaboratively on a dataset, requiring a merge of their work into a single assembly; problems can occur due to

different scales and positions of objects. This tool could be used to identify those problem areas, especially in larger datasets where it may be difficult to get a clear image of where objects spatially lie with respect to one another.

2.1.5 2.5D Clip Surfaces

A common technique for exploring volumetric models is using clip-planes, to cut a 3D model using a plane and show its cross-section at that point; the user could interactively move the plane up and down (or along a defined axis) along the length of the volume to see the cross-sections at each point of the volume (similar to slices in a CT scan). Trapp [16] builds on this idea but introduces a clip-surface; it is similar to a clip-plane but would be more flexible as it also contains additional offset information (represented as a texture map holding height values at each point on the plane); this way, the user can now not only do planar-cuts but also curving cuts that can create unique contours on the cut surface (**Figure 20**). It can also be used as a way to represent bumpy inner textures of a volume since many solid objects when broken into two halves are not cleanly divided by a plane.

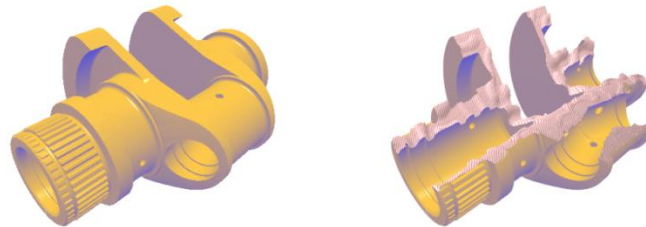


Figure 20: 2.5D clip-surface example demonstrating curving cuts. [16]

2.1.6 Adaptive Cut-Aways

Cut-away illustrations of 3D scenes are usually not effective when rendered as a static image. To gain a clear understanding of where objects lie spatially, we often need to be able to pan and orbit around the scene. In Burns' paper on Adaptive Cutaways [17], he introduces a view-dependent approach to cut-aways where cuts are adapted to the user's viewpoint. This can be useful for visualizing whole room interiors from an outside view, or for focusing in on characters moving through a building in a video game. The algorithm Burns' proposes is image-based and uses a depth-image to represent the cut-away surface (similarly to 2.5D surfaces) but instead of a height map on the local clip-plane, he uses a full-screen quad, or a depth-texture with each pixel representing a depth-threshold (or cut-off value) of how deep into the image a fragment must lie to be discarded and cut-away; this way cut-aways would dynamically change based on the user's view, specifically how far away they are from the object of interest (**Figure 21**).

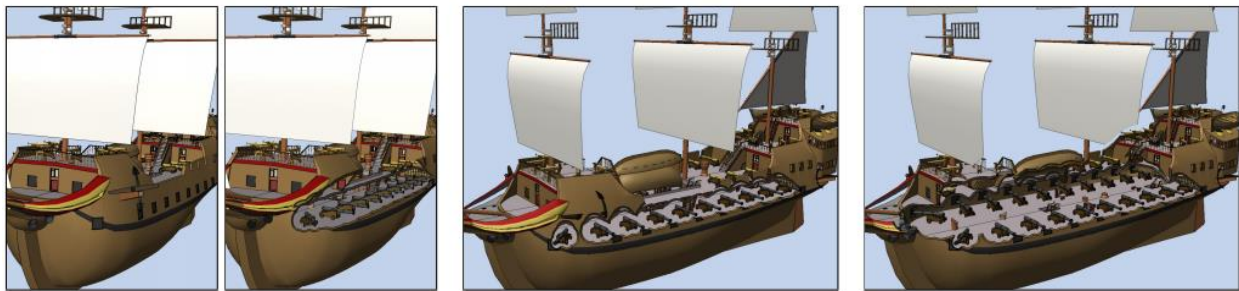


Figure 21: Adaptive cut-aways; galleon's hull is cut away to reveal interior cannons. [17]

He also combines this technique with non-photo realistic rendering techniques; he uses a ghosted view of removed objects to keep context of where they were initially placed (**Figure 22**). Although rather than rendering the removed object in its entirety, he only renders an outline of the object, similar to a toon rendering of the object.

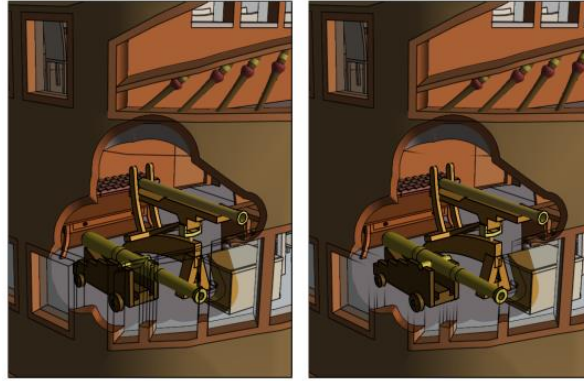


Figure 22: Ghosted view and faded ghosted view of stairs. [17]

In addition, the author takes the ghosted view rendering one step further and renders a partially faded ghosted view of the removed object to reduce the amount of visual clutter in the scene.

2.1.7 Automated Cutaways for Flow Illustration

As we have seen, geometric cut-aways can be positioned manually or by using a view-dependent approach. Sigg [21] states that the problem of finding an optimal placement for cutaway boxes can be an NP-hard problem and therefore finding an algorithm to solve it is not a feasible solution. He proposes a visualization technique that alternatively tries to automate reasonable placement of the cutaway template through the use of a Monte Carlo method instead (MC) [21]; that is, it finds a reasonably good solution through a directed, randomized search. This technique was designed for the use case where manually finding the optimal cut-away placement would take too long, (due to complexity of the dataset). The method requires a stage where users select interesting parts of the data based on background knowledge or insight into the data and an automated step where the algorithm finds an optimal placement based on the user's selection. Basically, a degree-of-interest (DOI) function is inputted into the algorithm, where the user assigns an importance value to certain vertices of the mesh and specifies other parameters such

as the maximum number of geometric primitives to be used for cutting out the object of interest. Inputting the data for this approach can be quite time consuming, making the visualization non-interactive, so it is likely only useful for very large and complex datasets where manual segmentation would take much longer; the author demonstrates this technique on turbulent flow data, rendering the user’s inputted importance value as a colour (from white to orange – lowest to highest) on the isosurfaces (**Figure 23**).

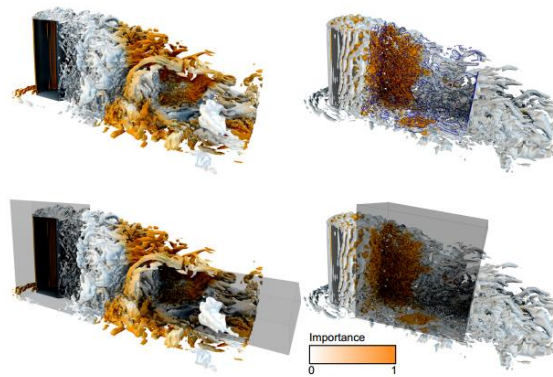


Figure 23: Automated cutaways in flow visualizations. © 2012 IEEE [21]

2.2 Transparency and Ghosted Views

Transparency is often used as a technique for occlusion management. Unfortunately, properly rendering scenes with many layers of transparency, with accurate results can be a demanding task. This is because surfaces must be rendered in correct depth order for it to appear correctly in the scene, (otherwise artifacts in self-intersecting geometry of translucent surfaces may appear). If the object is a simple convex shell with very little intersecting geometry or if they’re not intertwining other layers, sometimes the problem can be solved with a simple hack by rendering back faces of the surface first, followed by front faces. Sorting faces in the CPU is a very computationally expensive task especially since it must be performed every time the view

changes; there are GPU-based techniques that sort the faces in hardware; one of these techniques is Depth Peeling, with an improved version called Dual Depth Peeling invented in 2008 by NVIDIA [35]. Although it produces accurate results, it can be computationally demanding because of the multiple passes that must be executed in the pipeline. These techniques are classified in the category of Order Independent Transparency (OIT) techniques [35]. Some recent techniques in OIT (that run on modern graphics hardware) include using per-pixel linked lists and storing the depths of each fragment in these linked lists, then sorting them; this reduces the number of passes required for an OIT algorithm, but it is still in development [36]. Another technique, developed by McGuire [37] at NVIDIA, called *Weighted Blended Order-Independent Transparency* can be performed in a single pass but is just an image-based approximation rather than correctly sorting faces (**Figure 24**); basically, the algorithm uses additive blending and each fragment is weighted based on depth. Farther away translucent fragments would have a lower weight which means it would contribute less to the final colour, whereas fragments objects would have a higher weighting and contribute more to the final colour. The user can choose a weighting function appropriate for their data but certain weighting functions are more effective for particular datasets and might not perform as well in others [38].

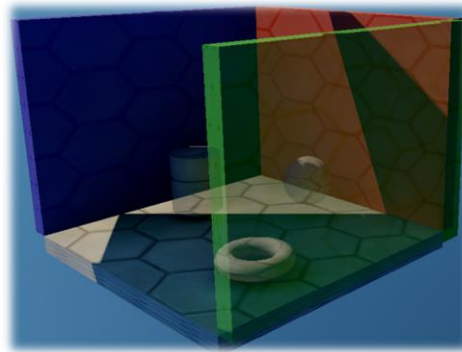


Figure 24: Weighted Blended Order-Independent Transparency example. [38]

2.2.1 *Interactive Ghosted Views*

Ghosted views are commonly used in technical illustrations to peer into the interiors of an object (**Figure 25**). Often, the opacity of less significant objects is reduced while the opacity of more detailed items is increased. The goal of this technique is to give users enough cues so that they can mentally complete a partly removed structure even if they're almost entirely removed from the scene [39].



Figure 25: Ghosted view of a car's interior. [39]

Cmolik [19] implemented a real-time ghosted view visualization technique where users are able to interact with the scene, (pan, zoom and rotate) while the object of interest remains visible. His technique first organizes objects into several groups that share a semantic meaning (the engine, seats, etc.) then each of these groups are ordered into layers. The opacity of certain objects in the layer is modulated based on their importance value; less important objects will appear less opaque and important objects will have a higher opacity. The layers are then composited from bottom to top resulting in a final ghosted view illustration.

2.3 Explosions

While ghosting techniques remove detail from occluding parts by fading them out, explosions preserve the occluding parts' details by just spatially shifting them out of the way. They are

effective at managing 3D occlusion when details of occluding objects must be preserved. Although they preserve visual details of the object, spatial relationships are distorted since objects are shifted from their initial positions. Traditional exploding diagrams usually move objects outwards in a radial fashion away from a focus point.

2.3.1 *Expanding Annotations with Explosion Probe*

A simple example of explosions is presented in Carpendale's paper on *Integrating Expanding Annotations with a 3D Explosion Probe* [40]. The paper tries to solve the problem of revealing hidden, interior components (that are not initially visible) but also have ancillary text connected to them. An example of this type of data could be anatomical models containing structures with labels storing their names and descriptions. The author solves this problem by developing a system that creates interactive explosion diagrams to dynamically reveal hidden components. When the mouse moves within an annotation trigger area, the hidden parts are revealed and translucent scrollable text is overlaid onto the area (**Figure 26**). A 3D probe is used to interactively explode objects apart and return them to their initial positions.

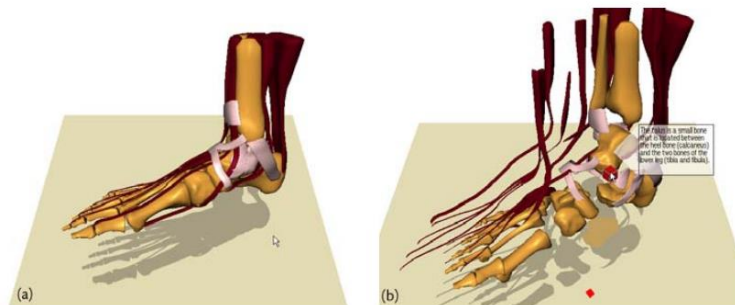


Figure 26: Annotated explosion diagrams. [40]

Explosions are implemented as simple radial explosions around the probe; all objects within the vicinity of the probe are transformed outwards. This means users can easily explode at different

points on the scene simply by moving the probe around using the cursor. The textbox only appears when the user selects the object associated with it. The rest of the text is expanded when the cursor moves closer to the centroid of the object; similarly, when the cursor moves away from the object's centroid the textbox contracts again. Rather than using the center of the object's bounding box as the centroid, the author chose to compute a skeleton for the object and a significant point on the skeleton is computed, which becomes the centroid. This is because the centre of the bounding box might not even be a point on the object. On the other hand, the skeleton method would determine a point that does lie on the mesh (**Figure 27**). The skeleton can be computed by first triangulating the mesh, then collapsing each edge into a vertex until no more triangles exist. The significant point would then be the point with the most skeleton segments or edges connected to it. This paper demonstrates a simple, but naïve approach to creating exploding diagrams based on a probe's position since it relies on simple radial explosions; while it is useful for revealing occluded structures within a mesh, the explosions are not constrained so parts are spread apart in all directions in a non-organized manner forming clutter in 3D space.

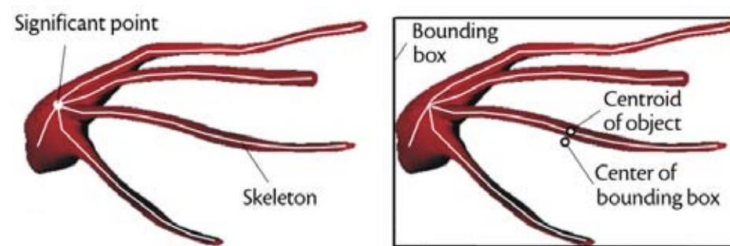


Figure 27: Finding the centroid using a skeleton vs bounding box method. [40]

2.3.2 Compact Explosion Diagrams

Building upon traditional exploding diagrams, Tatzgern [18] introduced another approach called *Compact Explosion Diagrams* where only subsets of assemblies are exploded. The system tries to automate the generation of these diagrams by trying to figure out a comprehensive arrangement of parts that would make sense to the user while reducing visual clutter. His approach tries to reduce the visual complexity of traditional explosion diagrams by reducing the number of exploded assemblies to only a subset of the scene; this means grouping identical pieces together and exploding a single *representative* piece for that group (**Figure 28**).

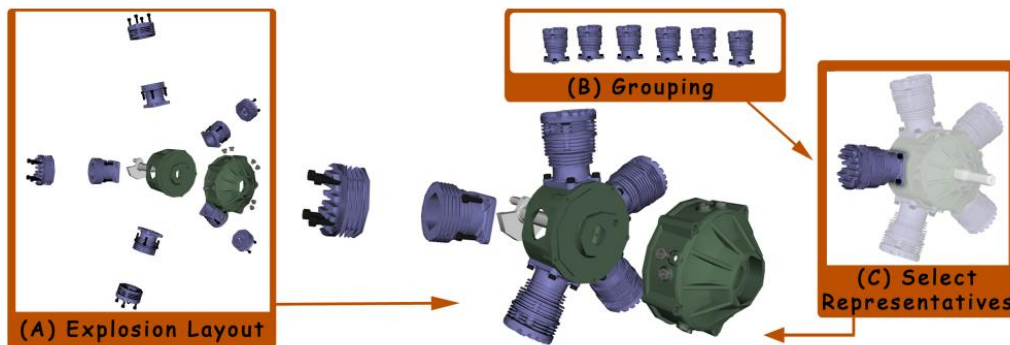


Figure 28: Compact explosion diagram; grouping identical pieces. [18]

This technique was designed particularly for models containing many identical pieces (such as the assembly of a chair that contains multiple identical screws). Usually, identical pieces are exploded in a similar fashion; traditional diagrams would repeatedly show the same explosion for each one of these pieces, repeating the same explosion multiple times, introducing redundancy in the scene. Tatzgern’s approach tries to decrease the amount of repetition by reducing it to a subset. The algorithm first searches for identical parts and places them into a group. It then generates an initial explosion diagram depending on a direction and distance chosen. The system then looks for a *Representative*, a term the author uses to mean a single

instance of an item in a group to be exploded. The exploding diagram is then only generated for that representative rather than for every repetition of the piece.

2.3.3 *HowThingsWork: Autogenerating Assembly Visualizations*

Whereas the previous visualization technique tries to generate a simple and compact explosion diagram for assembly models with many identical pieces, *HowThingsWork*, (a visualization tool developed by Mitra et al. [22]) generates an animated explosion diagram that depicts the motion of parts in a mechanical assembly (**Figure 29**). The motion of each part is inferred by the part's geometry along with constraints inputted by the user. For example, parts are organized into typical joint and gear categories, such as axle, crank, belt, lever, speed gear, worm gear, bevel gear, etc. [22] then using forward-kinematics, the system would compute the relative speed and motion for each part based on its joint type, resulting in a final motion-animated visualization.

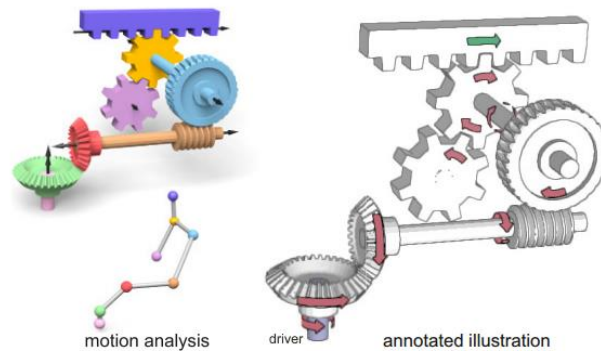


Figure 29: HowThingsWork; animated assembly diagram. [22]

2.3.4 *View-Dependent Explosions*

Bruckner [14] builds upon traditional explosion techniques by introducing a force-based, view-dependent implementation. His approach is tailored for volumetric data rather than surface data. The volume is first decomposed into separate parts that can be displaced. A *Selection Object* is

chosen (using a selection volume) to represent the object of interest. After the object is set, it would exert an explosion force on all surrounding parts pushing them away (**Figure 30**). The parts will then rearrange around the selection object according to given force constraints.

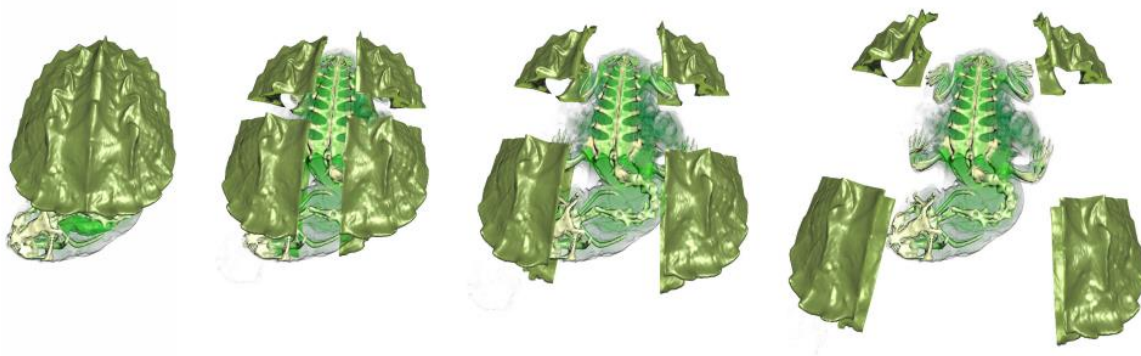


Figure 30: Exploded view of a turtle. © 2006 IEEE [14]

Return forces are also present in the system, which are attractive forces moving the exploded parts towards their original position. The explosion and return forces are defined using simple logarithmic springs. To solve the occlusion problems that may appear when users change their viewpoint, Bruckner introduces a viewing force. The viewing force tries to rearrange parts so that they do not occlude the selection object based on the viewing transformation. This is done to ensure exploded parts do not obstruct the object of interest no matter where the user is looking. A spacing force is also applied to ensure parts do not form clusters. Bruckner proposes using widgets such as joints to constrain the movement of parts in the explosion. Bruckner's paper provides a good framework for solving the occlusion problem; however, as stated before, spatial distortion is an inherent problem in explosion techniques and can be more apparent with the addition of viewing and spacing forces as well as other applied forces.

2.4 Magic Lenses

Magic Lenses are virtual interactive lenses that provide an alternative visual representation of the data (sometimes through distortion, such as a fisheye lens) [23]. They have been introduced in the early nineties specifically for 2D data, but lenses can also be used for 3D data (**Figure 31**); spherical lenses can be used to magnify volume data or gain insight into flow visualizations.

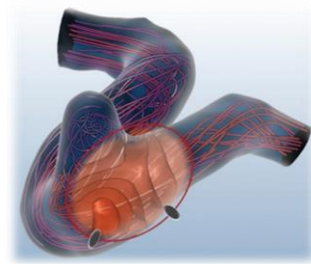


Figure 31: FlowLens; 3D lens displaying blood flow. © 2011 IEEE [41]

Magic lenses are often used to transform data in local areas of a dataset. Unlike regular lenses which simply magnify data, magic lenses can be used to distort it (**Figure 32**); for example, the simple magnification lens shown below (middle) would simply display data as is but zoomed in:

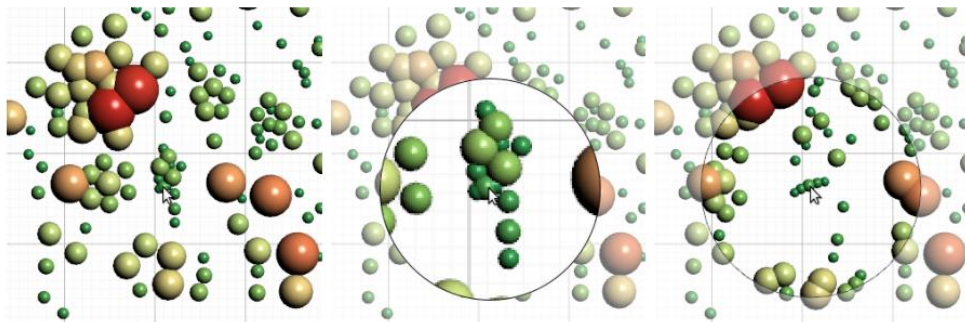


Figure 32: Magic Lenses; original data, magnification and fisheye lens. [23]

The spheres in the middle image (the magnification) are cluttered together, occluding one another. On the other hand, a Magic Lens could distort reality; in the case of the fisheye transformation on the right, the distortion untangles the clutter revealing the occluded structures.

Magic lenses are not only capable of distorting spatial information; they can change any graphical properties of the dataset. GimLenses [13] (as we've discussed before) is actually a visualization technique that combines cutaways with Magic Lenses.

2.5 Deformations

One of the more recent techniques for data exploration include using deformation techniques implemented in the GPU for visualizations. With the development of graphics hardware, deformations can now be efficiently implemented in vertex, fragment, geometry and tessellation shaders (or even in compute shaders), allowing flexible and creative control over illustrations.

2.5.1 Illustrative Deformation

Dr. Correa from Rutgers University uses the term, *Illustrative Deformation* [8] to describe an active approach to visualization where the user can manipulate a dataset (deform it) as if it were a real mouldable object (**Figure 33**).



Figure 33: Illustrative Deformation; peeling and twisting. [8]

The idea came from the fact that many traditional visualization techniques assume the user's role is passive, in that the system would pre-determine how the visualization should appear, giving them little control over the visualization. Correa wanted to develop a visualization technique

where the user's role was active, giving them more control over the data. Although automatic placement of deformation templates is useful for quickly peering into meshes, sometimes the user requires more flexibility.

In his paper, Correa talks about the challenges with visualizing volume data, stating that it was difficult due to many factors, including occlusion, clutter, noise in the data and acquisition pose [8]. For example, in this CT scan of a mouse fetus (**Figure 34**), it is difficult to see the head due to its acquired pose from the device. This can make visualization a challenging task, especially if the object of interest is originally occluded in the acquired dataset.



Figure 34: CT scan of a mouse fetus; the head is occluded. [8]

Correa describes illustrative deformation as useful for performing virtual operations on an object which do not conform to reality but are useful for understanding the structure of complex objects. He uses the idea of 3D displacement maps to perform the deformations stating that it can be applied to both volumetric and surface-based models. Surface-based models may require remeshing when large deformation operations are performed (cutting) since they contain connectivity information which may be modified when the operation is performed; in contrast, volumetric data is represented by voxels and connectivity information is not explicit therefore remeshing is not required. Correa's deformations are applied using deformation templates (**Figure 35**) which are placed onto the object being deformed; some examples of deformation templates described in the paper are: twist, poke, split, retract and slice.

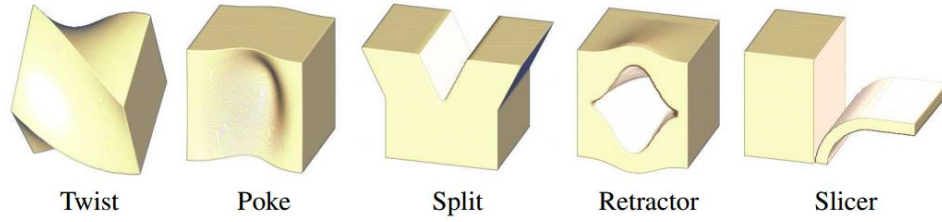


Figure 35: Deformation template examples. [8]

These deformation templates are represented using a displacement map which basically transforms the position of a point in 3D space. Correa splits the deformation templates into two categories; continuous and discontinuous deformations; discontinuous deformations include actions such as cuts, peels and slices where connectivity information of the mesh is lost whereas continuous deformations include less drastic actions such as twists, bends, pulls and pokes. The example below, (**Figure 36**) illustrates the result of applying a peeling deformation template followed by a wave template onto a tomato dataset:



Figure 36: Peel deformation followed by a wave. [8]

The displacement map for the wave deformation is described in (2.1) as a function of (x, y, z) where the displacement vectors are 0 for the x and z components (no change in displacement in the x - z plane) and the y -value is modified by a Sine function of x and z , with a as the amplitude of the wave and ω as its frequency:

$$D(x, y, z) = \begin{pmatrix} 0 \\ \text{asin}(\omega \sqrt{(x - 0.5)^2 + (z - 0.5)^2}) \\ 0 \end{pmatrix} \quad (2.1)$$

The paper also describes feature-sensitive operations which are operations performed specifically on a semantic component such as skin without affecting other components of the object. These are effective when the data can be separated semantically into different groups. Features of interest can also be “masked-out” so they are not affected by the deformation.

2.5.2 View-Dependent Peel-Aways

In Birkeland's paper, he builds on the peeling deformation technique by introducing view-dependent peel-aways [12]. In Correa's approach, the user is required to place a deformation template onto the volume but Birkeland proposes an algorithm to automatically place the template relying upon the user's viewing position and direction (**Figure 37**). This means that users no longer need to manually place this template and eliminates the extra step. The user is now only required to select a view; the template will be placed accordingly such that layers are peeled away from the user's viewpoint revealing features of interest.

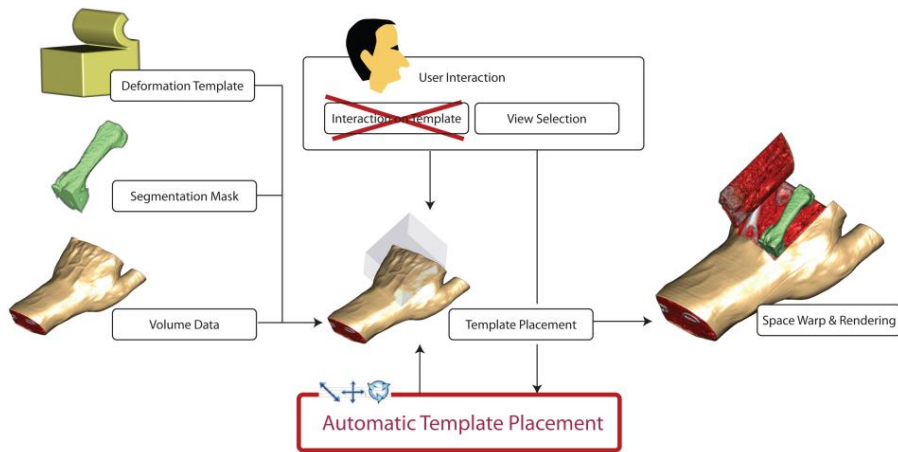


Figure 37: Automatic placement of the deformation template. [12]

Although the user has one less step to perform, he or she still needs to place something that Birkeland calls a *Segmentation Mask* [12] into the scene to select the features of interest which will be revealed when the peel is performed. In the system, when the view changes, the peel is animated from a *Closed Peel* state to an *Open Peel* state so that the original positions are restored each time providing the user a contextual understanding of the data at rest. After the features of interest are defined, the template is positioned and scaled according to the features' size, such that the peel roughly encapsulates them. The paper provides 3 types of peels that can be applied to the dataset: Rigid Peels, Cabinet Peels and Deformed Peels (**Figure 38**).

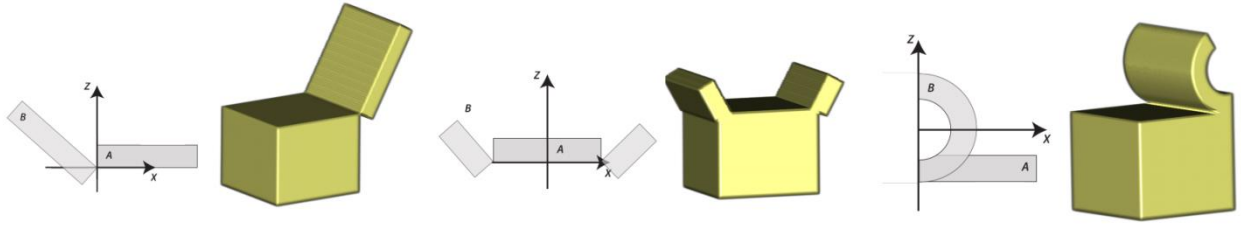


Figure 38: Rigid peel, cabinet peel and deformed peel examples. [12]

Rigid Peels are basically defined as a rotation of points around its local Y axis, cabinet peels are basically two rigid peels but with the peel on the right being a mirrored transformation of the peel on the left. The deformed peel, on the other hand bends the shape of the object and is a forward transformation based off of McGuffin's peeling template [7], defined in (2.2).

$$\begin{aligned} x' &= -z \cdot \sin\left(\frac{x}{R}\right) \\ z' &= z \cdot \cos\left(\frac{x}{R}\right) \\ y' &= y \end{aligned} \tag{2.2}$$

After the peel, original (x, y, z) coordinates are transformed into (x', y', z') . Points along the local y-axis are preserved whereas x and z points are transformed based on a Sine or Cosine function of x

and R , where R is the radius of the bend. Birkeland also defines an inverse of the forward transformation to undo the effects of the peel in (2.3).

$$\begin{aligned} z &= \sqrt{x'^2 + z'^2} \\ x &= R \cdot \cos^{-1}\left(\frac{x}{R}\right) \\ y' &= y \end{aligned} \tag{2.3}$$

This paper is similar to Bruckner's paper on Exploded views [14] for volume data in that both papers solve the problem of occlusion by automating placement of objects so that they do not obstruct the user's view. In the results, Birkeland states that rigid peels have the benefit of preserving spatial relationships for structures within the peel (since deformable peels have a bendable radius, structures within the peel and their spatial relationships might be slightly distorted). Cabinet peels are basically an extension of rigid peels but with two halves of the object pulled apart, so they share the same benefits and properties of rigid peels.

2.6 Elastic Band Metaphor

A recent technique for exploring spatial relationships in complex models was proposed by Otsuki et al. [20] using the idea of an elastic rubber band to observe relationships between parts of a model. Their system, which they call *Stretch 'n' Cut*, is meant to be used for learning purposes, specifically for learning complex models such as human anatomy or industrial products. Connections between parts are represented by virtual elastic bands (or lines); if a user lifts up an object (connected to other parts) to observe underneath, there will be lines drawn to show connections between parts. The user can cut the connecting bands if they want to remove a part from the group. All connected parts belong to a group; if the user lifts up an object then decides to let go of it, the object would snap back to its initial position. Otherwise, if the user

cuts the connection, the object will no longer belong to the group and will no longer snap back to its original position. The author calls this process *ungrouping* an object (**Figure 39**).

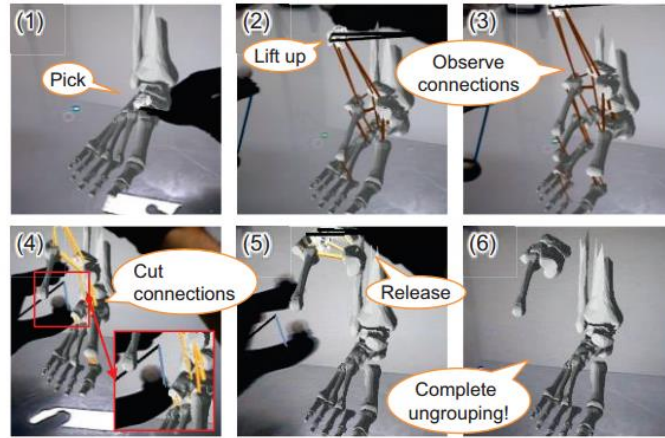


Figure 39: Lifting and ungrouping an object. © 2014 IEEE [20]

The strength of the connections between parts is represented by a line width; the stronger a connection, the thicker the width of the elastic band, and the weaker the connection, the thinner the elastic band. Their system uses a force-based algorithm to determine movement of parts. Parts are transformed using an elastic force, F_{elastic} and a repulsive force, $F_{\text{repulsive}}$. The elastic force uses a spring equation to compute the amount of force applied to each connected part; if a part is cut or separated from the group then its elastic force would be 0. The repulsive force is used to extend spacing between parts (essentially spreading them apart) so that spatial relationships are shown more clearly to the user (**Figure 40**).

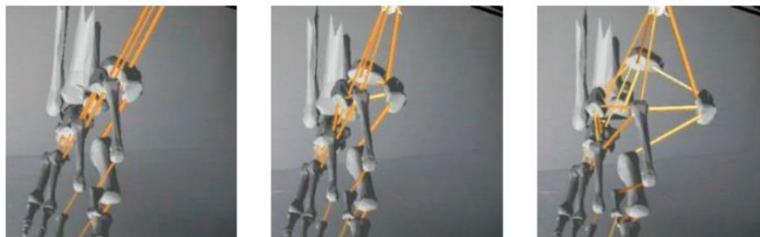


Figure 40: Gradually increasing repulsive force. © 2014 IEEE [20]

The system is implemented as a Mixed Reality (MR) application where the user wears a head-mounted display and uses a ViconPeaks [20] motion capture system to interact with the scene using their fingers. Hand gestures, such as crossing their hands will perform a cut, and pinching will pick and lift up an object. Contact relationships of parts are shown as a graph in **Figure 41**, with certain parts organized into individual groups.

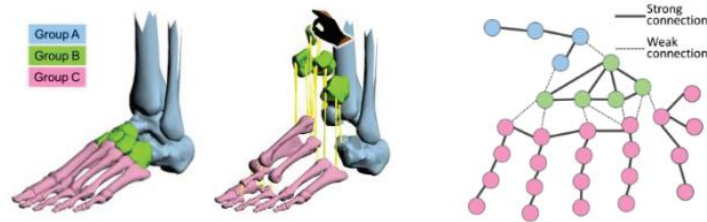


Figure 41: Contact relationships between parts shown as a graph. © 2014 IEEE [20]

When the user picks up a part, connected parts are gradually lifted up as well so that their spatial relationships can be observed. To prevent an entire object from moving when a part is lifted, the system does not move parts that are anchored (parts that contain only one adjacent edge on the graph).

3 Methodology and Implementation

In the previous chapter, we described visualization techniques that have been traditionally used along with more recent techniques for managing 3D scene occlusion. This chapter describes the mathematics and implementation details for the development of *Aperio*, a visualization system for revealing occluded structures in a complex 3D scene. The system implements a novel approach combining a mechanical analogy with superquadric-shaped tools to perform real-time cuts (cookie cutter and ribbon cuts), as well as hinge pieces apart and flexibly explode objects along a constrained path (similarly to beads on a string or wire). Interactions are analogous to real-world actions so users can quickly and intuitively figure out how to use the tools to perform each action.

Essentially, the system should lend itself to visual affordance, meaning users can figure out how to use the system based on suggested shapes and visible properties of the tools. The term, *Visual Affordance*, was originally coined by James J. Gibson, an American psychologist working in the area of visual perception and was redefined in the context of Human-Computer Interaction (HCI) by Donald Norman in 1988 [42]. In his book, *The Design of Everyday Things*, he defines affordance as follows,

[T]he term affordance refers to the perceived and actual properties of the thing, primarily those fundamental properties that determine just how the thing could possibly be used . . . A chair affords ("is for") support and, therefore, affords sitting.

Our system, *Aperio*, allows users to perform cuts, hinges and explosions using a single underlying geometric representation, a superquadric. We've chosen to use superquadrics because they have good shape coverage, meaning mathematically they can represent many different shapes (supertoroids, spheres, rounded and non-rounded cubes, cylinders, capsules, etc.) but are

clearly defined since they contain few controlling parameters. This means users can easily make flexible cuts or interesting explosion paths by simply moving and reshaping a superquadric. We also introduce a real-time cutter previewer (**Figure 42**) to interactively peer into the dataset by simply moving the cutter on the surface of the mesh similarly to moving around a lens.

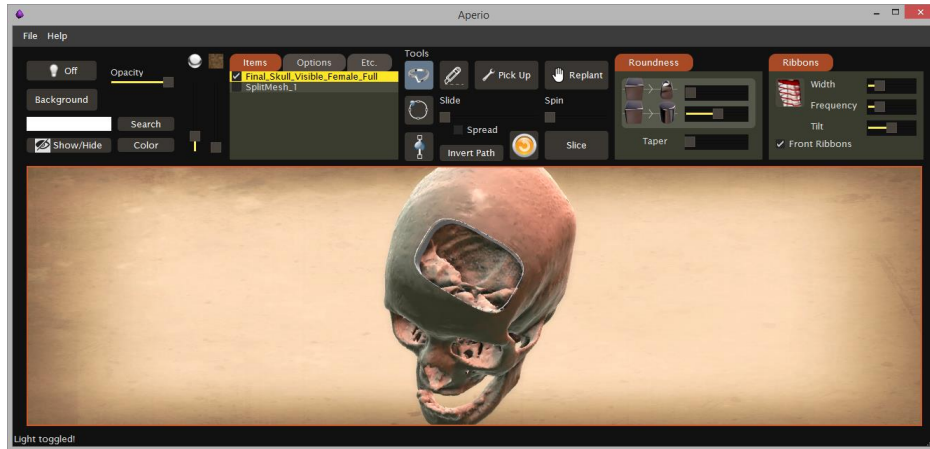


Figure 42: Aperio’s cutter previewer; peering into a skull.

Aperio is meant to be used for revealing occluded (or hidden) structures and to visualize spatial relationships between parts of a complex model; it is primarily created for anyone who wants to break apart a complicated system for the purpose of understanding. They can interactively manipulate the model for inspection/visualization purposes in real-time to explore and understand the model. It is not, however, a modeling package like Blender or Maya and does not focus on tools for the creation of new models.

3.1 Implementation

Aperio is written in C++ and uses OpenGL/GLSL for scene rendering. The OpenGL Extension Wrangler Library (GLEW²) is used to load modern OpenGL extensions and the Open Asset

² GLEW, <http://glew.sourceforge.net>

Import Library (Assimp³) is used for loading and preparing 3D models. We use Qt⁴ for the GUI, which is a popular cross-platform framework developed by Digia (**Figure 43**).

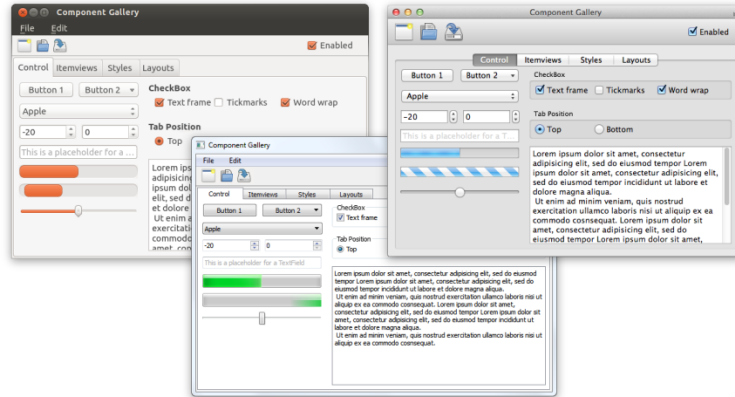


Figure 43: Example QT widget applications. [43]

Qt is used in many commercial software, including Autodesk Maya, Adobe Photoshop Elements, Skype, VLC Media Player, Mathematica, Bitcoin Core, VirtualBox and even in the desktop version of Google Earth and the Linux Desktop Environment, KDE. It supports multi-touch devices and starting with Qt 5.3, applications can be deployed to Android, iOS and Windows Phone devices. Qt also contains non-UI related components such as an API for accessing SQL databases, parsing XML/JSON, network support, file handling, threads and concurrency support, and image and sound support.

We decided to use The Visualization Toolkit⁵ (VTK) as well, a visualization framework containing a C++ class library and numerous algorithms for scalar, vector, tensor and volumetric methods; it contains built-in advanced modeling techniques for implicit modeling, polygon reduction, mesh smoothing/cleaning, cutting, contouring, and Delaunay triangulation. The upcoming version, VTK 6.2, overhauls the rendering subsystem, updating it to use features in

³ Assimp, <http://assimp.sourceforge.net/>

⁴ Qt, <http://qt-project.org>

⁵ VTK, <http://www.vtk.org/>

modern OpenGL. This means supporting faster polygonal rendering and a new GPU-based volume mapper (**Figure 44**).

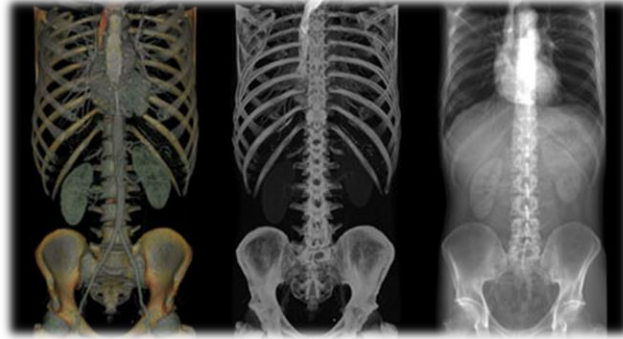


Figure 44: VTK’s modern volume rendering subsystem (Composite, Maximum Intensity Projection, and Additive Blending methods). [44]

VTK is developed and maintained by Kitware, who also maintains CMake⁶. CMake is a popular open-source build system for managing the build process of applications using compiler-independent methods. It is used by numerous applications and libraries such as Blender, Bullet Physics Engine, cURL, libpng, MiKTeX, MySQL, OGRE, OpenCV, OpenSceneGraph, Quantum GIS (QGIS), Qt, Point Cloud Library (PCL), Synergy, Second Life, LLVM/Clang, zLib, and VTK itself [45].

Finally, we decided to use Carve CSG⁷, a constructive solid geometry library written by Tobias Sargeant, for performing cuts on meshes and separating geometry. It is the same library used by Blender for performing Boolean operations and is the default library used in Blender’s Boolean Modifier starting with version 2.62 (**Figure 45**). Autodesk Maya also integrated Carve CSG with their Boolean Operation tools as of Maya 2014.

⁶ CMake, <http://www.cmake.org/>

⁷ Carve CSG, <https://code.google.com/p/carve/>

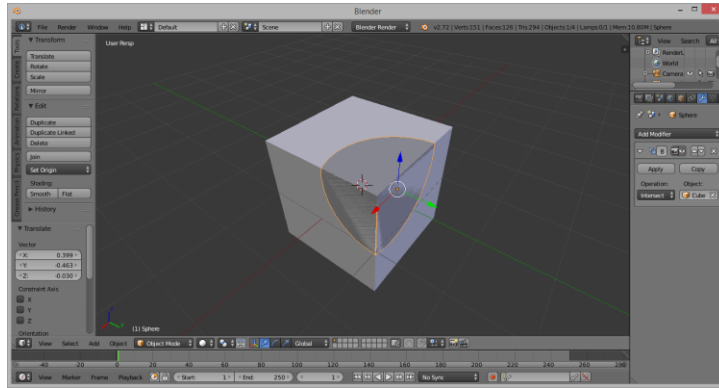


Figure 45: Blender’s Boolean Modifier example; intersecting a cube with a sphere.

3.1.1 VTK

VTK’s visualization pipeline is relatively simple; it begins with a *Source*, providing data input from either a file or auto-generates it from built-in classes in the class library (**Figure 46**)

VTK Visualization Pipeline

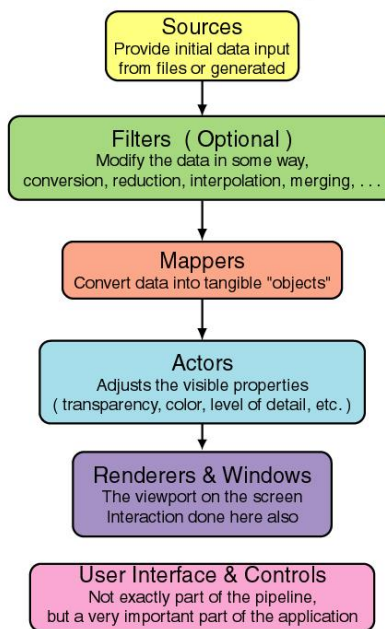


Figure 46: VTK’s pipeline. [46]

The output of the *Sources* can then optionally go through *Filters* that transform the data in some way; for example, translating, rotating, scaling, cleaning up polygonal mesh data, reducing the

number of polygons, extruding parts, triangulating, filling holes, etc. This stage of the pipeline is optional. The outputs of the *Sources* or *Filters* are then fed into mappers that prepare the objects for rendering. *Mappers* transform raw data into rendering primitives fed into a graphics library such as OpenGL and are sent to the screen. *Actors* are then created with *Mappers* assigned to them. *Actors* contain visible properties of the mesh, including shaders, material properties such as ambient, diffuse and specular colours, opacity and other physical properties such as transform position, orientation and scale. These *Actors* are then added to a *Renderer* object which is attached to a window and displayed on the screen.

3.1.2 *Multi-pass Rendering*

In *Aperio*, we make use of a multi-pass rendering pipeline for shading and other effects, but we also use it for our capping algorithm to preview cuts; the capping algorithm is used to fill in holes (in real-time) when the cutter is moved along the surface of the mesh. Cuts can be previewed instantly and interactively in the previewer before the mesh is actually cut and separated.

The multi-pass rendering algorithm fundamentally relies on Frame Buffer Objects (FBOs) [47], which are an OpenGL extension that became part of its core after version 3.0. They allow applications to render scene data and other information into textures, to be used later on in subsequent passes. With FBOs, we can render depth values at each pixel location, surface positions, or surface normals into textures. We can read this information back in from textures in future passes, and combine data to form a final composite image. With this pipeline in place, fancy post-processing effects can be applied on resulting images (**Figure 47**).

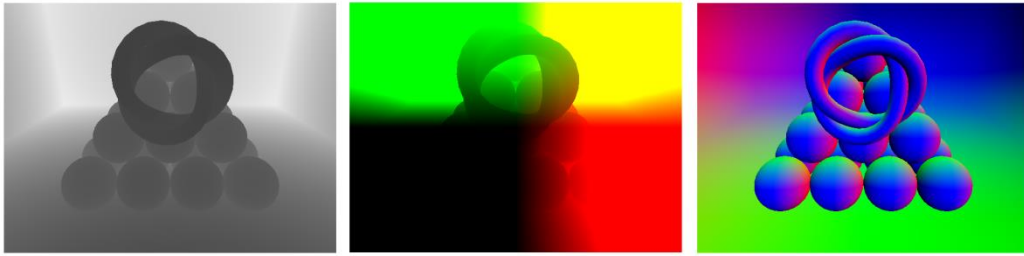


Figure 47: Depth, positions and normals rendered to textures (FBOs). [48]

One example of a post-processing effect that combines the use of depth, positions and surface normals is Screen Space Ambient Occlusion (SSAO) [49]. SSAO is a technique first developed by Vladimir Kajalin while working at Crytek on the Crysis video game in 2007; it aims to approximate the amount of ambient occlusion in a scene using an image-based technique (**Figure 48**). Ambient occlusion is essentially how exposed a point in a scene is to ambient lighting; so the corners of a wall would be less exposed to ambient lighting, hence they become darker as compared to the other portions of the wall that are completely exposed. They are a crude approximation to global illumination but are useful for emphasizing details on the surface such as creases, corners and holes.

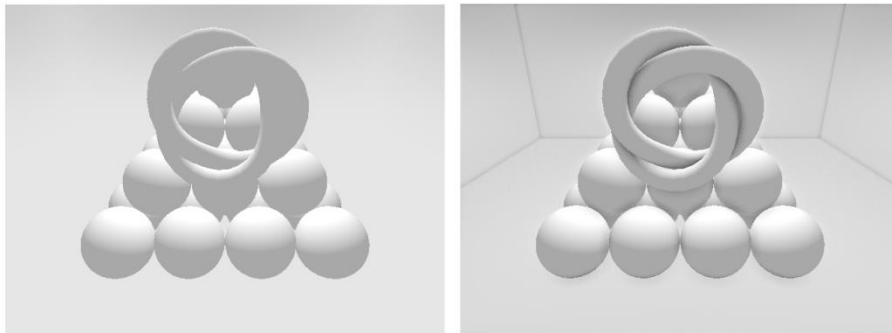


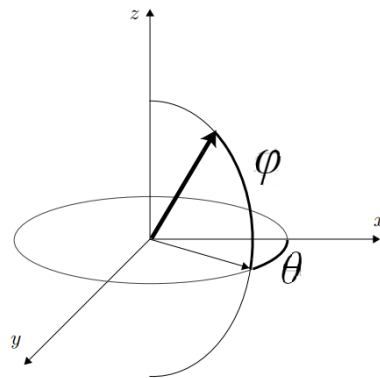
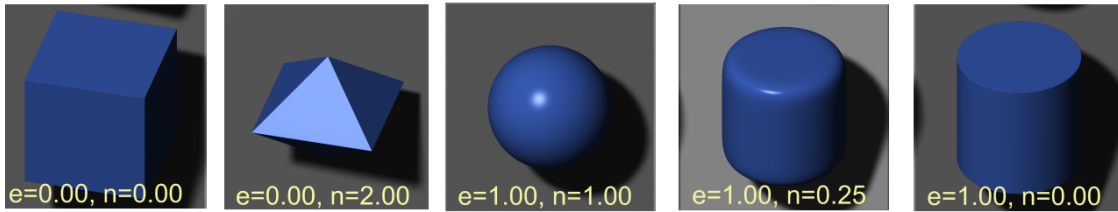
Figure 48: Scene without and with SSAO. [48]

In *Aperio*'s multi-pass rendering pipeline, we make use of Screen Space Ambient Occlusion (SSAO) to emphasize edges and other details in the model. We also combine the technique with

Fast Approximate Anti-Aliasing (FXAA) [50], an anti-aliasing algorithm to smooth jagged lines in the final rendering of the scene.

3.2 Superquadrics

Aperio uses a single consistent interaction model involving superquadrics; that is, all operations (cutting, hinging, sliding and exploding) are performed simply using superquadric-shaped widgets. Superquadrics were chosen because of their large shape coverage; they are able to transform into many different shapes (rounded and non-rounded cubes, boxes, spheres, cylinders, capsules, diamonds, wheels, supertoroids, etc.) but yet have few controlling parameters (phi and theta roundness) making them simple for users to work with. Theta roundness, θ , is longitudinal roundness (horizontally around a sphere) and Phi roundness, φ , is latitudinal roundness (vertically around the sphere) (**Figure 49**).



$e \Rightarrow \theta$ roundness

$n \Rightarrow \varphi$ roundness

Figure 49: Superquadric shapes; altering Theta (θ) and Phi (φ) roundness (e , n parameters). [51]

Just by altering the Theta and Phi roundness (which are controlled by e and n parameters in a superquadric's implicit equation), many different shapes can be created. The implicit equation for the superquadrics used in *Aperio* is defined in (3.1).

$$\left(|z|^{\frac{2}{e}} + |x|^{\frac{2}{e}}\right)^{\frac{e}{n}} + |y|^{\frac{2}{n}} - 1 = 0 \quad (3.1)$$

Where e is Theta roundness, n is Phi roundness, and (x, y, z) are the coordinates we can plug into the equation to test for *inside/outsideness*. By plugging in the coordinates, if the evaluated expression is less than 0, the point lies inside of the superquadric; if it is greater than 0, it lies outside of the superquadric and if it evaluates to 0, it lies directly on the surface of the superquadric.

Superquadrics have both an implicit and parametric equation. Implicit equations are useful for determining whether a point lies inside or outside of the surface; we use this property in the fragment shader to simulate a cut previewer, by discarding mesh fragments if they lay within the cutter's superquadric mesh. The parametric equation, on the other hand, can be used to generate points on the surface of the superquadric given a phi and theta angle (latitude and longitude). Superquadrics are defined in *Aperio* by the following parametric equation in (3.2).

$$x(u, v) = c(v, n) \cdot s(u, e) \quad (3.2)$$

$$y(u, v) = c(v, n) \cdot c(u, e)$$

$$z(u, v) = s(v, n)$$

where u and v are the longitude and latitude *angles*, $u = \text{Theta } (\theta)$ and $v = \text{Phi } (\phi)$ respectively; e and n are the longitude and latitude *roundness* (as defined in the implicit equations) and $c()$ and $s()$ are functions defined in (3.3). The functions, $c()$ and $s()$ correspond to *cos* and *sin* and computes a $sgn()$ function using the two inputted parameters from (3.2), shown here as w and m parameters:

$$c(w, m) = sgn(\cos w) |\cos w|^m \quad (3.3)$$

$$s(w, m) = sgn(\sin w) |\sin w|^m$$

$$sgn(x) = \begin{cases} -1, & x < 0 \\ 0, & x = 0 \\ +1, & x > 0 \end{cases}$$

Equation (3.2) returns the (x, y, z) coordinates of a point on the superquadric's surface given longitude and latitude angles u and v , but it assumes that the axis of symmetry is the z -axis, as the equation was gathered from the source code of VTK's superquadric class, which implements these equations using Barr's formulation [52]. If the superquadric's axis of symmetry is defined around the y -axis, the application simply needs to swap the resulting y and z coordinates returned from the equation. An object is said to be axially symmetric if it appears unchanged when rotated around an axis [53]; for example, the axis of symmetry of the cup below is the y -axis since it appears unchanged when rotated around that axis (**Figure 50**).

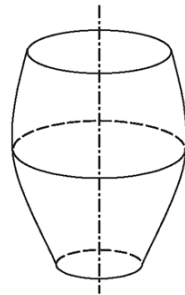


Figure 50: Axial symmetry around the y -axis. [53]

There are actually two sub-types of Superquadrics: Superellipsoids and Supertoroids. Superellipsoids are the solid shapes we have already seen above; Supertoroids, on the other hand, are very much the same but with a hole at its center similarly to a donut or wheel. They are still solid, water-tight meshes but the tubular structure of the donut-shape has a certain thickness. The implicit equation for a supertoroid, shown in (3.4), is very similar to its superellipsoid counterpart but with an extra thickness parameter.

$$a = \frac{1}{thickness} \tag{3.4}$$

$$tval = \left(|z|^{\frac{2}{e}} + |x|^{\frac{2}{e}} \right)^{\frac{e}{2}}$$

$$|tval - a|^{\frac{2}{n}} + |y|^{\frac{2}{n}} - 1 = 0$$

a (or alpha) is defined as the reciprocal of *thickness*, *tval* is an intermediate computation, e and n are longitudinal and latitudinal roundness (same as before). Once the (x, y, z) coordinates are plugged into the implicit equation, the resulting value will determine whether the point lies outside/inside or on the surface of the supertoroid mesh (greater than 0, less than 0 and equal to 0 respectively). In Aperio, the Superellipsoids are used for cutters and rods and Supertoroids are used as paths or rings in the exploder since they exhibit a ring-like shape (**Figure 51**).

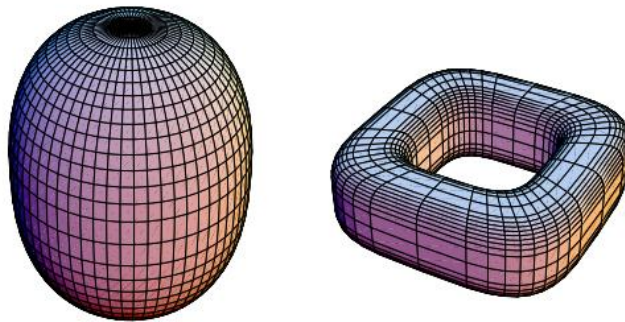


Figure 51: Superellipsoid and supertoroid examples. [51]

3.3 Constructive Solid Geometry

Cutaways in *Aperio* are performed using Constructive Solid Geometry (CSG) [54]. Constructive Solid Geometry is a common technique used in solid modeling to construct complex shapes by combining multiple meshes using Boolean operators. Interesting shapes can be formed simply by obtaining the intersection, union and difference of meshes (**Figure 52**). For example, a sphere and cube can be combined using a union ($\text{cube} \cup \text{sphere}$), a spherical chunk of the cube can be removed using a difference ($\text{cube} - \text{sphere}$) and finally we can obtain the spherical chunk that was removed from the cube using an intersection ($\text{cube} \cap \text{sphere}$).

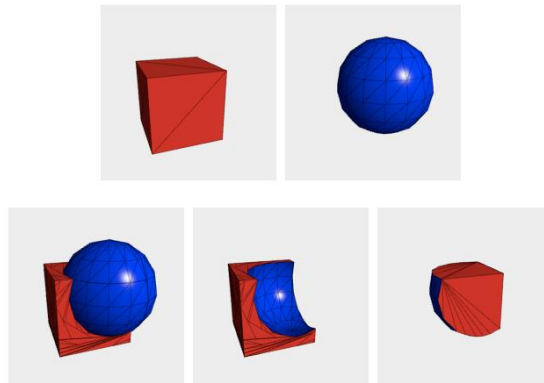


Figure 52: CSG; ($\text{Cube} \cup \text{Sphere}$), ($\text{Cube} - \text{Sphere}$) and ($\text{Cube} \cap \text{Sphere}$).

Cutaways in *Aperio* are implemented using exactly this method, performing a difference operation to obtain the resulting mesh after a sliced piece is removed and performing an intersection operation to obtain the sliced piece. In *Aperio*'s case, the cube in the figure above would be a mesh in the model and the sphere would be the superquadric cutter. Constructive Solid Geometry often assumes objects being operated on are *solid* or *water-tight* meshes, meaning they do not contain holes. However, to simulate meshes with holes, we can create a mesh containing an outer shell with some thickness that contours around the hole but the mesh would still remain water-tight (**Figure 53**).

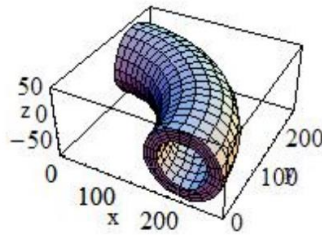


Figure 53: Water-tight mesh; outer shell with thickness example. [55]

In addition to *Aperio*'s cut-away feature, *Aperio* also contains a separate real-time previewer that previews cuts before they are performed; the difference between the previewer and the actual cut operation is that the previewer uses an image-based algorithm running in real-time in a fragment shader on the GPU, allowing computation to be fast enough so that previews can be shown interactively.

On the other hand, cut-aways performed on the mesh are done by actually separating the mesh's geometry into two new distinct meshes and requires holes to be filled (retriangulated) so that surfaces appear solid on the inside; this cut-away algorithm is performed on the CPU making it magnitudes slower than the image-based previewer (**Figure 54**). The previewer's GPU-based algorithm will be discussed later on in the chapter (with its image-based hole-capping algorithm); this section focuses on CSG libraries for performing CPU-based cut-aways that actually change the geometry and connectivity information of each mesh.

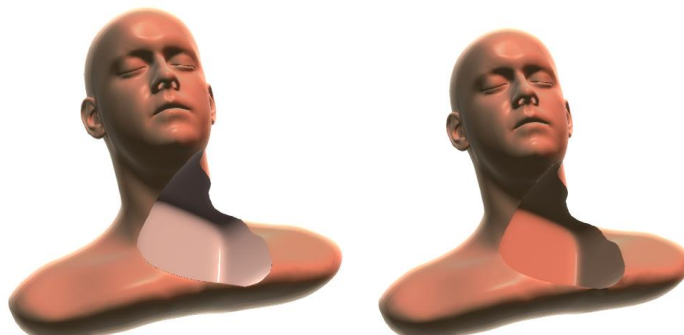


Figure 54: *Aperio*'s image-based preview and actual cut-away.

Several CSG libraries were compared before selecting the most suitable library for *Aperio*; the libraries considered include GTS (GNU Triangulated Surface Library), CGAL (Computational Geometry Algorithms Library), VTK's built-in class (*vtkBooleanOperationPolyDataFilter*) and Carve CSG library.

CGAL is an enormous library containing support for many reliable algorithms ranging from geometric computation to scientific visualization; CGAL also supports Boolean operations. Although algorithms are well supported, the library is excessive for our use since we only required the constructive solid geometry portion of the library. It performs Boolean operations accurately and robustly but performance is comparatively slower than Carve CSG [56].

GTS [57] is a smaller and simpler library than CGAL and also contains Boolean operations; Although simple and fast, Boolean operations are known to produce unstable results and the library itself has not been maintained in a while since it was last updated back in 2006 [57].

vtkBooleanOperationPolyDataFilter is a simple, out-of-the-box solution contained in VTK for performing Boolean operations, it is not designed to be optimized and performs magnitudes slower than other libraries such as GTS and Carve CSG. Finally, for *Aperio*, we selected Carve CSG as the constructive solid geometry library for performing cuts with the cutter and knife tool.

3.3.1 Carve CSG

Carve CSG [58] is a Constructive Solid Geometry library written by Tobias Sargeant from Walter and Eliza Hall Institute of Medical Research. It is designed to perform Boolean operations that are optimized for performance while still producing robust and correct output. It supports both closed and open surfaces and can be performed on meshes containing quads rather than just triangles as long as vertices of the polygon are coplanar. The algorithm is optimized and

tested on real-world data; as of Blender version 2.62, CSG operations in Blender’s Boolean Modifier are performed using Carve CSG (**Figure 55**). Autodesk Maya also integrated Carve CSG into their Boolean Operation tools as of Maya 2014. The author claims that “Carve is able to handle inputs of more than 100,000 vertices with ease”⁸. The most stable version of the library is from 2011 but regular updates have been made to the source code repository since June 2014. *Aperio* makes use of Carve CSG for performing cut-aways.

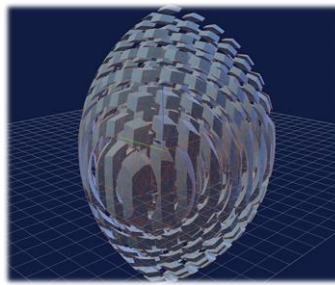


Figure 55: Carve; large model intersected with two shifted versions of itself (100k vertices).⁸

Since we use Visualization Toolkit (VTK) to store meshes, all meshes are contained within *vtkPolyData* objects which represent polygonal data in the form of vertices, edges and faces. In *Aperio*, we first convert the meshes from *vtkPolyData* objects into Carve’s *MeshSet* objects by constructing an instance of the *MeshSet* object then feeding in the mesh’s vertices as *carve::poly::Vertex* instances. Faces are then constructed by passing in the number of indices followed by a list of indices referencing the vertices. This conversion is performed in our own custom adapter class, *CarveConnector*. Once the conversion to *MeshSet* objects is complete, a *carve::CSG::csg* object can be created and its compute function can be called with the choice of operation; union, intersection, or difference that can be performed on the two *MeshSets*. The library supports the following list of operations: UNION, INTERSECTION, A_MINUS_B, B_MINUS_A, SYMMETRIC_DIFFERENCE. Assuming A is a mesh in the model, and B is the

⁸ <http://carve-csg.com/about>

superquadric cutter, we can use A_MINUS_B to obtain the resulting mesh after it is cut and use $INTERSECTION$ to obtain the piece that is cut-away (**Figure 56**).

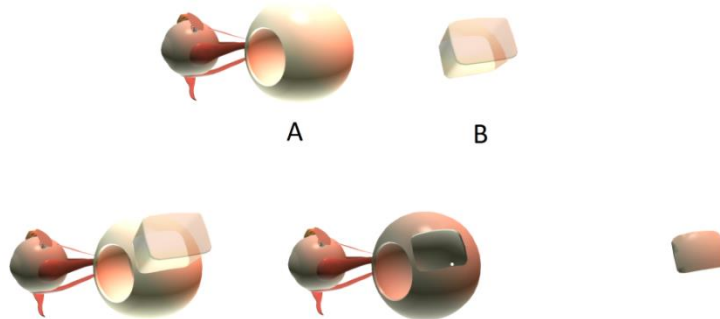


Figure 56: CSG with mesh A and cutter B. $(B \cup A)$, $(A - B)$ and $(A \cap B)$ respectively.

Although Carve CSG is able to perform constructive solid geometry on both open and closed meshes, for Aperio's purposes, we assume all meshes being cut are closed/solid on the inside (i.e. they are water-tight meshes); we use this assumption since holes are capped in the previewer. Before passing data into Carve CSG, we first feed it into a *vtkCleanPolyData* filter which basically merges duplicate vertices and removes degenerate cells (dangling points that are not used in any cells, triangles containing two merged points, etc.) We also assume meshes do not contain non-manifold geometry; non-manifold geometry is essentially geometry that cannot exist in the real world [59]. This is often avoided in models that are printed using 3D printers since the model will not print correctly if it contains non-manifold geometry. In mesh topology, manifold objects are usually 2-manifold, meaning that every edge should be connected by exactly two faces [59]. If an edge shares 0 faces, it is a stray edge. If it shares only one face, it is a boundary edge and the object is an open mesh (not closed as the edge is a boundary edge). If the edge shares 3 faces, then the third face is intersecting the closed mesh (**Figure 57**).

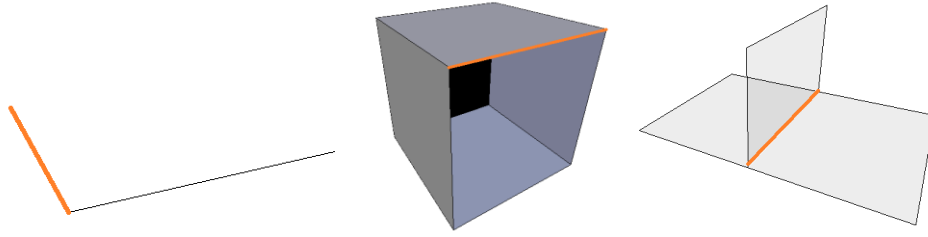


Figure 57: Non-manifold geometry; edge sharing (0 faces), (1 face) and (3 faces) respectively.

To solve the first case, we can simply remove stray lines, to solve the second case, holes can be filled (boundary edges disappear once holes are filled). The third case is a bit more difficult and may require manual editing of topology. Duplicate vertices in the same position are also considered non-manifold geometry but this has been taken care of in *vtkCleanPolyData* when duplicate points were merged. The diagram in **Figure 58** summarizes the process used in *Aperio* to perform CSG operations on two meshes (as previously described):

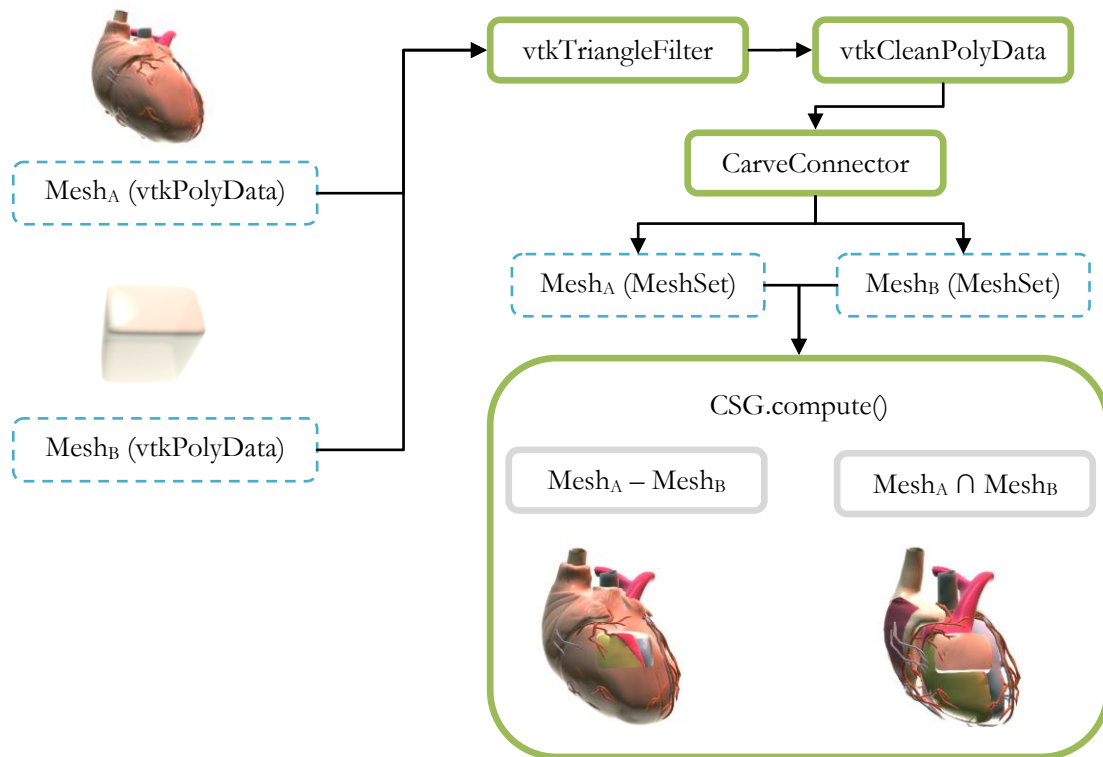


Figure 58: Aperio's steps for performing cuts with CSG.

3.4 Cutter

All of the tools in Aperio (the ring, rod, cutter and knife) follow a simple, unified approach for positioning and orienting. Essentially, the ring, rod, cutter and knife are all superquadrics that glide along the mesh's surface, and automatically reorient themselves according to the mesh's surface normals. The tools simply align their local up-axes or y-axes with the surface normals.

This makes it easy for the user to automatically place the tool in an orientation that makes sense; all they need to do is glide the tool along the surface and the tool will naturally orient itself respecting the mesh's geometry (**Figure 59**). The tool can then be planted in a desired position and applied onto the mesh.

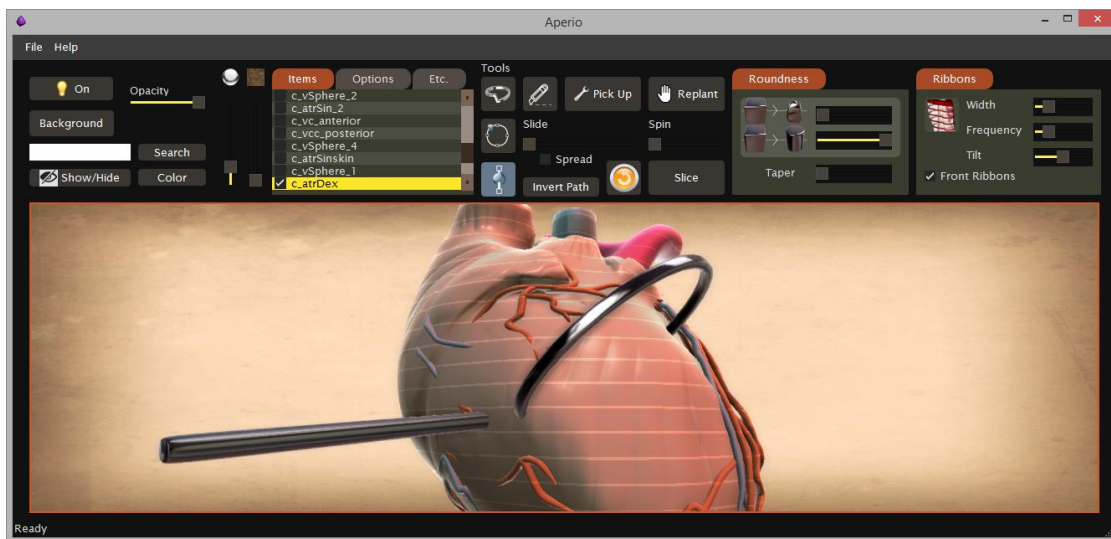


Figure 59: Aperio; ring and rod gliding on the surface of the mesh following surface normals.

In the case of the ring and rod, this technique is simply used for placement and orientation. Once the ring or rod is planted, they will intersect the selected meshes and the meshes will be associated with the tool. Because they are penetrated by the tool, the meshes can now easily slide along the tool's path like beads on a string; meshes that intersect will glide along the ring or

rod's defined path creating a constrained circular or linear explosion. In the case of the ring, the mesh will also rotate as it slides on the ring's path similarly to pages flipping/turning on a notebook ring.

The cookie cutter, on the other hand, is like a lens that glides along the surface of the mesh, cutting into many layers of meshes revealing the cross-sections of multiple objects (**Figure 60**). Since the algorithm is GPU-based, users can simply move the lens (i.e. cutter) along the mesh in real-time and see an interactive cut preview, carving into many layers of the mesh at once, revealing inner structures; this makes immediate and fluid exploration of the model possible.

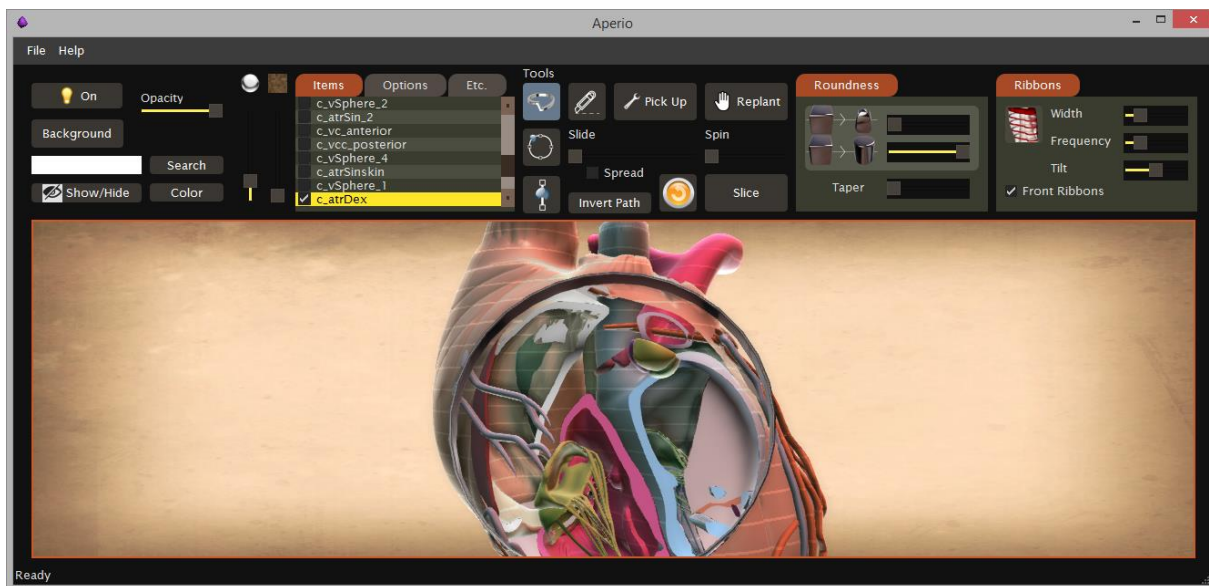


Figure 60: Aperio; drilling into a heart (interactive cut previewer).

3.4.1 *MeshMixer*

Our placement/orientation technique was inspired by several tools found in MeshMixer⁹, an application developed by Autodesk for easily (and creatively) manipulating existing meshes and

⁹ <http://www.meshmixer.com/>

transforming them into new meshes using simple drag-and-drop interactions. Specifically, Aperio's approach to placing and orienting tools was inspired by MeshMixer's drag-and-drop and brush select tools.

In MeshMixer, objects can be imported into the scene or pre-generated objects can be dragged into the scene. These objects are then automatically attached to existing objects, allowing users to drag the objects along the surface of the existing mesh. As objects are dragged, their local up-axes are aligned with the existing mesh's surface normals (**Figure 61**).

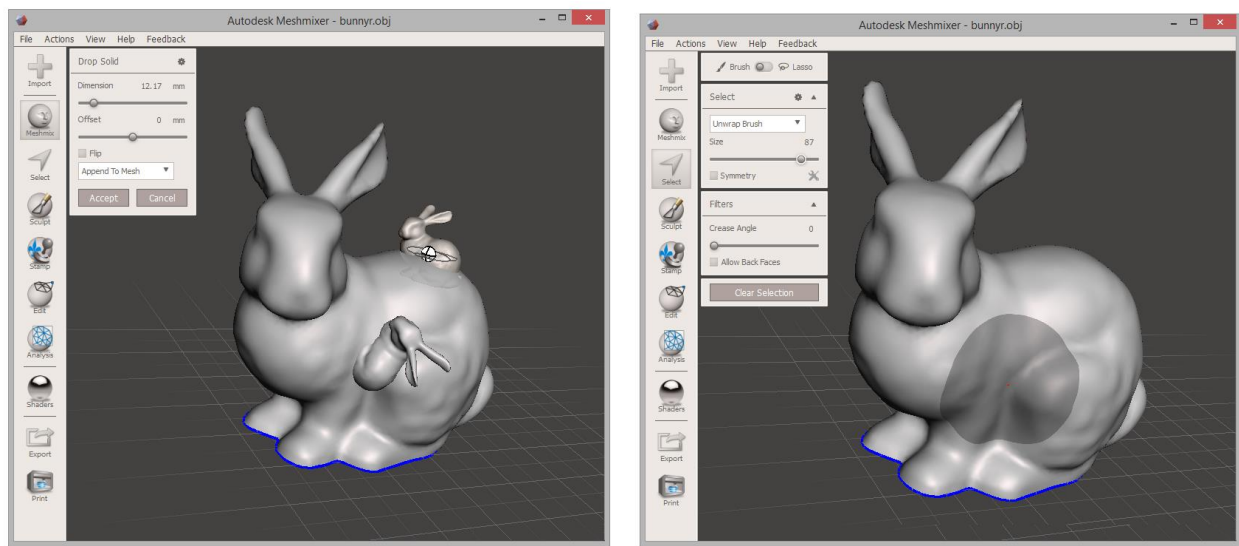


Figure 61: MeshMixer; dragging new objects into the scene and using brush select mode. [60]

In MeshMixer, face selection can be performed using a brush (in paint mode), by holding down the left mouse button and dragging across the surface of the mesh. The brush alone is simply rendered as a darkened circle that follows the contours of the mesh, reorienting itself as the mouse glides along the surface. Aperio uses both these ideas to create the interaction model for its lens (cut previewer). As the cutter (in the form of a superquadric) glides across the surface of the mesh, instead of rendering a darkened circle, we render a 3D superquadric tool and discard

mesh fragments inside of the superquadric so that parts of the mesh are cut-away allowing users to look inside and reveal hidden structures as shown in **Figure 60**. Since the objects being cut are assumed to be solid, naïvely discarding fragments is not enough because underneath the mesh’s surface it is hollow and empty. This means we need a capping algorithm to fill in the holes of the surface so that it appears solid to the user, just as it would be when performing the actual cut using Carve CSG [58]; this requires a multi-pass rendering algorithm.

3.5 Capping Algorithm

Several approaches to implementing the GPU-based capping algorithm were considered. The first option was to use an existing library such as OpenCSG to perform actual CSG operations using an image-based algorithm. An alternative option was to create our own custom capping algorithm and integrate it into Aperio’s existing multi-pass rendering pipeline.

3.5.1 *OpenCSG*

OpenCSG [61] is a C++ library that performs image-based CSG rendering using Frame Buffer Objects (FBOs) in OpenGL. The library uses both the depth and stencil buffers on graphics hardware to perform CSG; it implements stable CSG algorithms including the Goldfeather algorithm [62] and the Sequenced Convex Subtraction (SCS) algorithm [63] (**Figure 62**).

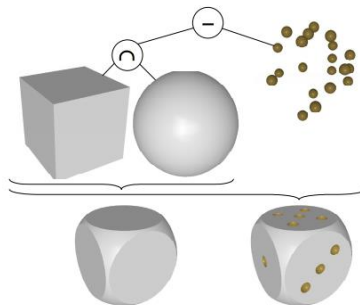


Figure 62: OpenCSG; modeling a dice. [61]

This approach would deliver accurate and robust results since it relies on standard and reliable CSG algorithms to generate correct images. For Aperio, since we aren't dealing with many arbitrarily shaped cuts, (as we only use simple superquadric-shaped cutters – and we even have their implicit equations available), it was easier to implement a cheap, light-weight algorithm that discards and caps fragments using only that equation. The algorithm would first determine whether a point on the mesh was inside the superquadric's surface and if so, it would discard its fragment and mark the discarded fragments in a texture; in a subsequent render pass the GPU would read the contents of the texture and use it to determine which fragments needed capping. OpenCSG would be useful for more complex shaped cuts where the implicit equations are not available, but since the equations are available, we opted to implement a custom capping algorithm.

3.5.2 *Custom Capping*

Our capping algorithm requires two render passes: a pre-pass to render information to textures and a subsequent main pass that reads in texture data and renders the final scene. Both passes will first discard all selected meshes' fragments that are inside the boundaries of the superquadric cutter. This means that the cutter (previewer) will essentially “cut” into all the selected mesh layers and reveal structures underneath. Since we are using surface data, cutting into layers will actually reveal a *hollow* shell underneath (**Figure 63**); this is why we need a capping algorithm, to fill in the holes of the shell as we currently assume all meshes are solid.

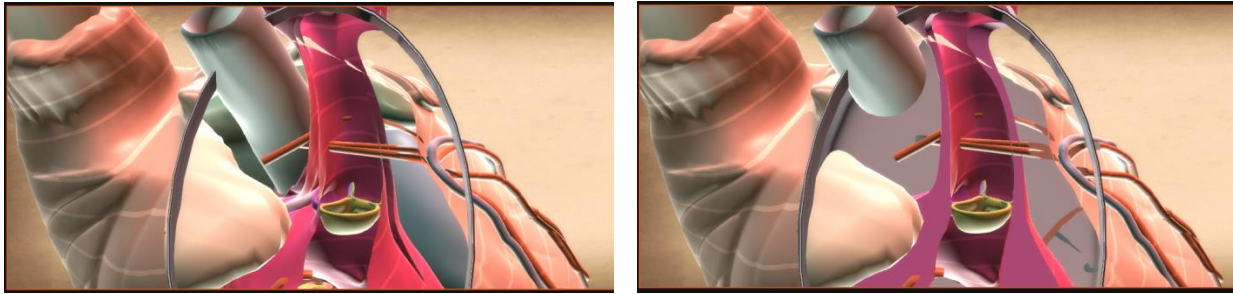


Figure 63: Aperio; no capping (hollow) vs with capping algorithm (solid).

In the pre-pass, after discarding the selected meshes' fragments, the depths of all selected meshes are rendered to a texture using Frame Buffer Objects (FBOs) along with the depths of the superquadric cutter. Specifically, we need to output 3 depth textures and 1 colour texture in the pre-pass (**Figure 64**).

1. The Selected Meshes' Depths for front-facing fragments
2. The Selected Meshes' Depths for back-facing fragments
3. The Selected Meshes' Colours
4. The Superquadric Cutter's Depths for back-facing fragments

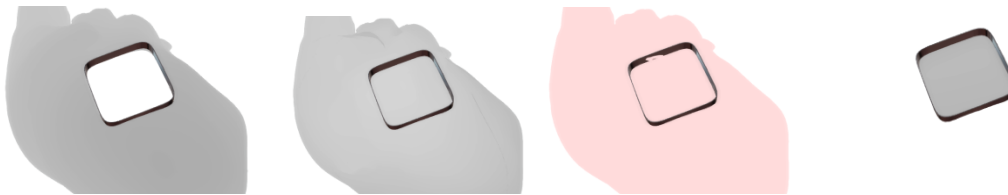


Figure 64: Selected mesh's front and backfacing depths, colour and cutter's backfacing depth.

We do not need to store front-facing fragments of the superquadric cutter because it would just cover up what we cut-away entirely which would make our cutter (previewer) pointless. We only need back-facing fragments of the superquadric since they are going to be rendered and used as the cap to fill up holes at the bottom of the cutter closing up the hollow shell created by discarding mesh fragments.

Depths are linearized between the values of 0..1 (0 being closest and 1 being farthest away from the user– black to white); they are encoded using an RGBA colour in the form of a (R,G,B,A) vector.

In the pre-pass, the depths of all front-facing fragments of selected meshes are saved into textures (first image in **Figure 64**) but by default we set all discarded or non-filled fragments to the RGBA colour (1, 1, 1, 1) – or infinite depth (purely white) so that they are distinguishable from actual depths of the mesh. In the subsequent pass we simply check if a fragment was discarded or non-filled at a position by checking if the depth at that position is infinite. If so, it likely requires capping.

In the main pass, the 4 textures are read in again (3 depth textures and 1 colour texture), except in this pass, the scene is actually rendered. To determine which back-facing fragments on the superquadric will be used to cap holes, we run **Algorithm 1** in the fragment shader:

Algorithm 1 Capping holes in selected meshes

```

for each backfacing fragment of the superquadric do
    if       $depth\_selectedmesh\_frontfacing \geq 1 \ \&\&$ 
            $depth\_superquadric\_backfacing \leq depth\_selectedmesh\_backfacing$ 
    then
        render fragment ( $colour\_selectedmesh$ )
    else
        discard fragment
    end if
end for

```

Basically, for every back-facing fragment on the superquadric cutter, if the front face of the selected mesh is discarded or non-filled ($depth_selectedmesh_frontfacing$ is infinite, meaning it is purely white or ≥ 1), and the depth of the superquadric cutter's back faces ($depth_superquadric_backfacing$) is closer (or less than) the depth of the selected mesh's back faces ($depth_selectedmesh_backfacing$); then this means the superquadric cutter hasn't cut past the end or back of the mesh), then render it. Otherwise, do not render the fragment. This algorithm will only render capping fragments that are not past the end of the selected meshes but are also within the boundaries of the superquadric cutter. The fragments of the superquadric cutter that follow this condition are then rendered using the selected mesh's colour ($colour_selectedmesh$) at that fragment (to make it appear as if the superquadric cutter fragments belong to the selected meshes themselves). The fragments are then rendered as usual using a local illumination model with the backfacing superquadric's surface normals and processed with Screen Space Ambient Occlusion (SSAO) [49] and Fast approximate anti-aliasing (FXAA) [50] (**Figure 65**).

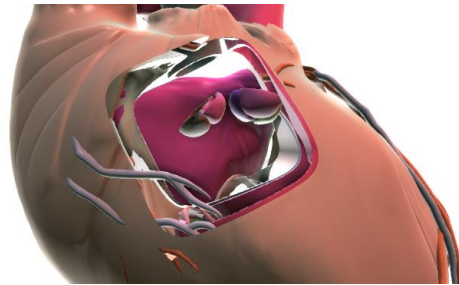


Figure 65: Aperio; resulting image of capping algorithm.

3.6 Real-time Cut Previewer

To simulate the real-time cut previewer, (hiding fragments as the cursor slides along the mesh's surface), we use the superquadric's implicit equation to determine fragments to discard. Since the superquadric is translated and rotated according to the mesh's surface normals we also need

to store the transformation matrices for each superquadric; the positions are stored in world-space coordinates as a vector, (x, y, z) and the orientation is stored as 3 normalized vectors (a right, up and forward vector) representing the superquadric's local axes defining its orientation.

(Figure 66)

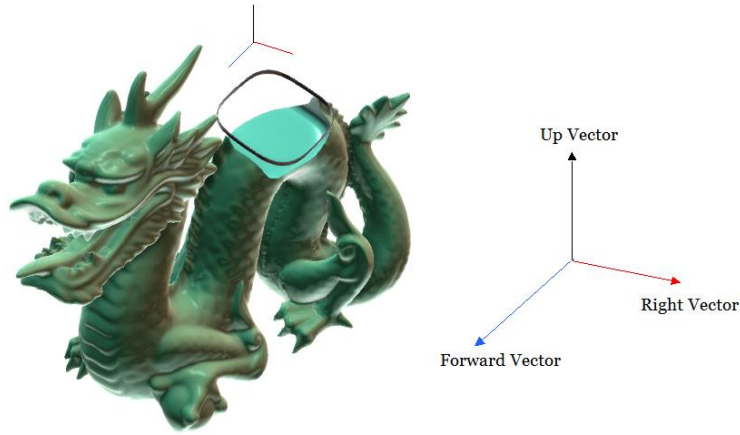


Figure 66: Superquadric's orientation as defined by Right, Up and Forward vectors.

The superquadrics' transformed points, (x', y', z') are then computed by multiplying the superquadric's local-space points, (x, y, z) by the transformation matrix in (3.5).

$$(x', y', z') = \begin{bmatrix} right_x & up_x & forward_x & position_x \\ right_y & up_y & forward_y & position_y \\ right_z & up_z & forward_z & position_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} \quad (3.5)$$

In the fragment shader, the *right*, *up*, *forward* and *position* vectors are passed in as uniform variables; since the implicit equation of a superquadric assumes points in local-space, to determine if a point on a mesh is inside a transformed superquadric (the rotated and translated version of the superquadric on the mesh's surface), we first have to untransform the mesh's point into local-space; we do this by multiplying the mesh's point by the inverse of the superquadric's transformation matrix (shown above) so that points on the mesh can easily be plugged into the

implicit equation (since they will be local coordinates) to determine if it lies inside the superquadric cutter and whether it should be discarded. The fragment shader discards the selected meshes' fragments using **Algorithm 2**:

Algorithm 2 Discarding selected mesh's fragments inside superquadric cutter

Inputs: *mesh_point*

right, up, forward, superquadric_position ▷ Superquadric parameters
theta, phi

for each fragment of selected meshes **do**

transformation_matrix \leftarrow matrix4x4(*right, up, forward, superquadric_position*)
mesh_point_local \leftarrow inverse(*transformation_matrix*) \cdot *mesh_point*

▷ Implicit equation of the Superquadric (local-space) :

val \leftarrow pow(pow(abs(*mesh_point_local.z*), 2.0/*theta*) + pow(abs(*mesh_point_local.x*), 2.0/*theta*), *theta/phi*) + pow(abs(*mesh_point_local.y*), 2.0/*phi*) - 1.0;

if *val* < 0 **then**

 discard fragment

end if

end for

The algorithm takes in *mesh_point*, (world coordinates of a point on the selected mesh's surface corresponding to the current fragment), the superquadric cutter's pose (*right, up, forward, superquadric_position*), and *theta, phi* for the superquadric cutter. To get local mesh coordinates, we multiply the *mesh_point* world coordinates by the inverse transformation matrix. The implicit equation for the superquadric shown above (stored in *val*) is the same one from the Superquadrics section of this chapter, from equation (3.1). Mesh fragments within the superquadric cutter's boundaries are discarded using this algorithm.

The superquadric cutter can also be stretched and scaled; the cutter can be deepened to create deeper cuts (scaled along the y-axis since the superquadric aligns its y-axis with the surface normal) or even stretched or resized along the x-axis or z-axis to create wider or lengthier cuts.

3.7 Explosion on a Constrained Path

Aperio also contains a ring and rod tool in addition to the cutter. The cutter tool allows users to explore the model by peering into multiple layers of the mesh in real-time (like a lens) as it slides across the mesh's surface. When the cutter is finally planted, the tool can then perform the actual *slice* action which cuts the mesh into several pieces (using the Carve CSG library [58]); the meshes will then be retriangulated so that their connectivity information is recreated for the new meshes.

On the other hand, the ring and rod tools are used for translating and rotating meshes along a constrained path. They can be combined with the cutter and be used to hinge open or explode new pieces generated from the cutter tool, sliding them along the ring or rod's path.

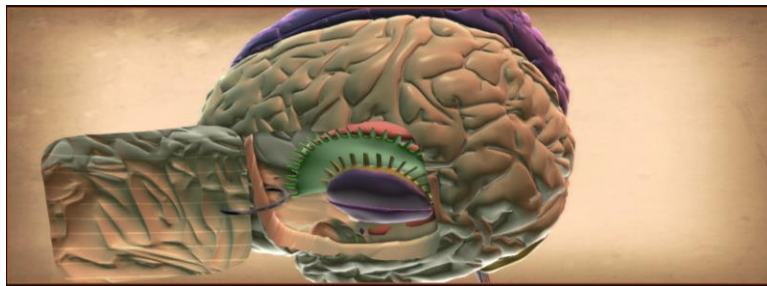


Figure 67: Aperio; hinging newly sliced piece of the left cerebral cortex using a ring.

In **Figure 67**, the cutter is first used to cut out a rounded rectangular piece of the left cerebral cortex. A ring is then placed onto the sliced piece as a way to hinge the sliced piece open and away from the rest of the cerebral cortex. The sliced piece then slides along the path of the ring similarly to how pages in a notebook would flip or turn on a notebook ring; the page would be

constrained to the notebook ring, but would rotate when sliding around the ring. One novel contribution of this thesis is the use of a small or narrow ring to create hinges for opening up and separating parts of a mesh, similarly to the behaviour of a door hinge.

The ring is not limited to creating hinges; its main use is to create constrained explosions of multiple meshes along a path for visualization purposes (**Figure 68**); since the ring itself is simply a supertoroid (a specific type of a superquadric), we have access to its parametric equation as well and can use it to determine all points on its path; it also inherits all the qualities of a superquadric, meaning we can change its phi and theta roundness to make the path more or less rounded. Using the transformation matrices, we can also stretch out the ring's path and orient it by spinning the ring around its centre or tilting it in different directions.



Figure 68: Aperio; creating a constrained explosion for multiple meshes along a ring.

Mesh parts that slide along the ring can also be spread apart or kept intact together. The rod is similar to the ring except that it penetrates the selected meshes and creates a linear explosion (along a straight-line path instead of a circular one). Meshes that intersect the rod can slide along its linear path and even be spun around the rod's axis (**Figure 69**), revealing the other half of the mesh (the side that is facing away from the camera).

Using this spinning technique, we can reduce clutching [64], which is the need to constantly rotate the scene in order to reveal the other half of the scene, (due to the limited perspective or

field of view). The term, clutching is defined as “momentary recalibration to avoid running out of input area” as defined by Casiez [64].

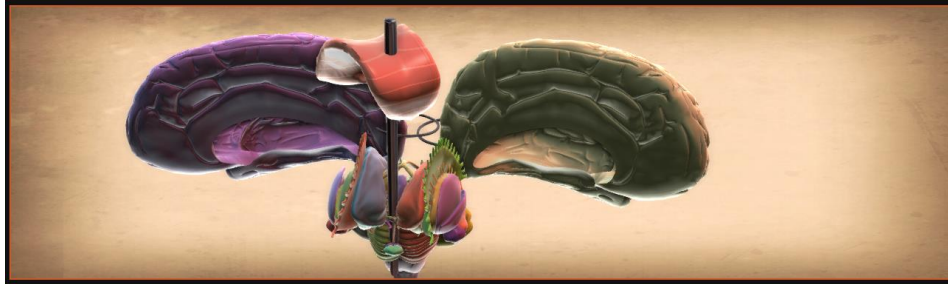


Figure 69: Aperio; placing a rod on the corpus callosum and spinning it.

Another feature implemented in Aperio to reduce clutching as well as help the user with positioning is the use of an Oriented Bounding Box (OBB) [65]; this technique tries to help the user place the ring by automatically aligning the ring’s orientation to the selected mesh’s OBB. Oriented Bounding Boxes (OBB) are boxes that define the boundaries of a mesh, but as the name implies, it is oriented with respect to the mesh’s geometry, as opposed to Axis-Aligned Bounding Boxes (AABB) [66], which are simple bounding boxes that align with the world’s (x, y, z) axes and have no orientation. Since the ring will snap to the Oriented Bounding Box’s surface it might be useful for users as they won’t have to deal with spinning and tilting the ring to get an acceptable orientation; although in some cases it might be useful for users to have the capability to place the ring with precision.

3.7.1 Ring

Since the ring is simply a supertoroid, we have access to its parametric equation. As defined in the Superquadrics section of this chapter– equation (3.2), we can use the parametric equation and input u and v parameters (longitude and latitude angles), to obtain a resulting point on the surface of the supertoroid as a vector of (x, y, z). In VTK, there is a class, *vtkSuperquadricSource* that

contains a method called *evalSuperquadric* which evaluates the parametric equation of a superquadric and gives back a point on the superquadric as well as its point normals.

By inputting u and v (theta and phi angles – or longitude and latitude angles) and roundness parameters for the superquadric, along with its dimensions, we can obtain a resulting (x, y, z) point on the superquadric surface corresponding to the longitude and latitudinal angles inputted (we can also obtain the normals at each point on the superquadric) We can use this function to generate a number of points on the ring to be used for the path simply by setting phi or $\varphi = 0$ and calling the function constantly n number of times with theta or θ ranging from $-180\dots180$ degrees or $-\text{PI}\dots\text{PI}$ radians (**Figure 70**). Doing this, we can obtain n points around the supertoroid (or ring) to be used for the sliding path.

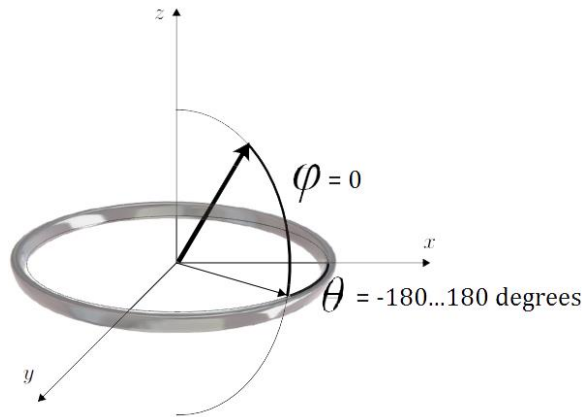


Figure 70: Finding points for the sliding path on the supertoroid.

After finding n points on the supertoroid (in our case $n = 360$ points), we use *evalSuperquadric* to also obtain the superquadric's normals at each point since it is part of the function's signature. We can then determine what orientation to give the meshes (at each point along the path) as it slides on the ring, simply by using the superquadric's normals combined with the tangents and bitangents at each point on the path (**Figure 71**).

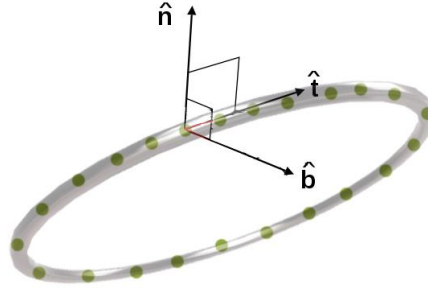


Figure 71: Using normals (\hat{n}), tangents (\hat{t}) and bitangents (\hat{b}) to orient meshes.

The normals are available by calling *evalSuperquadric*; to obtain the tangents at each point on the path, we can simply create a vector from the current point to the next point on the path (points are indicated as green dots in **Figure 71**). Once we have both the tangent and normal, the bitangent is simply the cross-product of the two since the cross-product, $(\hat{t} \times \hat{n})$ would give us the vector (\hat{b}) – the bitangent which is perpendicular to both the normal (\hat{n}) and tangent (\hat{t}).

The normal, tangent and bitangent vectors can then be used to orient the sliding meshes by plugging them into the transformation matrix as the up, forward and right vectors respectively. This is the same transformation matrix as in equation (3.5) that we used to orient the superquadric cutter, but this time it is used to define the orientation of meshes as it slides on the ring. To determine the intersection point between the ring and the mesh, we use a class *vtkIntersectionPolyDataFilter*, which returns the cells that intersect between two different *vtkPolyData*s (polygonal mesh data); we pass in the *vtkPolyData*s (polygonal mesh data) of both the supertoroid ring and mesh. This intersection point is then used as the point at which the mesh will rotate around (the pivot point) as it slides along the ring. If the ring does not intersect the mesh at any point, *Aperio* simply uses the center of the mesh's Oriented Bounding Box (OBB) [65] as an intersection point.

3.7.2 Rod

The rod performs similarly to the ring except its path is simplified and constrains movement to a linear path. This means the normal, tangent and bitangents are not required and n points on the path can be generated simply by generating n evenly spaced points along a vector of a specified magnitude. The magnitude is determined by the user as they can elongate the rod to any particular length they wish. The direction of the rod can be controlled by the user by tilting it in different directions or automated by gliding it along the mesh's surface; it will automatically align its y-axis with the mesh's surface normals. The rods are basically elongated superquadrics (superellipsoids) so they also exhibit superquadric properties such as the capability of changing phi and theta roundness. Lengthening or deepening the rod simply scales the rod along its y-axis.

Multiple meshes that intersect the rod can slide together along the rod's path or even be spread apart along the linear path, creating linear-constrained explosions. Multiple meshes can also be spun around the rod's axis, or even fanned like an arraying of cards (**Figure 72**); fanning simply rotates each mesh on the rod a different amount depending on its order. Parts that are farther out on the rod will be fanned a larger amount than parts that are closer to the start of the rod.



Figure 72: Spreading meshes apart on a planted rod vs fanning meshes.

3.8 Tool Appearance

The metallic or steel appearance of all tools in Aperio (ring, rod, cutter and knife) is implemented using MatCaps (which stands for Material Captures). MatCaps is a popular

technique used in many modelling and sculpting packages such as Pixologic's zBrush¹⁰ and Luxology's Modo¹¹ used to apply material properties to meshes (**Figure 73**). The technique basically uses Spherical Environment Mapping (SEM) [67], a traditional technique used to simulate reflection mapping by applying a texture of an environment to the shiny reflective surface of an object in the scene. MatCaps reuses this idea but instead of using it to simulate shiny surfaces, it can be used to simulate any type of material, including matte objects (in real-time). A texture of a lit sphere is used for mapping but instead of only reflective spheres, a more diffuse colour can be used to simulate other materials such as skin. The texture contains a sphere that has been rendered with global illumination to essentially "fake" the look of a material. The added benefit of using matcaps is that UV coordinates need not be available as they are automatically generated from the surface normals of the mesh.

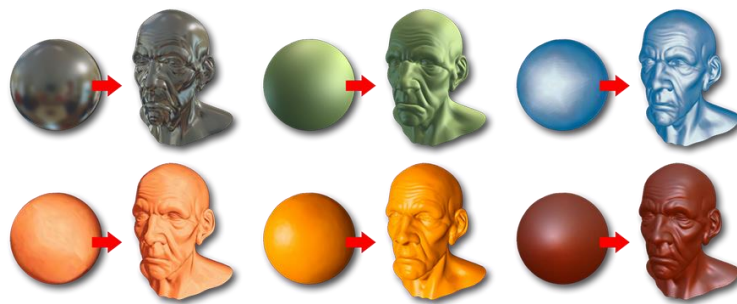


Figure 73: Matcaps; result of applying them to models in Luxology's Modo. [68]

Generation of matcaps is pretty much identical to spherical environment mapping and is defined using **Algorithm 3** for all superquadric tools in Aperio:

¹⁰ <http://pixologic.com/zbrush/downloadcenter/library/>

¹¹ <http://www.thefoundry.co.uk/products/modo>

Algorithm 3 Matcap Shader for Superquadric Tools

Inputs: *normal*, *position*, \triangleright interpolated superquadric normals and positions in view-space
matcap \triangleright *matcap* is a texture (of an illuminated sphere)

for each fragment on the superquadric tool **do**

\triangleright Standard Spherical Environment Mapping Formula: [67]

$r \leftarrow \text{normalize}(\text{reflect}(\text{position}, \text{normal}))$

$m \leftarrow 2 * \text{sqrt}(\text{pow}(r.x, 2) + \text{pow}(r.y, 2) + \text{pow}(r.z + 1, 2))$

$uv \leftarrow r.xy / m + 0.5;$ \triangleright Generate UV coordinates

$\text{final_color} \leftarrow \text{texture2D}(\text{matcap}, uv)$

render *final_color*

end for

The algorithm is performed on every fragment of the superquadric tool and takes in the superquadric's *normal* and *position* at each fragment in view-space. It also takes in a *matcap* texture. For every fragment, the algorithm obtains a reflection, *r* using the superquadric's *position* (as the incident vector) reflecting across the superquadric's *normal*. The reflection vector, *r* is plugged into a sphere's implicit equation, *m* and used to determine the *uv* coordinates. The surface can then be texture mapped with the matcap texture using the newly generated *uv* coordinate. The results of the matcaps are shown in **Figure 74** (applied to a ring and rod) with an alternative example of using a red paint-like matcap.



Figure 74: Matcaps applied to a ring and rod (metal and paint materials).

3.9 Ribbon Cuts Algorithm

The cutter tool (and its previewer) allows users to drill down into multiple layers of meshes in real-time and lets the user interactively move the superquadric tool along the surface of the mesh to visually explore hidden objects underneath those layers. Unfortunately, since it essentially creates cut-aways, it suffers from the same problems that traditional cut-aways suffer from; that is, context of the removed piece is lost since they are (in its entirety) visually removed from view. This means we cannot mentally reconstruct the shape or contours of the removed pieces. Ribbon cuts are one solution to this problem; they preserve an outline of the removed piece by rendering its surface as thin ribbon strips (**Figure 75**) rather than completely removing the surface from view. This allows the user to cut away objects in the scene to reveal hidden structures but still preserve an outline of the objects being cut.

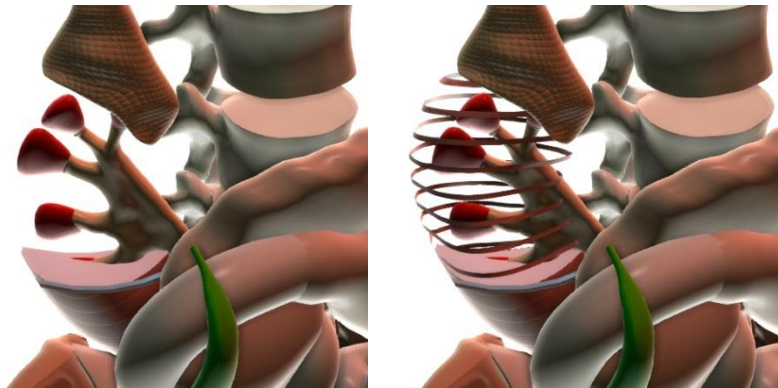


Figure 75: Aperio; entire cut-away vs ribbon cuts (into the kidney).

Aperio takes the idea of Ribbon Cuts one step further and implements a real-time ribbon cut previewer in the fragment shader; this allows the user to freely move the cutter around the surface of the mesh and preview cuts while still being able to see the contours or shapes of the layers they have drilled into. They can also resize the cutter interactively and see the boundaries of the ribbon cuts change; they can also modify the ribbon's width, frequency and tilt. Ribbon

cuts might be less effective when drilling into multiple layers since the ribbon-slices can be noisy when multiple mesh surfaces intertwine.

The ribbon cuts implementation (in the fragment shader) is a slightly modified version of an equation obtained from Unity’s Shader Reference [69] for creating world-space slices of a model; **Algorithm 4** shows Aperio’s implementation:

Algorithm 4 Ribbon Cuts Fragment Shader

Inputs: *mesh_point_local*

ribbon_width, ribbon_frequency, ribbon_tiltAngle ▷ Ribbon parameters

right ▷ Superquadric parameters

for each mesh fragment within the superquadric cutter’s boundaries **do**

if *ribbon_previewer* is on **then**

$tilt_matrix \leftarrow rotation_matrix(ribbon_tiltAngle, right)$

$ribbon_direction \leftarrow tilt_matrix \times right$

 ▷ Ribbon Equation:

if $fract(dot(mesh_point_local, ribbon_direction) * ribbon_frequency) -$

$ribbon_width < 0$ **then**

 render_fragment

else

 discard_fragment

end if

end if

end for

The algorithm takes in the local-space coordinates of points on a mesh (*mesh_point_local*), ribbon parameters (*ribbon_width, ribbon_frequency, ribbon_tiltAngle*) and the superquadric cutter’s *right* vector which is used to tilt ribbons rotating them around the cutter’s right-vector.

The ribbon cuts algorithm above runs for every mesh fragment inside the superquadric cutter's boundaries; if the *ribbon_previewer* is on, we first find the direction to orient the ribbons (*ribbon_direction*) by multiplying the cutter's currently oriented *right*-vector by a *tilt_matrix* (the tilt matrix tilts the ribbons by a tilt angle, *ribbon_tiltAngle* around the cutter's *right* vector – so that ribbons tilt forward or backwards). An implicit equation containing a *fract* (computing the fractional part of a number) is combined with a *dot* product (of *mesh_point_local* and *ribbon_direction*) along with ribbon parameters (*ribbon_width*, *ribbon_frequency*, *ribbon_direction*). If the equation returns a value below 0 then we render the fragment since it will be part of the visible ribbons; otherwise we discard the fragment since it is part of the sliced ribbons (the parts that are cut-away).

3.10 Knife Tool

The knife tool is the final tool in Aperio and is actually the only tool that has a slightly different interface since it is meant to create incisions and cannot be picked up as it is mainly used for slicing a mesh into two separate halves. We can slice a mesh into two halves and then using a ring, we can hinge one half of the mesh open by placing a ring on that half (**Figure 76**).

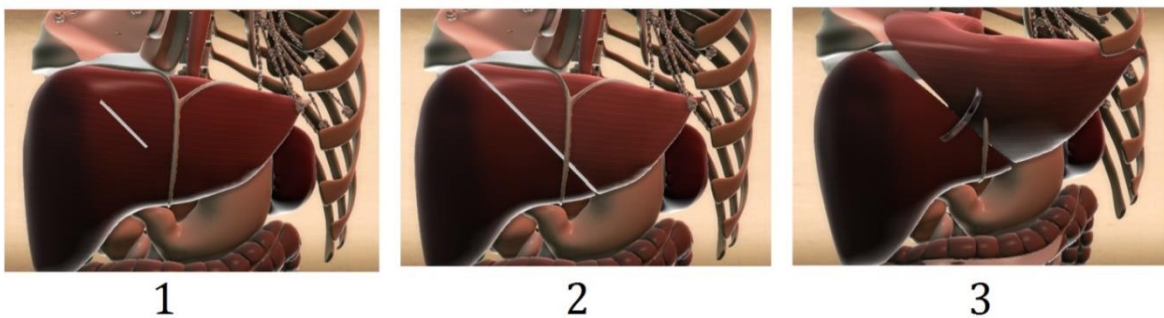


Figure 76: Aperio; using a knife to cut the liver in half and hinge one half open with a ring.

Once the user selects the knife tool, they can make an incision by drawing a line. Once they release the left mouse button the incision extends the cut such that it cuts all the way through the

mesh in depth and also extends the length of the line in both directions. The knife tool is rendered as two tapered blades that cuts through the mesh. Tapering was added by simply entering an additional line into the parametric equation (assuming the superquadric has axial symmetry around the y-axis– switch y and z for z-axis symmetry); this is shown in (3.6).

$$x' = (taper * y / dims_y + 1) * x \quad (3.6)$$

$$z' = (taper * y / dims_y + 1) * z$$

$$y' = y$$

This transforms the inputted (x, y, z) points on the superquadric into tapered (x', y', z') coordinates. Points on the superquadric along the local y-axis will remain unchanged since $y' = y$ that is, in tapering, its y-axis would be aligned with the mesh's surface normal so its length would remain unchanged but x and z points will transform using the tapering formula above, where $dims$ is the dimensions of the superquadric and $taper$ is the amount of tapering applied (**Figure 77**).

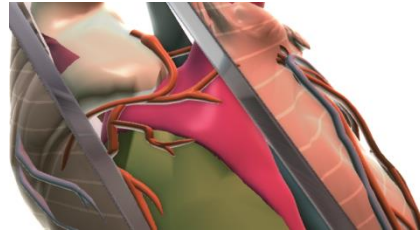


Figure 77: Aperio; tapered cutter blades (spreaders) used to widen the cut.

The knife tool uses Carve CSG [58] to perform the cuts; it feeds into Carve CSG two datasets: the mesh and the superquadric knife (which is a narrow lengthened cutter). The two *vtkPolyDatas* (polygonal mesh data) are first converted into Carve's *MeshSets* using the *CarveConnector* class we developed. *MeshSets* are data structures used in Carve for storing geometry data for the polygonal meshes. We then perform an A_MINUS_B CSG operation,

feeding in the mesh and superquadric knife (A = mesh, B = superquadric knife). Since the superquadric knife cuts all the way through the mesh, we should end up with two resulting disconnected regions of the mesh but by default the application will treat the two regions as one mesh. To separate them, we need to separate connected regions into different *vtkPolyDatas* (polygonal mesh data). We do this by using the *vtkPolyDataConnectivityFilter* class which is a class that extracts different regions of the mesh based on geometric connectivity. After running the data through the filter, two separate *vtkPolyDatas* (polygonal mesh data) were returned for the two halves of the incision.

3.11 Control Panel and Tooltip Interaction

The different tools in Aperio, (rod, ring, cutter and knife) can all be created by simply clicking the respective buttons on the control panel. Aperio’s control panel display is shown in **Figure 78**. Once a tool is created, it becomes the current “tooltip”. The user can then fluidly and smoothly slide the tooltip along the surface of the mesh using the mouse. The tool will automatically orient itself to match the mesh’s surface normal. The tooltip can also be optionally “planted” at any time to establish a view, and “picked up” at a later time. Buttons for creating tools, planting and picking up tools are located in the control panel display.

The panel also contains a list of meshes (items) currently in the scene and GUI sliders, including a **Slide** slider to slide selected meshes along the ring/rod’s path and a **Spin** slider to spin meshes around the rod’s axis; meshes can also be fanned similarly to an arraying of cards. A **Spread** toggle is available to spread the meshes apart simulating an exploded view of meshes along the path. A **Slice** button can be used with the Cutter tool to slice off a part of the mesh or used with the Knife tool to split a mesh into two separate halves. Each tool’s superquadric shape can be interactively modified by changing its roundness parameters in both Phi and Theta directions and superquadric tools can also be tapered. Ribbons can be toggled on and off and ribbon parameters (width, frequency and tilt) can be modified interactively and in real-time showing changes to the ribbons slices within the cutter as the tool is slid across the surface of the mesh.

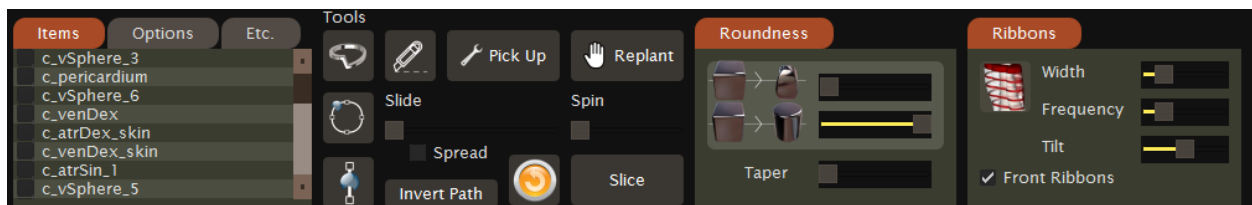


Figure 78: Aperio’s control panel.

4 Evaluation and Results

In this chapter, we present a user study performed on Aperio to evaluate its effectiveness for occlusion management. Two versions of Aperio were compared in the study, a free-form version (without tools) and a version with tools (**Figure 79**). The user study focuses on two criteria between both versions of Aperio: perception (how visually understandable the system is and its feasibility for occlusion management) and user interface (ease of use/control for the user).

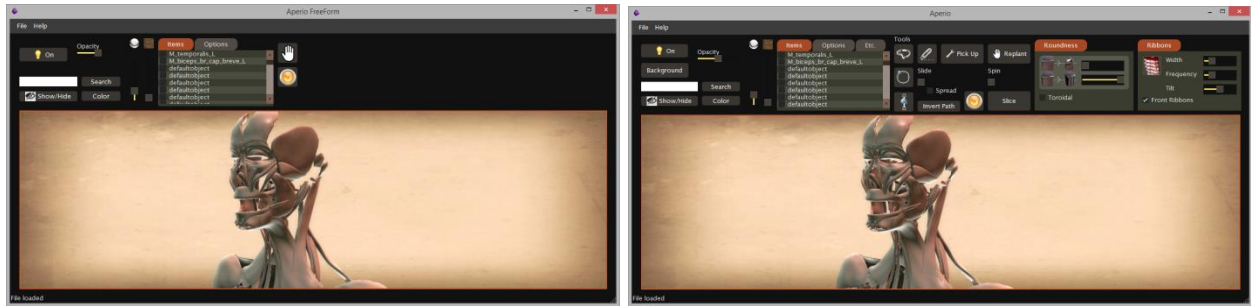


Figure 79: Aperio FreeForm (no tools) vs. Aperio Tools.

4.1 User Study

We performed the user study on 18 participants; each had a one-hour session where they were asked to perform visualization tasks in Aperio – particularly, matching tasks where each user was given images of a scene to replicate using either the mechanical tool version of Aperio or its freeform version. The participants were aged 19-34 with an average age of approximately 26 years; there were 14 males and 4 females. Two participants had extensive 3D modeling experience (decided by long-term use of modeling or animation packages for paid projects) and one was a Medical Radiation Technologist. The average mouse usage per week (as inputted by users) was approximately 53 hours, with 9 hours a week playing video games and 1.3 hours using 3D modeling software.

For the user study, two versions of Aperio were created; an existing version of Aperio using the ring, rod and cutter tools (referred to as “Aperio Tools”) and a second version without tools but with a very similar mouse and control key interface (referred to as “Aperio FreeForm”). Both versions shared mouse and key controls for performing common actions such as selecting a model and navigating the scene. Instead of using mouse and control keys to position the rod and ring and using GUI sliders to slide the mesh along the rod and ring’s path, Aperio FreeForm only uses mouse and control keys to directly manipulate the model. The freeform version of Aperio is meant to resemble traditional modeling packages providing 3D modelers with a larger degree of freedom, allowing objects to be freely translated and rotated along multiple axes simultaneously.

In addition, to ensure some measure of fairness, we decided to add functionality to Aperio Freeform to make it simpler for users to restore a model to its initial position and orientation; this way, users would not need to spend too much time trying to get the model into the exact pose. We used the Oriented Bounding Box (OBB) coordinate axes of a model and measure the distance from the current position of the model OBB center to the initial position of the OBB center. We also measure the angular difference between the current orientation of the model OBB coordinate axes and the initial OBB coordinate axes. Once the user decides to return the model to its initial position, they press a key and Aperio Freeform determines if the model is close enough. If so, the model is “snapped” to its exact initial position. The error distance (translational and angular) was set to be quite “loose”; if the user places the model relatively close, the model will snap back into place. We also provided a “restore” button to return a selected model to its initial position regardless of its current position.

4.1.1 Process

The user study consisted of three trials in total: one practice trial (**Figure 80**) to familiarize users with the controls and two real trials. In each trial participants were asked to perform a scene matching task. Participants were shown several scene images. The images depicted a subsystem of the anatomy data set in various “stages” of model rearrangement. Participants were then asked to use Aperio to match each stage of the model rearrangements. The matching task also included restoring the position and orientation of models to their original state. Participants were informed that they were not being timed and their performance was not being measured. To begin the study, each participant was randomly shown the functionality and interface of either Aperio Tools or Aperio Freeform. They were then allowed to play with the system using a demonstration dataset (a heart dataset was used for the practice round). To ensure fairness, the dataset used in the practice trial differed from the ones in the real trials so that users are not simply repeating identical experiments.

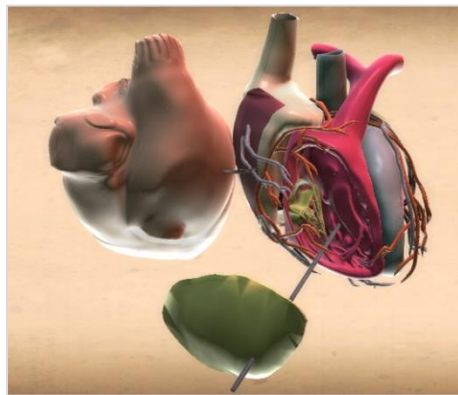


Figure 80: User study’s practice round dataset (heart).

An interface reference guide printed on a sheet of paper was also provided that listed the mouse and key controls and visually depicted the result of using the controls. For the interested reader, Aperio’s Quick Reference (Controls) is available in Appendix A of this thesis.

Once users were familiar with the interface, they were asked to perform the practice trial using demonstration dataset scene images. They then repeated the matching task in two more trials (**Figure 81**), each using different anatomical model subsystems (a brain dataset and a skeletal/intestinal system).

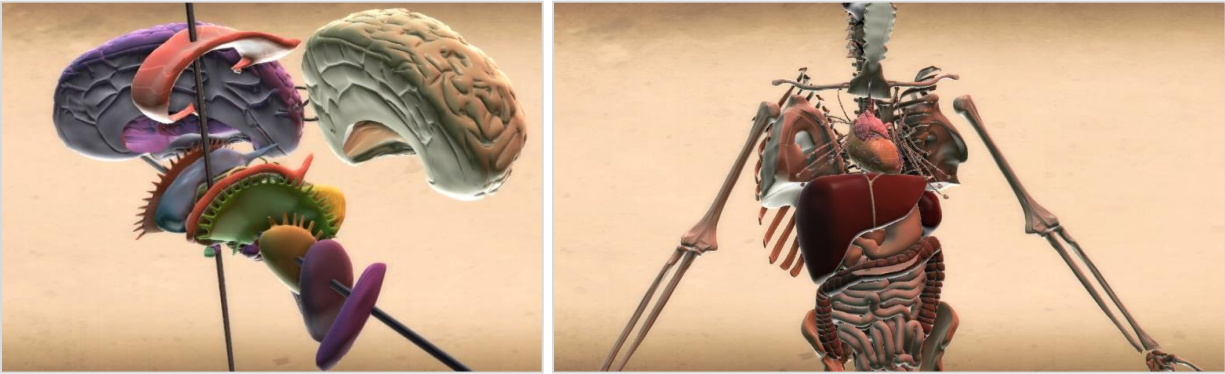


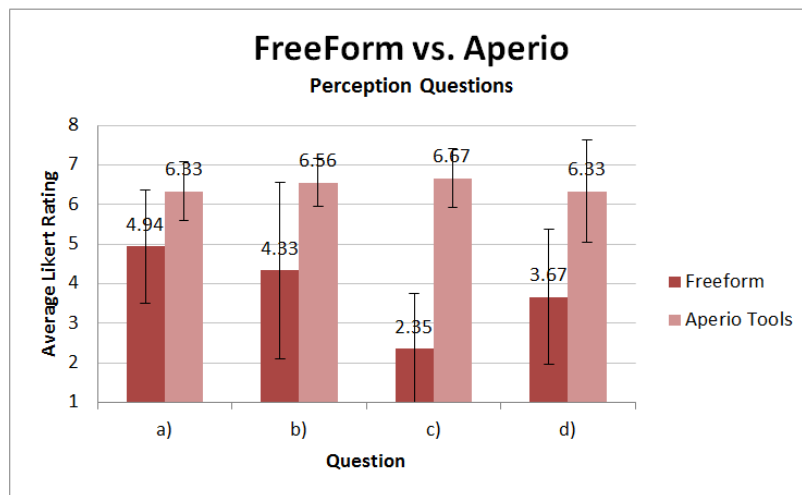
Figure 81: User study's two additional rounds (brain and skeletal/intestinal datasets).

The participants then repeated the study (demonstration of Aperio interface, playing with the system, practice trial and real trials) with the other Aperio version. Finally, once the participants had finished the scene matching tasks, they were asked to play with the cutter tool and ribbon cutter on one of the trial data sets.

After the trials for each version of Aperio, the participants filled out a questionnaire and indicated their level of agreement or disagreement with each statement using a 7-point Likert scale. The survey included user interface questions, (how easy it was to learn and control “Aperio FreeForm” vs “Aperio Tools”) and perception questions (how visually effective both versions were for understanding relationships between parts and for restoring/recreating views of the data). Users were also asked which tool/interface (rod, ring, or Aperio Freeform) they preferred. The survey questions are available in Appendix B of this thesis.

4.1.2 Results

Results of the user study suggest that generally users preferred Aperio Tools over Aperio FreeForm for perception tasks that involved restoring and recreating different views of the dataset. Of the eighteen participants, 14 found the Aperio Tools interface intuitive, 4 did not. For tool preference, 9 users preferred the rod tool, 8 users preferred the ring tool and only 1 user preferred the Aperio Freeform interface. With respect to the cutter tool versus the ribbon cutter, 15 users thought the ribbon cuts were an effective alternative to a full cutaway (for preserving the contours of the cut-away object), 2 did not think it was effective and 1 chose “OK”. **Figure 82** shows that the average Likert scale rating for questions (a) and (b) (how effective the technique is for performing actions, and maintaining an understanding of what transformations have been done to object models, respectively) in Aperio Freeform is 4.94 and 4.33 compared to Aperio Tools (6.33 and 6.56).



Freeform/Aperio Tools help to...

- a) perform actions (move and orient objects)
- b) maintain understanding of what have been done to objects
- c) restore original positioning and orientation of objects
- d) understand layering and spatial relationships between objects

Figure 82: User study’s perception questions (FreeForm vs Tools)

There was a large difference between Aperio Freeform versus Aperio Tools in response to question (c), “restoring original position and orientation” (2.35 in Aperio FreeForm and 6.67 in Aperio Tools). However, admittedly the model restoration control of Aperio Freeform could have been made more automatic, but the user study suggests visual cues in the form of a ring and rod helped users understand the path traversed by the meshes, similarly to creating a “breadcrumb” trail. There is also a significant difference in response to “understanding layering and spatial relationships between objects”. The average score for Aperio Freeform is 3.67 while Aperio Tools average score was 6.33. We also attempted to gain some insight into the simplicity and effectiveness of Aperio’s user interface (**Figure 83**).

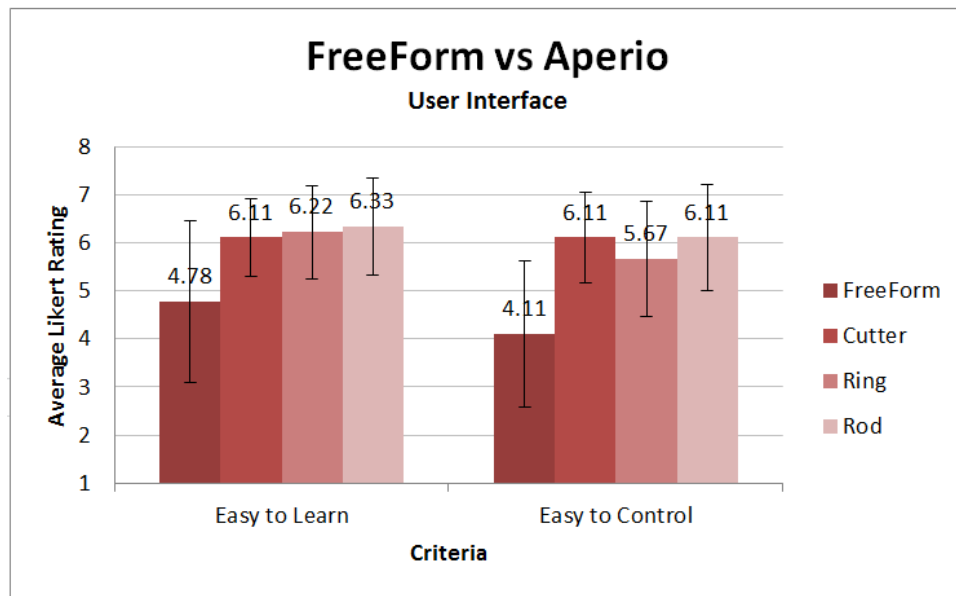


Figure 83: User study’s user interface questions (FreeForm vs Tools)

Overall, users found Aperio Freeform harder to learn and to control than Aperio Tools, although this may be in part due to the short time participants had to learn the various key combinations. Aperio Tools relies on similar keys for tool positioning but also uses GUI sliders to perform

model rearrangements. All three cutter, ring and rod tools seem to be easy to learn as they all had an average likert rating above 6/7, whereas the rod and cutter seem to be easier to control than the ring; this might be due to the ring being primarily suggestive of rotation and secondarily of translation making ring orientation an important part of determining the explosion path; the interface for orienting the ring can be simplified in future work by improving the OBB-positioning mode. The ring still seems to fare reasonably well compared to other tools considering its average likert rating for ease of control is approximately 5.67/7. Overall, the results seem to indicate that the cutter, ring, and rod tools are all relatively easy to learn and control and that they seem to be effective for iteratively creating and restoring immediate views of multi-part mesh data.

5 Conclusion and Future work

5.1 Conclusion

In this thesis, we presented a novel interaction model for managing 3D scene occlusion and for visualizing and understanding spatial relationships among parts of a multi-part mesh-based system. We introduced a mechanical tool analogy through the creation of 4 distinct tools: a (cookie) cutter, ring, rod and knife tool. Each tool uses a single underlying superquadric formulation to represent its shape making them simple to work with since they have few controlling parameters.

The ring and rod tools are used for translation and rotation of meshes; the ring tool allows meshes to slide along its circular path similarly to beads on a wire and primarily suggests rotation due to its circular shape and secondarily of translation. On the other hand, the rod tool allows meshes to slide along its linear path primarily suggesting translation and secondarily of rotation. Meshes can also be spread apart along the ring or rod to create exploded views of mesh parts; they can also be spun around the rod's axis or fanned similarly to an arraying of cards.

The ring and rod tools are presented as an alternative interaction technique for creating exploded views of mesh data along a constrained path, (rather than freely exploding meshes in every possible direction like traditional radial explosions). Explosions can also be reversed (restored) by simply moving the slider (along with the connected meshes) back to their initial rest positions.

In addition to exploding meshes along a constrained path, the ring tool also introduced a way to hinge different mesh parts open simply by planting a smaller or narrower-sized ring on the mesh

to represent a pivot point of rotation on the hinge; this is a corollary of the ring’s rotational nature and allows meshes to be rotated apart similarly to pages being turned on a notebook ring.

The cutter tool introduced gives users the freedom to interactively explore the model by cutting into meshes in real-time, previewing cuts as they slide the cutter along the surface before deciding where to perform the actual cut. The real-time, GPU-based previewer discards fragments in the fragment shader wherever the superquadric tool is positioned and the capping algorithm fills up holes underneath the surface creating an illusion that the meshes are solid. We opted to create a custom capping algorithm rather than use an image-based CSG library (like OpenCSG[61]) since the implicit equation of the superquadric tools was available making capping a simpler problem. For future work, using an image-based CSG library for capping might generate more accurate output. To perform the actual cut, and generate new geometry and connectivity information for newly cut parts, the Carve CSG [58] library was used, which is a fast and reliable CPU-based CSG library used in popular modeling packages such as Autodesk Maya and Blender. We also introduced a Ribbon Cut interface to the cutter; instead of cutting away the mesh entirely, we only remove part of the occluding geometry and retain polygonal strips called “ribbons” or solid thick “slices” to provide an outline of the occluding surface while still being able to reveal the interior. The ribbon cuts were GPU-based and could be modified and updated in real-time as the cutter slides along the surface of the mesh providing interactive exploration.

Finally, the knife tool was introduced as a way to divide meshes into separate parts by simply drawing an incision path along the mesh; after the user creates the incision, a “spreader” appears within the incision cut. The spreader can then be used to interactively widen the incision cut to

any desired degree. The resulting two pieces created from the knife tool can be treated in exactly the same manner as any other object model; this means that newly generated pieces can be exploded or hinged apart using the ring or rod. Each piece could be further divided with knife cuts or instantly mended by using the restore button on the control panel. The user study performed comparing these tools to freeform actions suggests that the tools are relatively simple to use and are effective visual cues for iteratively creating and restoring multiple views of mesh data.

User interaction models are an integral part of effective interactive data visual exploration. If the interaction model is too complex, for example one based on a complicated widget with many handles and controls, the user may spend too much time applying the technique and the exploration work-flow is affected. On the other hand, if the visualization technique is too slow, then an intuitive interaction model is wasted. In our experience with Aperio, fast application of multiple, simple tools that can be dynamically adjusted and that can be combined with other tools may be a more effective strategy for maximizing exploration work-flow. On the other hand, if the tools are too simple then tool proliferation may result, complicating the interface and reducing interaction consistency. Furthermore, while simple tools and restrictive constraints are arguably necessary for the average user, they may prevent expert users from creating more complex views such as freeform shaped cutaway regions or custom curving explosion paths. It is therefore desirable to allow constraints to be “loosened” by expert users.

5.2 Future work

One of the ways we can improve Aperio for expert users is to allow them the capability of creating curving paths. For example, superquadrics can be deformed using well known global deformation functions, such as bending, tapering and twisting, expanding their range of paths and cutaway shapes. Tapering has already been added to Aperio, specifically for the spreaders inside an incision of a knife cut; the amount of tapering on the outer edges of the spreaders can be changed so that users can visibly see the spreaders without having to rotate the scene to another angle, reducing clutching.

In addition, multiple superquadrics can be blended to form complex shapes while still providing a well-defined inside-outside function [70]; this can be implemented using meta-balls and a geometry shader. Multiple tools can then be welded together creating more complex shaped tools or even new additional tools. It is also possible to use a supertoroid central axis as the control poly-line of a spline curve to create more flexible curving paths as well. Currently, meshes are geometrically transformed along the path to simulate rigidly hinging meshes apart but for future work we can add deformable peels to the meshes simulating soft-body dynamics and creating more flexible views into the data. To improve the usability of Aperio, we can add a more flexible surface snapping mode; currently users can go into an OBB-mode where the ring and rod align and snap to the contours of the mesh's Oriented Bounding Box (OBB) making it easier to position the tool but this doesn't allow much control over its pose since it relies only on an OBB which might not accurately represent the mesh's local orientation. Another feature that can be added is allowing the path of the ring or rod to be interactively changed while meshes slide along the tool without resetting them to their initial positions; this would allow users to interactively

adjust the path anytime without losing context of where the mesh was along the path. Collision detection can also be added to the system to preserve layering relationships between parts. In addition, to visually inspect the meshes while they are sliding along the path, we can attach a ball and socket joint to the meshes allowing users to pick up any mesh along the path and simply inspect it by rotating the ball attached to the mesh around its socket; a 3D haptic device might be more suitable for this interaction model.

In terms of limitations, meshes must be manifold meaning there are no stray edges and no boundary edges (meshes must be closed and water-tight) with no self-intersecting faces. Although this thesis demonstrates Aperio on anatomy data, in theory, since we have not made any particular assumptions on the data's geometry, tools in Aperio could essentially be used on any other datasets as well; this includes industrial engines of automobiles, mechanical assemblies, architectural/urban datasets, airliners, etc.

On the other hand, we have made some assumptions using a mechanical tool analogy, which brings up the interesting question of whether it makes sense to use Aperio for exploring other data. Would it make sense to the user if the datasets were a mechanical assembly? The tools themselves would camouflage with the data being explored due to their metallic properties even though functionality of the tools (cutter, knife, ring, rod) would still be applicable for translation and rotation; this might not make much sense to the user since the analogy itself makes an assumption on the dataset.

For future work, perhaps we can extend the metaphor to include other tools such as a plier that would modify and deform tools themselves or a hammer to create explosions. We can also

modify the material properties of the tools so that they are applicable to a wider range of datasets.

At its core, Aperio implements a systematic application of constraints for translating/rotating mesh parts around the scene; this approach sacrifices direct model manipulation (implemented in Aperio FreeForm) for organizational and perceptual advantages provided by visual cues; an example of this is the use of a ring and rod for creating constrained paths. These constraints support exploration of spatial relationships between mesh parts and allow “iterative” manipulations meaning we can quickly regenerate various views by simply moving meshes along a pre-set path while maintaining coherency of the mesh system. Achieving this with freeform with individual manipulations (while fast), would be more problematic since we do not have a “breadcrumb” or trail of where meshes have been translated/rotated making it difficult to restore views. On the other hand, mechanical tools visually depict the exact path or history of the meshes’ traversal. Users can then visually understand how to put things back together.











We hope that by applying these tools using path and shape constraints, combined with a fluid and dynamic interaction model where users can quickly and iteratively generate views by picking up and planting (repositioning) tools, Aperio would be an effective exploration system for gaining insight into 3D mesh-based systems and for occlusion management.

Appendices

Appendix A: Aperio Quick Reference (Controls)

Tips: Select objects first BEFORE planting Rod or Ring.
Left-click Drag to Rotate Scene, Mouse Wheel to zoom scene

Aperio Controls – Quick Reference

FreeForm Version			Ring, Rod, Cutter Version		
	Select	Double-Click	Select, Unselect and Unselect ALL (Same as FreeForm)		
	Select Multiple	Shift + Double-Click		Create New Tool	Click Tool's Icon (only when NOT holding a tool)
	Unselect	Right-click		Spin Held Tool	Alt + Mouse
	Unselect ALL	Double Right-click		Tilt Held Tool	Alt + Mouse Wheel
Toggle Transform Mode		Spacebar		Plant Held Tool	Enter (NO longer holding Tool)
	Spin (z-axis)	ALT + Mouse	Delete Held Tool Del		
	Turn (y-axis)	ALT + Shift + Mouse	Resize Tool Ctrl + Mouse Wheel		
	Tilt (x-axis)	ALT + Mouse Wheel		Deepen Tool (Rod, Cutter)	Ctrl + Shift + Mouse Wheel
			Shift Held Tool Shift + Mouse Wheel (in/out of scene)		

Appendix B: User Study Questions

Background Information and Questions

Study: 3D Visualization and Human-Computer Interaction Methods

Institution: Ryerson University

Department: Computer Science

Name: _____

Age: _____

Gender: _____

Experience (hours per week):

Computer Mouse Usage: _____

Video Games: _____

3D Modelling Software (Blender, Maya, etc.): _____

Unconstrained Controls (FreeForm)

The **FreeForm** technique is easy to learn

Strongly Agree

☐☐☐☐☐

Strongly Disagree

☐☐

The **FreeForm** technique is easy to control

Strongly Agree

☐☐☐☐☐

Strongly Disagree

☐☐

1. The **FreeForm** technique helps to...

a) perform actions (moving and orienting objects)

Strongly Agree

☐☐☐☐☐

Strongly Disagree

☐☐

b) maintain understanding of what have been done to objects

Strongly Agree

☐☐☐☐☐

Strongly Disagree

☐☐

c) restore original positioning and orientation of objects

Strongly Agree

☐☐☐☐☐

Strongly Disagree

☐☐

d) understand layering and spatial relationships between objects

Strongly Agree

☐☐☐☐☐

Strongly Disagree

☐☐

Cutter, Ring and Rod Controls

The **Cutter** technique is easy to learn

Strongly Agree

☐ ☐ ☐ ☐ ☐

Strongly Disagree

☐ ☐

The **Cutter** technique is easy to control

Strongly Agree

☐ ☐ ☐ ☐ ☐

Strongly Disagree

☐ ☐

The **Ring** technique is easy to learn

Strongly Agree

☐ ☐ ☐ ☐ ☐

Strongly Disagree

☐ ☐

The **Ring** technique is easy to control

Strongly Agree

☐ ☐ ☐ ☐ ☐

Strongly Disagree

☐ ☐

The **Rod** technique is easy to learn

Strongly Agree

☐ ☐ ☐ ☐ ☐

Strongly Disagree

☐ ☐

The **Rod** technique is easy to control

Strongly Agree

☐ ☐ ☐ ☐ ☐

Strongly Disagree

☐ ☐

Overall: Did you find the 3D Interaction and Controls Intuitive?

☐ Yes ☐ No

2. Metal Tools (**Ring, Rod, Cutter**) and their visual cues help to...

a) perform actions (move and orient objects)

Strongly Agree Strongly Disagree
☐ ☐ ☐ ☐ ☐ ☐ ☐

b) maintain understanding of what have been done to objects

Strongly Agree Strongly Disagree
☐ ☐ ☐ ☐ ☐ ☐ ☐

c) restore original positioning and orientation of objects

Strongly Agree Strongly Disagree
☐ ☐ ☐ ☐ ☐ ☐ ☐

d) understand layering and spatial relationships between objects

Strongly Agree Strongly Disagree
☐ ☐ ☐ ☐ ☐ ☐ ☐

Which positioning/orienting technique did you like the most?
(Why? Answer in Comments section below)

☐ Ring ☐ Rod ☐ FreeForm

Cut-aways remove pieces from view to reveal objects behind them. Are ribbon cuts an effective alternative for preserving the outline of the removed piece?

☐ Yes ☐ No

Additional Comments/Suggestions

REFERENCES

- [1] N. Elmqvist and P. Tsigas, “A Taxonomy of 3D Occlusion Management for Visualization,” *Visualization and Computer Graphics, IEEE Transactions on*, vol. 14, no. 5, pp. 1095–1109, 2008.
- [2] W. Li, M. Agrawala, B. Curless, and D. Salesin, “Automated generation of interactive 3D exploded view diagrams,” in *ACM Transactions on Graphics (TOG)*, 2008, vol. 27, no. 3, p. 101.
- [3] “3D Custom Woody Beach Cruiser.” [Online]. Available: http://www.acme-3d.com/3d/3d_woody/3d_woody_phantom.jpg.
- [4] “Image Foundry - Projects.” [Online]. Available: <http://www.imagefoundry.co.uk/images/projects/cutaway.jpg>.
- [5] “Tegra K1 Mobile Processor from NVIDIA Tegra.” [Online]. Available: <http://www.nvidia.com/object/tegra-k1-processor.html>.
- [6] E. Limer, “Holy Crap, Nvidia’s New Tegra K1 Has 192 Cores?!,” *Gizmodo*. 2014.
- [7] M. J. McGuffin, L. Tancu, and R. Balakrishnan, “Using Deformations for Browsing Volumetric Data,” in *Proceedings of the 14th IEEE Visualization 2003 (VIS’03)*, 2003, p. 53–.
- [8] C. D. Correa, *Illustrative Deformation of volumetric objects and other graphical models*. ProQuest, 2007.
- [9] J. Sinnott and T. Howard, “SQUIDS: Interactive Deformation of Superquadrics for Model Matching in Virtual Environments,” in *Proceedings of the Eurographics UK Conference. Eurographics UK Chapter, Abingdon*, 2000, pp. 73–80.
- [10] I. Quilez, “Modeling with Distance Functions.” [Online]. Available: <http://www.iquilezles.org/www/articles/distfunctions/distfunctions.htm>.
- [11] T. McInerney and P. Crawford, “RibbonView: interactive context-preserving cutaways of anatomical surface meshes,” in *Proceedings of the 6th international conference on Advances in visual computing - Volume Part II*, 2010, pp. 533–544.
- [12] A. Birkeland and I. Viola, “View-dependent peel-away visualization for volumetric data,” in *Proceedings of the 2009 Spring Conference on Computer Graphics*, 2009, pp. 121–128.
- [13] C. Pindat, E. Pietriga, O. Chapuis, and C. Puech, “Drilling into complex 3D models with gimlenses,” in *Proceedings of the 19th ACM Symposium on Virtual Reality Software and Technology*, 2013, pp. 223–230.
- [14] S. Bruckner and M. E. Groller, “Exploded views for volume data,” *Visualization and Computer Graphics, IEEE Transactions on*, vol. 12, no. 5, pp. 1077–1084, 2006.
- [15] W. Li, L. Ritter, M. Agrawala, B. Curless, and D. Salesin, “Interactive cutaway illustrations of complex 3D models,” in *ACM SIGGRAPH 2007 papers*, 2007.
- [16] M. Trapp and J. Döllner, “2.5D Clip-Surfaces for Technical Visualization,” *Journal of WSCG*, vol. 21, no. 1, pp. 89–96, 2013.
- [17] M. Burns and A. Finkelstein, “Adaptive Cutaways for Comprehensible Rendering of Polygonal Scenes,” *ACM Transactions on Graphics (Proc. SIGGRAPH ASIA)*, vol. 27, no. 5, p. 124:1—124:9, Dec. 2008.

- [18] M. Tatzgern, D. Kalkofen, and D. Schmalstieg, "Compact Explosion Diagrams," in *Proceedings of the 8th International Symposium on Non-Photorealistic Animation and Rendering*, 2010, pp. 17–26.
- [19] L. Cmolik, "Relational Transparency Model for Interactive Technical Illustration," in *Smart Graphics*, 2008, pp. 263–270.
- [20] M. Otsuki, A. Kimura, F. Shibata, and H. Tamura, "Stretch 'n' cut: Method for observing and ungrouping complex virtual objects in 3D space using elastic band metaphor," in *3D User Interfaces (3DUI), 2014 IEEE Symposium on*, 2014, pp. 27–30.
- [21] S. Sigg, R. Fuchs, R. Carnecky, and R. Peikert, "Intelligent cutaway illustrations," in *Pacific Visualization Symposium (PacificVis), 2012 IEEE*, 2012, pp. 185–192.
- [22] N. J. Mitra, Y.-L. Yang, D.-M. Yan, W. Li, and M. Agrawala, "Illustrating how mechanical assemblies work," *ACM Transactions on Graphics (TOG)*, vol. 29, no. 4, p. 58, 2010.
- [23] C. Tominski, S. Gladisch, U. Kister, R. Dachsel, and H. Schumann, "A survey on interactive lenses in visualization," 2014.
- [24] S. Spencer, *ZBrush Character Creation: Advanced Digital Sculpting*. John Wiley & Sons, 2011.
- [25] J. Bertin, W. J. Berg, and P. Scott, *Graphics and Graphic Information Processing*. De Gruyter, 1981.
- [26] K. Power, "3D Object Representations." [Online]. Available: http://glasnost.itcarlow.ie/powerk/GeneralGraphicsNotes/meshes/polygon_meshes.html.
- [27] L. Y. Deouell, D. Deutsch, D. Scabini, N. Soroker, and R. T. Knight, "No disillusion in auditory extinction: perceiving a melody comprised of unperceived notes," *Frontiers in Human Neuroscience*, vol. 2, no. 15, 2008.
- [28] "Elsevier Inc.: Introduction to CT Physics." 2004.
- [29] T. Weyrich, M. Pauly, R. Keiser, S. Heinzle, S. Scandella, and M. Gross, "Post-processing of Scanned 3D Surface Data," in *Proceedings of the First Eurographics Conference on Point-Based Graphics*, 2004, pp. 85–94.
- [30] A. Cockburn, A. Karlson, and B. B. Bederson, "A review of overview+detail, zooming, and focus+context interfaces," *ACM Comput. Surv.*, vol. 41, no. 1, pp. 2:1–2:31, Jan. 2009.
- [31] A. Yakobovich, "ViC: virtual cadaver - a prototype extendible system for building photorealistic interactive visualizations of human anatomy using game development technology," *Theses and dissertations*, Jan. 2011.
- [32] R. Kelc, "Zygote Body: A New Interactive 3-Dimensional Didactical Tool For Teaching Anatomy," 2012.
- [33] I. Viola and E. Gröller, "Smart visibility in visualization," in *Computational Aesthetics*, 2005, pp. 209–216.
- [34] D. A. Bowman, *3D user interfaces: theory and practice*. Boston: Addison-Wesley, 2005.
- [35] L. Bavoil and K. Myers, "Order independent transparency with dual depth peeling," *NVIDIA OpenGL SDK*, 2008.
- [36] J. C. Yang, J. Hensley, H. Grün, and N. Thibieroz, "Real-time Concurrent Linked List Construction on the GPU," in *Proceedings of the 21st Eurographics Conference on Rendering*, 2010, pp. 1297–1304.
- [37] M. McGuire and L. Bavoil, "Weighted Blended Order-Independent Transparency," *Journal of Computer Graphics Techniques (JCGT)*, vol. 2, no. 2, pp. 122–141, Dec. 2013.

- [38] MJP, “Weighted Blended Order-Independent Transparency,” *The Danger Zone*. .
- [39] S. Bruckner, S. Grimm, A. Kanitsar, and M. E. Gröller, “Illustrative context-preserving volume rendering,” in *EuroVis*, 2005, pp. 69–76.
- [40] H. Sonnet, S. Carpendale, and T. Strothotte, “Integrating expanding annotations with a 3D explosion probe,” in *Proceedings of the working conference on Advanced visual interfaces*, 2004, pp. 63–70.
- [41] R. Gasteiger, M. Neugebauer, O. Beuing, and B. Preim, “The FLOWLENS: A focus-and-context visualization approach for exploration of blood flow in cerebral aneurysms,” *Visualization and Computer Graphics, IEEE Transactions on*, vol. 17, no. 12, pp. 2183–2192, 2011.
- [42] D. A. Norman, *The design of everyday things*. Basic books, 2002.
- [43] “QT Controls.” [Online]. Available: http://qt-project.org/images/qquick_controls1.png.
- [44] L. Avila, “Volume Rendering Improvements in VTK,” *The Source*. .
- [45] K. Martin and B. Hoffman, “Mastering CMake: A Cross-Platform Build System,” 2013.
- [46] J. T. Bell, “Visualization Toolkit (VTK) Tutorial.” [Online]. Available: http://www.cs.uic.edu/~jbell/CS526/Tutorial/VTK_Pipeline.jpg.
- [47] S. Green, “The OpenGL framebuffer object extension,” in *Game Developers Conference*, 2005, vol. 2005.
- [48] N. Meyer, “Screen Space Ambient Occlusion (SSAO),” *Nutty Software*. [Online]. Available: http://www.nutty.ca/?page_id=352&link=ssao.
- [49] L. Bavoil and M. Sainz, “Screen space ambient occlusion,” *NVIDIA developer information: <http://developers.nvidia.com>*, vol. 6, 2008.
- [50] T. Lottes, “FXAA (Whitepaper),” Tech. rep., NVIDIA, 2011. 2, 2009.
- [51] A. Jaklic, A. Leonardis, and F. Solina, “Superquadrics and Their Geometric Properties,” in *Segmentation and Recovery of Superquadrics*, vol. 20, Springer Netherlands, 2000, pp. 13–39.
- [52] A. H. Barr, “Superquadrics and Angle-Preserving Transformations,” *Computer Graphics and Applications, IEEE*, vol. 1, no. 1, pp. 11–23, Jan. 1981.
- [53] T.-C. Publishing, “Modelling Techniques (Finite Element Method) Part 2,” *what-when-how*. 2012.
- [54] Leadwerks Corporation, “What is Constructive Solid Geometry?” [Online]. Available: <http://www.leadwerks.com/files/csg.pdf>, 2006.
- [55] G. Collaboration, “Geant4 User’s Guide for Application Developers,” *Accessible from the GEANT4 web page [1] Version Geant4*, vol. 9, 2012.
- [56] J. Gregson, “pyPolyCSG, A Polyhedral CSG library for Python.” 2012.
- [57] S. Popinet, “GNU Triangulated Surface Library.” 2006.
- [58] S. Tobias, “Carve CSG.” 2014.
- [59] J. Slick, “Preparing a Model for 3D Printing,” *About Technology*, 2014.
- [60] R. Schmidt and K. Singh, “Meshmixer: An Interface for Rapid Mesh Composition,” in *ACM SIGGRAPH 2010 Talks*, 2010, pp. 6:1–6:1.
- [61] F. Kirsch and J. Döllner, “OpenCSG: A Library for Image-based CSG Rendering,” in *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, 2005, pp. 49–49.

- [62] J. Goldfeather, S. Monar, G. Turk, and H. Fuchs, "Near Real-Time CSG Rendering Using Tree Normalization and Geometric Pruning," *IEEE Comput. Graph. Appl.*, vol. 9, no. 3, pp. 20–28, May 1989.
- [63] N. Stewart, G. Leach, and J. Sabu, "Linear-time CSG rendering of intersected convex objects," 2002.
- [64] G. Casiez, D. Vogel, Q. Pan, and C. Chaillou, "RubberEdge: reducing clutching by combining position and rate control with elastic feedback," in *Proceedings of the 20th annual ACM symposium on User interface software and technology*, 2007, pp. 129–138.
- [65] D. M. Advisor, M. C. Lin, F. P. Brooks, S. Gottschalk, and S. Gottschalk, "Collision Queries using Oriented Bounding Boxes," 2000.
- [66] H. A. Sulaiman, M. A. Othman, M. M. Ismail, M. A. Meor Said, A. Ramlee, M. H. Misran, A. Bade, and M. H. Abdullah, "Distance computation using axis aligned bounding box (AABB) parallel distribution of dynamic origin point," in *Emerging Research Areas and 2013 International Conference on Microelectronics, Communications and Renewable Energy (AICERA/ICMiCR), 2013 Annual International Conference on*, 2013, pp. 1–6.
- [67] J. S. Elias, "Creating a Spherical Reflection/Environment Mapping shader," *ClickToRelease*. 2014.
- [68] "Matcap Shader," *MODO 801 Inline Help System*. 2013.
- [69] "Surface Shader Examples," *Unity Manual*. 2014.
- [70] L. Faynshteyn and T. McInerney, "Context-Preserving Volumetric Data Set Exploration Using a 3D Painting Metaphor," in *Advances in Visual Computing*, vol. 7431, G. Bebis, R. Boyle, B. Parvin, D. Koracin, C. Fowlkes, S. Wang, M.-H. Choi, S. Mantler, J. Schulze, D. Acevedo, K. Mueller, and M. Papka, Eds. Springer Berlin Heidelberg, 2012, pp. 336–347.