Chapter 7

Incision: A hands-on system for manipulation of volumetric data sets

7.1 Introduction

Volumetric data present unique challenges in terms of visualization and interaction. Correa in his paper, “Visualizing what lies inside”, illustrates the importance of visualization for internal features [Cor09]. As opposed to tackling the challenges of internal visualization simply through visual representations, a system was derived that supports rapid pressure sensitive multipoint interaction. Data is manipulated and visualized directly on the graphics processing unit (GPU), enabling highly interactive rates that allow users to manipulate the model on a per voxel basis, akin to work seen in the field of volume sculpting.

Early digital volumetric sculpting was performed by Gaylen and Hughes in 1991 [GH91] who used a custom made 3D positioning system to cut, add, and smooth a virtual block of clay. After each manipulation, an iso-surfacing pass was applied to create a polygonal mesh, used for subsequent visualization.

Wang, et al., used ray casting to create cuts through voxel grids based on standard mouse and keyboard input [WK95]. Wang visualized these results using a
progressive ray marching approach based on a single isovalue. For refined imaging, the system used out-of-core ray tracing. In 2007, Perng used a modified marching cubes approach for volumetric carving with GPU support [PWFO01].

Several researchers have looked into haptic input devices as better interface tools for the analysis of volumetric data sets, and much of the work on volume sculpting relies on six degree of freedom (6-DOF) input devices [CS02] [CHS04]. Chen et al. used a 6-DOF input device to mimic the metaphors of melting, burning, painting, construction, stamping and peeling [CS02].

Figure 7.1: Examples of various reference data sets used for system tests. Courtesy of http://www.volvis.org/ and http://www.vis.uni-stuttgart.de/~engel/pre-integrated/data.html

One of the major challenges with volumetric sculpting is the need to visualize results at interactive rates. In Chen’s 2002 paper, the significance of this problem is apparent from Table 1 [CS02]. As voxel manipulation regions increase, the system’s interactivity decreases exponentially. While bus and processor speeds have increased significantly since Chen’s paper, they are still a limiting factor when performing large volumetric updates as shown in the results section below.

Bruckner et al. demonstrate a 3D illustration environment in their project VolumeShop [BVG05]. VolumeShop allows users to manipulate selected sub-volumes. These subregions can be easily manipulated, but, as shown in the paper, interaction rates are slowed dramatically as subregion sizes increase. Correa et al. demonstrated an engaging method for applying virtual peelers, retractors, pliers and dilators to volumetric data [CSC06]. While this method allowed for interactive modifications, it required data to be preprocessed. Additionally, multiple modifications caused problems when modification regions overlapped. McGuinness et al.
used deformation strategies on semantic layers which compose a volume data set [MTB03]. While this approach is incredibly powerful, it requires users to segment data into discrete layers such as skin, bone, muscle, etc.

Once data has been loaded into graphics memory, several techniques are commonly applied to extract meaningful information [Elv92] such as constructing tessellated models based on a single isovalue, using for example the marching cubes [LC87] algorithm. The advantage of this or similar techniques is that a single processing pass can be applied to extract the needed 3D information and standard 3D rendering and acceleration techniques can be subsequently used for the creation of interactive visuals. Johansson showed how this process could be moved to the GPU to further increase performance [JC06]. For the purposes of his paper, it was important to show multiple depth values concurrently and a GPU-accelerated ray marching visualization approach was selected.

Akeley proposed the possibility of using dedicated 3D hardware to accelerate volume rendering [Ake93] as part of the RealityEngine graphics pipeline and in 1994, Cabral et al. used an accelerated volumetric reconstruction via graphics hardware [CCF94]. Subsequently, Westermann [WE98] efficiently used commodity graphics hardware in volume rendering applications and Kniss et al., used graphics hardware to apply volumetric lighting and shadowing to volume models [KKH01].

Along with visualization, intuitive interaction is critically important. Yet, the aforementioned sculpting techniques only work on a single interaction point. Additionally, 6-DOF devices are generally decoupled from the display, forcing users to virtually feel the area in which they are manipulating.

Multi-touch surfaces support a natural coupling of visualization and user interface, allowing users to intuitively understand where and how they are changing the volume. Multi-touch surfaces also allow for multiple points to be inspected simultaneously.

The Virtual Autopsy Table [Ryd] demonstrated the ability to use multi-touch for visualizing medical data. This work utilized well known multi-touch pan and zoom metaphors to move volume data around. The manipulation of the volumetric data consisted of standard cutting planes and volume rendering.
As shown in Chapter 6 and Bonanni et al. [BXH+09], multi-touch devices can be used as an intuitive way to explore multi-layered data. Transitioning these types of interface modalities from layered imagery to volumetric data requires a new and novel approach.

This chapter presents Incision, a method for interactive, hands-on analysis of volumetric data, using intuitive metaphors. Hands-on volume analysis is enabled by a custom-built multi-touch system with a touch vocabulary and grammar, allowing users to freely transform the target volume, drill, scrape, and restore volumetric data types on localized regions and provides users with the ability to annotate regions of interest. Position, gesture and pressure information are used for voxel density and depth specific operations. Three-dimensional framebuffer-objects provide the support structure needed for GPU-centric processing and allow data to be modified at fixed cost. Realistic visualizations are produced through a multi-pass, multi-targeted rendering pipeline, supporting effects such as shading and shadowing, adding visual depth cues and realism. Figure 7.1 shows example data sets being manipulated, while Figure 7.2 illustrates user interaction.

With volume rendering, volume manipulation and multi-touch interaction closely interconnected, each is described to capture the research contribution. First the hardware assisted rendering algorithms are described, equations, associated data structures, equations, and rendering techniques. Since the volume manipulation uses the depth map generated in the volume rendering pass, the methodology for data manipulation is presented next. Finally, the hardware, metaphors, and vocabulary used for multi-touch centric modeling are described.

### 7.1.1 FrameBuffer Objects

Framebuffer objects (FBOs) provide a powerful mechanism for manipulating data on of the GPU. Standard two-dimensional FBOs enable the graphics card to render viewpoints to a texture rather than to the screen. Three-dimensional framebuffer objects enable rendering to a volume and work much like a stack of two-dimensional framebuffer objects. The framebuffer object attaches to a z-slice, and any of the x and y pixels of this attached slice can be modified on the GPU.
To change pixels with different z values, their z-slice must be attached accordingly.

**Figure 7.2:** A user exploring volumetric data using the polygon mode.

For the volume to be modified with feedback, two three-dimensional FBOs are needed, one which functions as a read buffer and the second which functions as a write buffer. The buffer to be read from is named the front buffer and the buffer to be written to is named the back buffer, as seen in the OpenGL pipeline. Once the render-to-texture update finishes, the pointers to the front and back texture buffers are swapped. This step is necessary to avoid unwanted feedback loops between write and read operations. For Incision, five additional two-dimensional FBOs are used to store raster information for the rendered viewpoint. These FBOs serve as buffers for the rendering equation, shadow map, depth map, normal map, and shadow mask, as shown in Figure 7.3(A)-(E).

### 7.2 Volume Rendering

The initial volume rendering pass is similar to the approach presented by Cabral [CCF94]. Data slices are rendered parallel to the viewing direction in back-
to-front order. Each of these data slices contains a transfer function. The volume rendering equation as shown in [EKE01] can be represented as the integral

$$I = \int_0^d \text{color}(x(\lambda)) \exp\left(-\int_0^\lambda \text{extinction}(x(\lambda'))d\lambda'\right)d\lambda$$  \hspace{1cm} (7.1)

where the viewing ray $x(\lambda)$ is parameterized by the distance from the viewpoint and that color density and extinction values may be calculated anywhere along this ray. Also, represents the maximum distance from the camera that can be stored.

The approximation for this integral is given by

$$C_i = \alpha_i C_i + (1 - \alpha_i) C_{i+1}$$  \hspace{1cm} (7.2)

This representation can be approximated on the GPU by using a series of rectangles that are drawn in back-to-front order and bound to the volume texture. The number of slices should be equal to the depth of the volume at a minimum, but can also be increased to achieve ray super-sampling. Assuming a z-in orientation, the texture coordinates for each rectangle consist of the bounds of the volume in the $x$ and $y$ direction, and the $z$ value equal to the slice, which is currently being rendered. By binding a fragment program to the render loop, the rendering equation is evaluated as shown in Figure 7.3(A).

It is important to note that the process to render every frame is the exact same regardless of content. This allows the entire drawing process to be handled through static display lists for additional performance.

### 7.2.1 Multiple Rendering Targets

With the volume rendering step being a costly operation, it is preferable to render it only once and produce additional visual effects based on the two-dimensional outputs. Since it is possible to render to multiple outputs in a single fragment program, multiple viewpoints and renderings can be produced in a single pass as shown in Figure 7.3. In the first rendering pass through the volume, three targets are specified to render to, and the volume is rendered in back-to-front order.
to ensure proper blending as shown in Figure 7.3(A). The output of the rendering equation is stored in the first render target. The second render target stores the shadow map and the third render target the depth map (Figure 7.3(B)(C)). The second pass uses the output of the depth buffer to generate the shadow mask and the normal map as shown in Figure 7.3(D)(E). Finally, the rendering output, normal map, and shadow mask are all used to generate the final composite (Figure 7.3(F)).

### 7.2.2 Matrix Manipulation

In order to change the viewpoint from which the volume is being studied, the transformation matrix is multiplied by the texture coordinates in the vertex program before being passed to the fragment program. If the modelview transformation matrix for the users viewpoint is stored in the matrix $M$, and the original unit cube texture location is $T_{IN}$, then the resulting texture lookup for the fragment program will be:

$$T_{Eye} = M^{-1}T_{IN}$$  \hspace{1cm} (7.3)

similarly, the texture lookups for the light can be computed by

$$T_{Light} = M^{-1}(L^{-1}T_{IN})$$  \hspace{1cm} (7.4)

where $L$ is the transformation matrix for the light. The light is rotated with the eye to keep it in the same relative position with the object, emulating the effect of moving the volume rather then the viewing direction. If the desired effect is that of moving the viewing direction, the $T_{Light}$ would not need to be multiplied by $M^{-1}$.

### 7.2.3 Lighting and Shading

One common shortcoming of the approach proposed by Cabral et al. [CCF94] is that it does not include shading or lighting. Lum et al. documented the improve-
Figure 7.3: Illustration of the multi-pass rendering pipeline, showing (A) result of the front to back rendering, (B) shadow map, and (C) depth map created during the first rendering pass, the (D) normal map, and (E) shadow mask created during the second rendering pass and (F) the resulting composited image.
ments lighting adds to volume rendering [LM04] by providing helpful topological and depth cues.

For the Phong shading model, the lighting equation can be represented as

\[
C = C_{\text{voxel}} * (k_a + k_d * \text{MAX}(N \ast L, 0)) + k_s * \text{MAX}((N \ast R)^n, 0) \quad (7.5)
\]

where \(C_{\text{voxel}}\) is the incoming color, \(N\) is the normal vector, \(L\) is the normalized light vector, \(R\) is the reflection vector, \(k_d, k_s, k_a\) are the diffuse, specular, and ambient representation respectively, and \(n\) is the specular shininess of the material [LM04].

The difficulty of applying this type of shading is in computing the surface normal, as there is inherently no predefined normal as which comes from triangulated meshes.

### 7.2.4 Normals

In order to apply the Phong shading model, surface normals need to first be determined. While Sobel gradients, as described in [HLSR08] produce high quality results, they require 26 additional volumetric texture lookups and only determine normals over a 3x3x3 voxel grid.

Lum [LM04] and Hadwiger [HLSR08] propose using forward gradients which only check three surrounding pixels as an optimized way of producing these gradient values. As shown in Hadwiger’s paper, this produces faster but lower quality results [HLSR08].

As a compromise of speed and quality, pseudo normals are created based on the camera’s depth map. Vectors can then be created using the texture values for \(x\) and \(y\) using the camera depth map to infer \(z\) values. By creating vectors for the surrounding depth map pixels, the surface normal can be approximated for localized regions as shown in Figure 7.3(D). While this approximation produced smoother looking normals, it also increases the frame rate by 200% compared to using sobel gradients. Pseudo code is shown in the Appendix A.3.
7.2.5 Shadow Maps

Shadows are a powerful cognitive tool for relating depth and space to a user. Shadows also add an extra sense of realism for rendered scenes. In their 1998 paper, Behrens and Ratering found that shadows significantly improved the spatial understanding of volume data if rendered at interactive frame rates [BR98].

Behrens presented an algorithm for storing shadows inside of the volumetric data structure. This approach unfortunately was not fully parallelizable as each subsequently rendered slice needed to be fully completed before the next slice could be rendered. As shown in this paper, this method causes performance to degrade up to 50% [BR98].

Methods such as Deep Shadow Maps can be computed on the GPU, but require much of the information to pre-computed as well as pre-compressed [HKSB06]. Since the volume manipulation for the hands-on volume analysis technique has to be dynamic, this was not an option. Zhang implemented a GPU based approach without pre-computation, but as shown in the paper frame rates declined to 10 fps when dealing with volume data [Zha09].

When generating shadow maps with polygonal rendering, the scene must be rendered twice, first from the point of view of the user and subsequently from the point of view of the light. As shown above, this can be done in parallel using multiple render targets.

The depth map is computed first for the eye using the lookup equation from above. The equation for transforming a vector from the eye’s point of view ($V_{Eye}$) to the light’s viewpoint can be expressed as

$$V_{Light} = M^{-1}(L^{-1}(MV_{Eye}))$$  \hspace{1cm} (7.6)

where $M$ is the eyes transformation matrix and $L$ is the light’s transformation matrix. By comparing the depth value stored in the shadow map to that of the projected value stored in the depth map for the same raster point, the shadow mask can be determined (Figure 7.3(E)). Utilizing the benefits of multiple render targets, producing the shadow map comes at a minimal cost. Pseudo code for the lighting and shading used is shown in the Appendix A.4.
7.2.6 Compositing

The final result is generated by compositing the end products of the previous rendering passes (Figure 7.3(F)). The resulting pixel color is defined as

\[
C_{\text{result}} = \text{ShadowMask}_{xy} \ast \text{ColorMap}_{xy} \ast \\
(k_a + k_d \ast \text{MAX}(\text{NormalMap}_{xy} \ast L, 0)) \\
+k_s \ast \text{MAX}((\text{NormalMap}_{xy} \ast R)^n, 0)
\]

where the \(k_d\), \(k_a\), \(k_s\) and \(n\) are all input parameters, \(L\) and \(R\) are calculated in the shader, and ShadowMask, ColorMap, and the NormalMap are sampled from previous rendered outputs based on the current raster position \(xy\).

7.3 Interaction

As a pre-requisite for an intuitive and interactive environment, a highly responsive system with well known interface widgets such as buttons and sliders was targeted as the baseline.

7.3.1 Widgets

Surface widgets allow users to customize the visualization, mode of operation, and interaction preferences. Since the interface is designed for multi-touch from the bottom up, sliders, buttons, and volume manipulations can all be manipulated simultaneously. For example, this allows users to modify the volume with one hand while spinning the volume with the other. The ability to detect the pressure applied by the user while interacting with that multi-touch surface, supports pressure and pressure range specific operations.

7.3.2 Transformations

At any point in time the user may rotate the model via sliders on the top and bottom of the model’s view area. Additionally, the user can switch into the
transformation mode. In this mode, user’s simply touch the virtual model and use a trackball metaphor to spin it freely via hand movements on the table surface.

![Illustration of the different interrogation modes](image)

**Figure 7.4**: Illustration of the different interrogation modes (A) showing the scraping modality, (B) the drilling modality, (C) the healing modality, (D) the annotation modality.

### 7.3.3 Volume Modification

Multiple modes are used to change the volume, relying on a similar methodology. First, up to sixteen points with radius are packed and uploaded to the GPU. The number sixteen was arbitrarily chosen and the code could easily accommodate more. Currently if more than 16 touches are on the screen, the shader will be run multiple times until all of the touches are accounted for.

To modify the volume, it is first rendered in slices along the z direction. Each of these slices are attached to, and modified directly on the GPU. Each pixel fragment will be transformed from the original unit coordinate system into the eye space coordinate system. The surface nearest to the eye can also be modified using the depth map. As the depth map was already computed for the rendering of the volume, detecting the nearest surface incurs no additional cost.

### 7.3.4 Scraping Mode

In the scraping mode, users can wipe away surfaces mimicking the mode of operation seen in [BXH+09] and Chapter 6. The pressure mapping can be easily controlled via a pressure widget on the side. This produces straightforward means to peel off volume layers such as flesh, muscle and bone for medical data sets as shown in Figure 7.4(A).
Pseudo code is as follows:

\[ T_{EyeSpace} = M \ast T_{IN} \]
\[ T_{EyeSpace_z} = \text{tcxLookup}(depthMap, T_{EyeSpace_{xy}}) \]
\[ \text{if}(\text{abs}(T_{EyeSpace} - T_{IN}) < \text{blobSize}) \land \text{AND}(\text{Pressure} > T_{IN_{data}}) \]
\[ \text{Volume}_{OUT_{mask}} = 0 \]

7.3.5 Drilling Mode

In the drilling mode, each of the user’s touches will cut through the volume based on the user’s viewpoint and the pressure of each touch. This is useful for drilling into the volume in order to see data on the inside as shown in Figure 7.4(B).

Pseudo code is as follows:

\[ T_{EyeSpace} = M \ast T_{IN} \]
\[ \text{if}(\text{Pressure} > T_{EyeSpace_z}) \]
\[ \text{Volume}_{OUT_{mask}} = 0 \]

7.3.6 Healing Mode

![Figure 7.5: Automatic volume healing image sequence. Frame numbers are shown in the lower right-hand corner. For this example, the autohealing was activated at frame 0 and took 20 frames to complete.](image)

Healing mode works to restore previously removed sections. Conceptually the inverse of scraping, voxels which are visibly nearest to the camera are re-added when the isovalue is less than or equal to the pressure. This allows for lesser isovalues to easily be restored as shown in Figure 7.4(C).

Pseudo code is as follows:
\[ T_{\text{EyeSpace}} = M \cdot T_{IN} \]

\[ T_{\text{EyeSpace}_z} = \text{texLookup}(\text{depthMap}, T_{\text{EyeSpace}_{xy}}) \]

if \( \left| T_{\text{EyeSpace}} - T_{IN} \right| < \text{blobSize} \) AND \( \text{Pressure} > T_{IN\text{data}} \) AND \( T_{IN\text{mask}} < 1 \)

\[ \text{VolumeOUT}_{\text{mask}} + = \text{healAmout} \]

### 7.3.7 Annotation Mode

The annotation mode is used to visually annotate the volume itself (Figure 7.4(D)). The voxel closest to the viewing direction along the path of the eye ray has its color channel set to the current annotation look up color, similar to the procedure used in [BVG05]. As this lookup is only 8 bits, no more than 256 colors can be used on the scene simultaneously. In practice, users felt 256 colors were a sufficient for volume annotation. Since these colorizations are volumetric, they can subsequently be manipulated with any of the other analysis metaphors.

Pseudo code is as follows:

\[ T_{\text{EyeSpace}} = M \cdot T_{IN} \]

\[ T_{\text{EyeSpace}_z} = \text{texLookup}(\text{depthMap}, T_{\text{EyeSpace}_{xy}}) \]

if \( \left| T_{\text{EyeSpace}} - T_{IN} \right| < \text{blobSize} \)

\[ \text{VolumeOUT}_{\text{color}} = \text{colorLookupValue} \]

### 7.3.8 Polygon Mode

Quick regional inspection of volumetric data is possible with a polygonal mode. For this mode, all user touches are run through a convex hull algorithm [Gra72] in order to determine the encompassing polygon. The color for each vertex of the encompassing polygon is matched with pressure of the corresponding touch. The encompassing polygon is then rendered into a FBO. This FBO is used as a lookup for volume deformation based on the current viewpoint.

Pseudo code is as follows:

\[ T_{\text{EyeSpace}} = M \cdot T_{IN} \]

\[ p = \text{texLookup}(\text{pressureMap}, T_{\text{EyeSpace}_{xy}}) \]

if \( p > T_{\text{EyeSpace}_z} \)

\[ \text{VolumeOUT}_{\text{mask}} = 0 \]
7.3.9 Automatic Volume Healing

Since the entire volume is being analyzed on the GPU, it is also possible to create “self-healing” technique that can dynamically revert from the current state to the visual representation of the original, unmodified volume. This mimics the effect shown in the Khronos projector [CI05] on a three-dimensional volume as opposed to a two-dimensional clipping plane. This effect would be difficult to implement in realtime via the CPU since the entire volume has to be parsed and large sections need to be changed.

This approach functions as a three-dimensional dilation filter with the addition that dilation can only occur between voxels with similar isovalues, allowing skin to heal surrounding skin while bone would not be able to heal skin. The speed at which this healing takes place is fully adjustable. When set to rapid healing rates, the algorithm acts almost as a virtual “flashlight”, providing updated information for the section that is currently being analyzed. As soon as the user stops interacting with the target region or moves away from it, localized changes are immediately reversed. An example frame series of volume healing is shown in Figure 7.5.

Pseudo code is as follows:

\[
\text{if}(T_{IN}\_mask == 0) \\
\text{commonNeighbor} = 0; \\
\text{for voxels } V \text{ between } (-1,-1,-1) \text{ to } (1,1,1) \\
\text{if}(T_{IN} - V_{xyz}\_data) \text{ is common to } T_{IN} \\
\text{AND } ((T_{IN} - V_{xyz}\_mask) == 1) \\
\text{commonNeighbor} = 1; \\
\text{if}(\text{commonNeighbor}) \\
\text{Volume}_{OUT}\_mask = 1
\]

7.4 Design Decisions and Limitations

The primary design decisions for the algorithms and interfaces at the software and hardware level are based on the desire to create highly-interactive and realistic visuals of volumetric data. This approach is not without limitations though.
The most notable is that it requires a graphics card which supports the glFrame-
bufferTexture3DEXT extension and texture memory which is large enough to hold
the entire volume. In addition, for the ability to scrape, drill, heal and annotate,
three color-channels must be stored per voxel increasing the required memory foot-
print. Furthermore, to prevent feedback loop issues, front and back volumes must
be stored. While this large memory space is limiting at present, it is important to
note that this method removes unneeded data transfers and is an effective way for
modifying and visualizing mass amounts of data as long as this condition is met,
as the results section demonstrates.

Given the feature set of current graphics cards this constraint appears to be
tolerable when working with most mainstream volume data set sizes. There also is
evidence that with the current growth rate of graphics card texture memory and
alternate memory architectures, this issue may soon be mute. As shown in the
results section below, as long as the data does fit into the graphics card memory,
the method described in this paper scales in linear fashion.

One of the major limitations of this approach comes from the implementa-
tion of the three-dimensional framebuffer objects. Three-dimensional framebuffer
objects can be randomly written to in the x and y plane, but only for a given at-
tached z slice. This is limiting for the implementation of how data can be modified
and proves to be a major bottleneck in terms of performance. To render the entire
volume, every single z-slice must be attached and drawn into.

One common optimization of volumetric data visualization is to run a pre-
processing step to pre-compute needed data structures. Attributes such as normals,
geometry, and volume hierarchies can all be computed a priori and lead to quicker
processing for drawing and the ability to visualize data sets which are too large to
fit into texture memory. Since the objective was to allow the user to swiftly and
continuously manipulate the data, these pre-optimizations are not helpful. Addition-
ally, volume hierarchies such as oct-trees and kd-trees do not fit well into the
render-to-texture model.

Commonly, when render times cannot keep up with the desired frame rate,
a preview example is rendered and shown until the full rendering can be completed.
This can be done by either using less samples in the raster x and y directions, or by sampling the ray less along the z-in direction. This optimization unfortunately would greatly reduce the effectiveness and intuitiveness of the system described in this paper. The primary reason for this is that as soon as users attempt to manipulate the volume, the render engine may be forced to switch to a lower resolution preview, with visual artifacts of this preview including incorrect object transparency and physical position. These effects are incredibly unnatural and jarring, which is why this optimization was not chosen.

7.5 Applications

The approach described in this paper has many applications in the fields of volume analysis and sculpting with applications in medicine and non-destructive evaluation, to name a few. Volume sculpting is usually done with only a single point of interaction. By allowing multiple points of interaction of various sizes, volume sculpting can be done in accelerated and more intuitive fashion. The presented approach also allows for data analysis to be done in localized regions. Generally, volumetric data interrogation is only done on a global scale. While iso-surface values can be modified, they are changed over the entire volume. The presented system allows for localized areas to be modified without effecting surrounding regions.

Medical professionals have shown a great deal of interest and enthusiasm in this approach. By taking CT scans of a patient, a surgeon for example, can flexibly reveal the anatomy, practice cuts, label areas of interest and conceptualize three-dimensional relationships of topological features and different tissue types. This interactive and realistic representation could also be used to demonstrate the procedure in front of the patient in ways the patient could easily understand. Plastic surgeons could also use this visualization to model modifications on a realistic representation of the patient based on previous medical scans.
7.6 Results

Incision’s hands-on techniques for the analysis of arbitrary sized regions in volumetric data sets were evaluated through an initial user survey and a set of performance tests of small, medium and larger sized baseline samples, including the CT volume of a human head, an MRI scan of the upper body, and a computer generated volume, respectively. The derived performance metric distinguishes between volume manipulation and volume rendering.

7.6.1 Initial User Testing

Users were asked to identify how internal volume anomalies are correlated to surface features for a set of baseline cases. Anomalies consisted of regions which were filled with abnormally high density values and regions which were filled with abnormally low density values. Users were instructed to locate these internal abnormalities underneath a grid, etched on the boundary of the volume model. For each trial the size of the abnormality decreased, from an initial size of $10^3$ voxels, to $5^3$ voxels to finally a size of $3^3$ voxels.

![Average Time Per Trial](image)

**Figure 7.6**: Average time taken to complete each trial for the initial user study.

As a comparison to the Incision system, MeVisLab was selected, which
implements both VTK and ITK in a graphic user environment [KSR+06]. Its volume renderer with enabled cutting planes was selected as comparison basis, as it most closely matches the capabilities of the Incision system.

Figure 7.7: User ratings comparing Incision to volume rendering via MeVisLab. Black bars indicate the maximum and minimum score for the given question.

Six users were selected which had little experience on either system. This group consisted of two females and four males all ages twenty to forty. As show in Figure 7.6 users were able to accomplish the given task using the Incision system in less then a third of the time required for MeVisLab. Furthermore as users became more accustomed to the Incision system, the time required to finish the required task decreased further, even with increasing task difficulty. In comparison, the
time required for Trial 3 of MeVisLab was approximately 70% greater than the time required for Trial 1.

Furthermore in qualitative ratings, the Incision system scored better in every category compared to MeVisLab as shown in Figure 7.7. While one participant ranked Incision worse for user interaction, in all other categories Incision was considered to be better than MeVisLab. Furthermore, in every category Incision received at least one vote that it was a better than MeVisLab and when locating small anomalies Incision received a unanimous affirmation that it was the better system to use.

7.6.2 Volume Modification

The common approach of modifying volumetric data is to modify the data on the CPU and then upload the changes to the GPU in order to visualize the results. This approach works well for input devices that set a 3D position and modify small amounts of data. This setup is comprised of three steps, changing the volume data in the CPU, filling in a data structure for upload, and finally uploading the data to the GPU. Figure 7.8 shows the time required to change and upload data using this traditional CPU-centric technique in contrast to the GPU-centric approach described in this paper.

As shown, the GPU approach has a fixed cost when changing the entire volume, which is at the same complexity of changing an individual voxel. This is due to the fact that the entire volume is rendered in the render-to-texture pass. While optimizations could be made to only re-render the desired z-slices to be modified, the time it would take to calculate which z-slices needed to be rendered would take longer than the time to render the entire volume itself.

The major bottleneck in the GPU approach was attaching the z-slices. As shown in Figure 7.8, this enables this GPU approach to scale linearly with the number of depth-planes for the volume.

Conversely, the CPU approach scales proportionally to the number of voxels in the data set. While this approach is actually faster than the GPU in cases where only a small amount of data is changed, for larger amounts of data this approach
quickly becomes incredibly time consuming as shown in Figure 7.8. It should also be noted that the changed data must fit inside of the axis aligned bounding box, which encompasses the changes. This means that even if only a moderate amount of data needs to be updated, if the data is spread out, a bounding volume as large as the distribution would need be uploaded, causing detrimental performance.

### 7.6.3 Visualization

The visualization pipeline scales as a function of the number of rays and the sampling rate of the rays. Table 7.1 shows the frame rates for three differently sized volumetric data sets with various sample rates. As expected, the higher the sampling rate, the higher the quality of the result, and the lower the frame rate. For general use, $256^3$ sized volume sets could easily perform at highly interactive rates (greater than 60 Hz) as shown with a single voxel per sample.
Table 7.1: Performance of various sampling rates

<table>
<thead>
<tr>
<th>Volume Size: 256x256x240</th>
<th>Samples per Voxel</th>
<th>1</th>
<th>4</th>
<th>8</th>
<th>64</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>FPS</td>
<td>63</td>
<td>42</td>
<td>32</td>
<td>12</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Volume Size: 512x512x460</th>
<th>Samples per Voxel</th>
<th>1</th>
<th>4</th>
<th>8</th>
<th>64</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>FPS</td>
<td>12</td>
<td>7</td>
<td>5</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Volume Size: 817x817x512</th>
<th>Samples per Voxel</th>
<th>1</th>
<th>4</th>
<th>8</th>
<th>64</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>FPS</td>
<td>7</td>
<td>5</td>
<td>2.5</td>
<td>0.7</td>
</tr>
</tbody>
</table>

7.7 Conclusion

This chapter presents a method for intuitive, hands-on analysis of volumetric data, using simple metaphors. Interactive rendering rates are achieved by utilizing the massive parallelism of the GPU for multi-pass rendering, supporting dynamically changing visuals. The write-to-volume capability provided by three dimensional framebuffer objects is used to create the support structure for in-core modifications directly on the GPU, allowing large amounts of data to be modified at fixed cost. Multi-pass, multi-targeted rendering supporting effects such as shading and shadowing enhances visual depth cues and overall realism.

Hands-on volume analysis is enabled by a custom-built multi-touch system with a touch vocabulary and grammar, allowing users to freely transform the target volume, drill, scrape, annotate and restore volumetric data types on localized regions. Position, gesture and pressure information are used for voxel density and depth specific operations. With all interaction metaphors at most one touch or hand-gesture away, users were able to begin analyzing data with no or limited self-training and were at large proficient within seconds or minutes.
7.8 Acknowledgments

This chapter is currently being prepared for submission for publication of the material. Ponto, K., Doerr, K., and Kuester, F. The dissertation author was the primary investigator and author of this paper.
Bibliography


[STA] C. STANDARD. THE MPEG VIDEO COMPRESSION STANDARD.


[VOT]


[wms]